Títol:	Implementació d'algorismes eficients per resoldre problemes matemàtics
Autors:	Pol Mauri Ruiz i Félix Miravé Carreño
Data:	28 de Juny de 2012
Director:	Salvador Roura Ferret
Departament del director:	Llenguatges i Sistemes Informàtics
Titulació:	Enginyeria en Informàtica
Centre:	Facultat d'Informàtica de Barcelona
Universitat:	Universitat Politècnica de Catalunya

Índex

1 Introducció **6**

1.1 Motivacions . . . 6
1.2 Desenvolupament i implementació de les solucions . . . 6
1.3 Correctesa de les solucions . . . 7

2 UVa Online Judge **8**

UVa 1018. Building Bridges . . . 9
UVa 1020. Riding the Bus . . . 14
UVa 1022. Covering Whole Holes . . . 19
UVa 1023. Combining Images . . . 26
UVa 1025. A Spy in the Metro . . . 31
UVa 1030. Image Is Everything . . . 35
UVa 1032. Intersecting Dates . . . 39
UVa 1059. Jacquard Circuits . . . 44
UVa 1061. Consanguine Calculations . . . 49
UVa 1062. Containers . . . 54
UVa 1064. Network . . . 56
UVa 1068. Air Conditioning Machinery . . . 60
UVa 1069. Always an integer . . . 64
UVa 1073. Glenbow Museum . . . 68
UVa 1076. Password Suspects . . . 71
UVa 1078. Steam Roller . . . 75
UVa 1079. A Careful Approach . . . 80
UVa 1080. My Bad . . . 84
UVa 1083. Fare and Balanced . . . 90
UVa 1086. The Ministers' Major Mess . . . 95
UVa 1087. Struts and Springs . . . 100
UVa 1089. Suffix-Replacement Grammars . . . 105
UVa 1092. Tracking Bio-bots . . . 109
UVa 1093. Castles . . . 113
UVa 1096. The Islands . . . 117
UVa 1099. Sharing Chocolate . . . 121
UVa 1103. Ancient Messages . . . 124
UVa 1105. Coffee Central . . . 129
UVa 1107. Magic Sticks . . . 133
UVa 1110. Pyramids . . . 136
UVa 12302. Nine-Point Circle . . . 140
UVa 12307. Smallest Enclosing Rectangle . . . 142
UVa 12350. Queen Game . . . 147
UVa 12424. Answering Queries on a Tree . . . 150
UVa 12425. Best Friend . . . 155
UVa 12427. Donkey of the Sultan . . . 159
UVa 12428. Enemy at the Gates . . . 162
UVa 12429. Finding Magic Triplets . . . 164
UVa 12430. Grand Wedding . . . 167
UVa 12435. Consistent Verdicts . . . 171
UVa 12436. Rip Van Winkle's Code . . . 175

UVa 12437. Kisu Pari Na 2 . . . 179
UVa 12439. February 29 . . . 183
UVa 12440. Save the Trees . . . 186
UVa 12441. Superb Sequence . . . 189
UVa 12442. Forwarding Emails . . . 194

3 ACM-ICPC Live Archive 196
ACM-ICPC Live Archive 5780. The Agency . . . 197
ACM-ICPC Live Archive 5782. Condorcet Winners . . . 200
ACM-ICPC Live Archive 5784. The Banzhaf Buzz-Off . . . 202
ACM-ICPC Live Archive 5785. GPS I Love You . . . 206
ACM-ICPC Live Archive 5788. Wally World . . . 208
ACM-ICPC Live Archive 5851. Gift from the Goddess of Programming . . . 210
ACM-ICPC Live Archive 5852. The Sorcerer's Donut . . . 214
ACM-ICPC Live Archive 5854. Long Distance Taxi . . . 220
ACM-ICPC Live Archive 5855. Driving an Icosahedral Rover . . . 226
ACM-ICPC Live Archive 5856. City Merger . . . 232
ACM-ICPC Live Archive 5857. Captain Q's Treasure . . . 236
ACM-ICPC Live Archive 5860. Round Trip . . . 244
ACM-ICPC Live Archive 5881. Unique Encryption Keys . . . 251
ACM-ICPC Live Archive 5888. Stack Machine Executor . . . 254
ACM-ICPC Live Archive 5889. Good or Bad? . . . 259
ACM-ICPC Live Archive 5892. Collateral Cleanup . . . 262
ACM-ICPC Live Archive 5894. Lightning Lessons . . . 266
ACM-ICPC Live Archive 5896. Speed Racer . . . 269
ACM-ICPC Live Archive 5897. The Status is Not Quo . . . 271
ACM-ICPC Live Archive 5899. Tree Count . . . 276
ACM-ICPC Live Archive 5902. Movie collection . . . 279
ACM-ICPC Live Archive 5903. Piece it together . . . 282
ACM-ICPC Live Archive 5904. Please, go first . . . 287
ACM-ICPC Live Archive 5906. Smoking gun . . . 290

4 Timus Online Judge 295
Timus 1100. Final Standings . . . 296
Timus 1101. Robot in the Field . . . 298
Timus 1102. Strange Dialog . . . 302
Timus 1104. Don't Ask Woman about Her Age . . . 305
Timus 1105. Observers Coloring . . . 307
Timus 1106. Two Teams . . . 310
Timus 1107. Warehouse Problem . . . 313
Timus 1108. Heritage . . . 315
Timus 1109. Conference . . . 318
Timus 1110. Power . . . 321
Timus 1111. Squares . . . 323
Timus 1112. Cover . . . 327
Timus 1114. Boxes . . . 330
Timus 1115. Ships . . . 332
Timus 1116. Piecewise Constant Function . . . 334
Timus 1117. Hierarchy . . . 337
Timus 1118. Nontrivial Numbers . . . 339

Timus 1119. Metro . . . 341
Timus 1120. Sum of Sequential Numbers . . . 343
Timus 1121. Branches . . . 345
Timus 1122. Game . . . 347
Timus 1124. Mosaic . . . 350
Timus 1125. Hopscotch . . . 352
Timus 1126. Magnetic Storms . . . 354
Timus 1127. Colored Bricks . . . 356
Timus 1128. Partition into Groups . . . 359
Timus 1129. Door Painting . . . 362
Timus 1133. Fibonacci Sequence . . . 365
Timus 1134. Cards . . . 371
Timus 1135. Recruits . . . 373
Timus 1136. Parliament . . . 375
Timus 1137. Bus Routes . . . 377
Timus 1138. Integer Percentage . . . 380
Timus 1139. City Blocks . . . 382
Timus 1140. Swamp Incident . . . 384
Timus 1142. Relations . . . 386
Timus 1143. Electric Path . . . 388
Timus 1145. Rope in the Labyrinth . . . 390
Timus 1146. Maximum Sum . . . 393
Timus 1147. Shaping Regions . . . 395
Timus 1148. Building Towers . . . 399
Timus 1150. Page Numbers . . . 402
Timus 1151. Radiobeacons . . . 404
Timus 1152. False Mirrors . . . 407
Timus 1154. Mages Contest . . . 409
Timus 1155. Troubleduons . . . 412
Timus 1156. Two Rounds . . . 415
Timus 1157. Young Tiler . . . 419
Timus 1158. Censored! . . . 421
Timus 1159. Fence . . . 426
Timus 1160. Network . . . 429
Timus 1161. Stripies . . . 431
Timus 1162. Currency Exchange . . . 433
Timus 1164. Fillword . . . 435
Timus 1165. Subnumber . . . 437
Timus 1167. Bicolored Horses . . . 444
Timus 1168. Radio Stations . . . 446
Timus 1169. Pairs . . . 448
Timus 1171. Lost in Space . . . 450
Timus 1172. Ship Routes . . . 454
Timus 1173. Lazy Snail . . . 459
Timus 1174. Weird Permutations . . . 462
Timus 1175. Strange Sequence . . . 468
Timus 1176. Hyperchannels . . . 470
Timus 1178. Akbardin's Roads . . . 472
Timus 1179. Numbers in Text . . . 474

Timus 1180. Stone Game 476
Timus 1182. Team Them Up! 478
Timus 1183. Brackets Sequence 482
Timus 1184. Cable Master 485
Timus 1186. Chemical Reactions 487
Timus 1249. Ancient Necropolis 491
Timus 1250. Sea Burial 493
Timus 1252. Sorting the Tombstones 496
Timus 1253. Necrologues 498
Timus 1255. Graveyard of the Cosa Nostra 501
Timus 1275. Knights of the Round Table 503
Timus 1276. Train 506
Timus 1278. "... Connecting People" 509
Timus 1279. Warehouse 511
Timus 1282. Game Tree 513
Timus 1453. Queen 516
Timus 1456. Jedi Riddle 2 519
Timus 1457. Heating Main 523
Timus 1486. Equal Squares 525
Timus 1542. Autocompletion 528
Timus 1544. Classmates 3 532
Timus 1547. Password Search 535
Timus 1549. Another Japanese Puzzle 538
Timus 1551. Sumo Tournament 541
Timus 1552. Brainfuck 543
Timus 1553. Caves and Tunnels 546
Timus 1554. Multiplicative Functions 550
Timus 1560. Elementary Symmetric Functions 552
Timus 1584. Pharaohs' Secrets 555
Timus 1585. Penguins 559
Timus 1586. Threeprime Numbers 561
Timus 1588. Jamaica 563
Timus 1590. Bacon's Cipher 566
Timus 1591. Abstract Thinking 569
Timus 1888. Pilot Work Experience 571
Timus 1891. Language Ocean 575
Timus 1893. A380 578

5 Anàlisi Econòmica **580**

6 Conclusions **580**
6.1 Resultat a la final de l'ICPC 580
6.2 Visió de futur 580

7 Bibliografia **581**

1 Introducció

1.1 Motivacions

Els dos autors d'aquest projecte hem dedicat durant els darrers anys una part important del nostre temps a resoldre problemes de caire algorísmic o matemàtic mitjançant la programació. Això és en gran part conseqüència de la participació de la Universitat Politècnica de Catalunya en l'ACM-ICPC (Association for Computing Machinery - International Collegiate Programming Contest), en el qual competeixen equips de 3 estudiants de diferents universitats. En el cas de la UPC, el professor Salvador Roura, del departament de Llenguatges i Sistemes Informàtics, organitza una competició interna que consisteix en resoldre individualment problemes de programació d'estil similar als de l'ICPC. Els 9 primers classificats participen, dividits en 3 equips de 3 persones, al SWERC (South Western European Regional Contest), que és la fase regional de l'ICPC. L'equip guanyador del SWERC es classifica per participar a la final mundial de l'ICPC, en la qual competeixen els millors equips de cada regional. En el SWERC 2011/2012 el guanyador va ser l'equip de la UPC format per Marc Vinyals i els dos autors d'aquest projecte, de manera que ens vam classificar per participar en la fase final. Tenint això en ment, a més del nostre interès pel tema, vam decidir dedicar una gran part del nostre temps durant els 4 mesos previs a la competició a resoldre problemes de tipus matemàtic, algorísmic i de programació.

En aquest document recollim gairebé tots els problemes que vam resoldre durant els 4 mesos previs a la final mundial de l'ICPC; concretament, de cada problema en presentem el seu enunciat i un programa (escrit en C++ o Java) que el resol, juntament amb un petit comentari explicatiu precedint al codi. Considerem que aquesta feina, a més de servir-nos als autors per millorar com a programdors i també en l'àmbit de la resolució de problemes matemàtics i algorísmics, pot ser útil per a futures generacions de participants de l'ICPC en forma de document de consulta. Per a molts dels problemes que presentem és difícil, si no impossible, trobar per internet idees de com resoldre'ls, i per això hem donat importància al comentari explicatiu que precedeix al codi. També creiem, però, que sovint no és trivial passar d'una solució en paraules o en notació matemàtica a un programa que resolgui el problema, i per aquest motiu hem considerat essencial presentar un codi font corresponent a la solució de cada problema.

1.2 Desenvolupament i implementació de les solucions

Les 173 solucions presentades en aquest document han estat desenvolupades per nosaltres mateixos, possiblement amb la col·laboració de Marc Vinyals o del professor Salvador Roura en alguns casos. La implementació també l'hem dut a terme nosaltres mateixos excepte en fer servir codis que resolen tasques concretes, implementats anteriorment per algú altre, com per exemple la classe BigInt per treballar amb enters de milers de dígits (implementada pel professor Salvador Roura), l'algorisme de Dinic per resoldre el problema de Max Flow (implementat per Alex Alvarez), o el codi per construir eficientment l'arbre de sufixos d'una string (implementat per Josep Àngel Herrero).

Principalment hem fet servir el llenguatge de programació C++ perquè té una sèrie d'avantatges que el fan el més adequat per les nostres necessitats:

- Té un conjunt de biblioteques standard (Standard Template Library) que permeten al programador no haver d'implementar les estructures de dades i funcions bàsiques, i permeten que el codi d'un programa complex sigui relativament curt i senzill.

- El funcionament de l'entrada i sortida de text és senzill, la qual cosa és notable al resoldre problemes d'aquest tipus perquè sempre hi ha una entrada i una sortida.

- La compilació d'un programa en C++ genera un executable que serà interpretat directament pel processador, a diferència d'altres llenguatges com Java en els quals el resultat de la compilació és interpretat per una màquina virtual, de manera que en general un mateix programa trigarà menys temps en ser executat si està escrit en C++ que si està escrit en Java.
- Les seves instruccions de codi tenen una traducció gairebé directa a llenguatge màquina, de manera que es té un gran control sobre el comportament del programa.

Tot i així, en dos dels problemes (Timus 1453 i Timus 1547) ens ha resultat més pràctic fer servir el llenguatge de programació Java, perquè les seves biblioteques standard inclouen una classe per treballar amb enters de milers de dígits, mentre que la STL de C++ no en té. Tot i que podríem haver fet servir la classe BigInt que ja teníem implementada, en una competició seria una pèrdua de temps i una potencial font d'errors implementar-la des de zero tenint la possibilitat de programar en Java, i és per això que hem fet servir Java com a exercici.

1.3 Correctesa de les solucions

Cada any es realitza arreu del món un gran nombre de competicions de programació, i la major part de les competicions més destacades (Codeforces, Google Code Jam, ICPC, TopCoder entre altres) segueixen un mateix mètode per avaluar si un programa resol el problema. Bàsicament, es té un conjunt extens de possibles entrades pel programa amb les seves corresponents sortides correctes, i el programa s'executa amb cada una de les possibles entrades. Si per cada entrada el programa produeix una sortida igual a la sortida correcta, sense sobrepassar les limitacions de memòria i temps d'execució, es considera que el programa resol el problema.

Existeix uns "jutges online" que contenen problemes de competicions ja finalitzades, als quals es pot enviar en qualsevol moment del dia i qualsevol dia de l'any, sense límit d'enviaments, un intent de solució d'un problema, i el jutge avalua si el programa resol el problema o no, tot indicant quina és la causa de que no el resolgui (el programa triga massa, dóna una resposta errònia, fa servir massa memòria, etc.) en cas de no resoldre'l. Els problemes presentats en aquest document estan disponibles o bé al UVa Online Judge, o al Timus Online Judge, o al ACM-ICPC Live Archive, i hem considerat que el veredicte de "Accepted" donat per aquests jutges online confirma la correctesa de les nostres solucions.

2 UVa Online Judge

L'UVa Online Judge és un jutge online creat el 1995 per Miguel Ángel Revilla, de la Universidad de Valladolid. Aquest jutge és mundialment conegut i compta amb més de 100000 usuaris registrats d'arreu del món. Conté una col·lecció de més de 3500 problemes de diverses competicions de programació, entre els quals s'hi pot trobar tots els problemes de finals mundials de l'ICPC des de 1990, i també milers de problemes que originalment es van fer servir en competicions locals de diverses universitats del món.

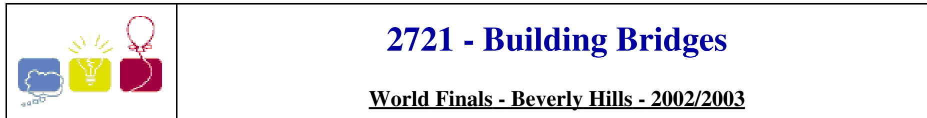
De tant en tant, en aquest jutge s'hi fan concursos online en temps real, és a dir, concursos en què hi ha una col·lecció reduïda de problemes (habitualment al voltant de 10) i els participants poden enviar solucions durant un interval de temps determinat (normalment de 5 hores de duració), amb unes normes similars a les de l'ICPC. Els participants poden veure el progrés dels altres participants (temps que han trigat a resoldre cada problema, quantitat d'intents que han necessitat, etc.). Aquestes competicions són obertes a tothom i no existeix cap tipus de premi, es fan amb la finalitat de servir com a entrenament.

El jutge accepta solucions en 4 llenguatges de programació diferents: C, C++, Java i Pascal. Els programes que són enviats com a solució no poden fer crides a sistema o fer servir biblioteques que no siguin estàndard, com és usual en tots aquests jutges online.

La pàgina web del jutge disposa d'un fòrum on els usuaris poden parlar sobre els problemes, compartir idees, discutir algorismes, demanar ajuda, etc. També existeixen fòrums externs a la UVa on es parla de problemes d'aquest jutge i de similars, com per exemple el fòrum de TopCoder.

A continuació presentem els enunciats i solucions dels 46 problemes que vam resoldre de l'UVa Online Judge.

UVa 1018. Building Bridges



2721 - Building Bridges

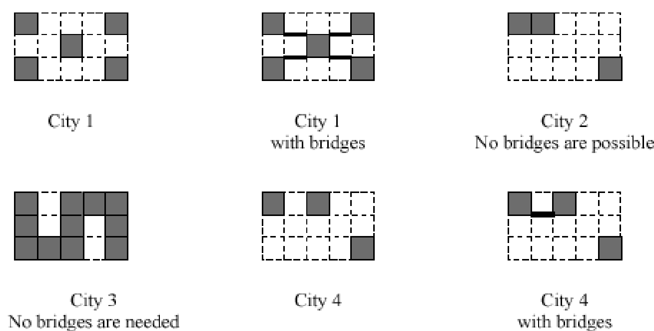
World Finals - Beverly Hills - 2002/2003

The City Council of New Altonville plans to build a system of bridges connecting all of its downtown buildings together so people can walk from one building to another without going outside. You must write a program to help determine an optimal bridge configuration.

New Altonville is laid out as a grid of squares. Each building occupies a connected set of one or more squares. Two occupied squares whose corners touch are considered to be a single building and do not need a bridge. Bridges may be built only on the grid lines that form the edges of the squares. Each bridge must be built in a straight line and must connect exactly two buildings.

For a given set of buildings, you must find the minimum number of bridges needed to connect all the buildings. If this is impossible, find a solution that minimizes the number of disconnected groups of buildings. Among possible solutions with the same number of bridges, choose the one that minimizes the sum of the lengths of the bridges, measured in multiples of the grid size. Two bridges may cross, but in this case they are considered to be on separate levels and do not provide a connection from one bridge to the other.

The figure below illustrates four possible city configurations. City 1 consists of five buildings that can be connected by four bridges with a total length of 4. In City 2, no bridges are possible, since no buildings share a common grid line. In City 3, no bridges are needed because there is only one building. In City 4, the best solution uses a single bridge of length 1 to connect two buildings, leaving two disconnected groups (one containing two buildings and one containing a single building).



Input

The input data set describes several rectangular cities. Each city description begins with a line containing two integers r and c, representing the size of the city on the north-south and east-west axes measured in grid lengths ($1 \le r \le 100$ and $1 \le c \le 100$). These numbers are followed by exactly r lines, each consisting of c hash (`#') and dot (`.') characters. Each character corresponds to one square of the grid. A hash character corresponds to a square that is occupied by a building, and a dot character corresponds to a square that is not occupied by a building.

The input data for the last city will be followed by a line containing two zeros.

Output

For each city description, print two or three lines of output as shown below. The first line consists of the city number. If the city has fewer than two buildings, the second line is the sentence `No bridges are needed.'. If the city has two or more buildings but none of them can be connected by bridges, the second line is the sentence `No bridges are possible.'. Otherwise, the second line is `*N* bridges of total length *L*' where *N* is the number of bridges and *L* is the sum of the lengths of the bridges of the best solution. (If *N* is 1, use the word `bridge' rather than `bridges.') If the solution leaves two or more disconnected groups of buildings, print a third line containing the number of disconnected groups.

Print a blank line between cases. Use the output format shown in the example.

Sample Input

```
3 5
#...#
..#..
#...#
3 5
##...
.....
....#
3 5
#.###
#.#.#
###.#
3 5
#.#..
.....
....#
0 0
```

Sample Output

```
City 1
4 bridges of total length 4

City 2
No bridges are possible.
2 disconnected groups

City 3
No bridges are needed.

City 4
1 bridge of total length 1
2 disconnected groups
```

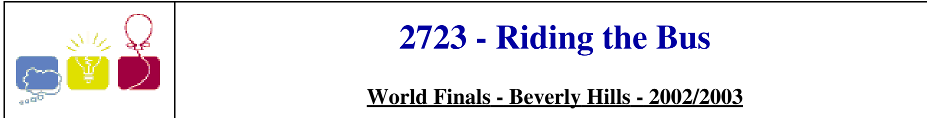
Beverly Hills 2002-2003

```
/*
UVa 1018. Building Bridges
Build a graph where each node corresponds to a building in the map.
For each possible bridge that can be constructed, put an edge
between the two corresponding nodes, with the length of the bridge
as its weight. Put edges with cost 0 between every pair of touching
buildings. Now, find a Minimum Spanning Tree in each connected
component (with Kruskal's algorithm, for example) that minimizes
the total length of the bridges. Note that the number of bridges
in each connected component if fixed and minimal by the fact that
we are finding a tree, while the total length has to be minimized.
The "disconnected groups" in the problem correspond to the
connected components in the afresaid graph.
*/
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

#define PB push_back
#define X first
#define Y second

typedef pair<int, int> P;
typedef pair<int, P> PP;
typedef vector<int> Vi;
typedef vector<PP> Vpp;

int R, C;
string mapa[110];
Vi tree;

int root(int n) {
  if (tree[n] == -1) return n;
  return tree[n] = root(tree[n]);
}

inline int tonum(int x, int y) {
  return x*C + y;
}

int main() {
  int cas = 1;
  while (cin >> R >> C and R > 0) {
    for (int i = 0; i < R; ++i) cin >> mapa[i];

    int comp = 0;
    Vpp ares;
    for (int i = 0; i < R; ++i)
      for (int j = 0; j < C; ++j) {
        if (mapa[i][j] == '.') continue;

        ++comp;
        int st = tonum(i, j);

        for (int k = j - 1; k >= 0; --k)
          if (mapa[i][k] == '#' or (i + 1 < R and mapa[i + 1][k] == '#')) {
            if (mapa[i][k] == '.') ares.PB(PP(j - k - 1, P(st, tonum(i + 1, k))));
            else ares.PB(PP(j - k - 1, P(st, tonum(i, k))));
            break;
```

```
        }

      for (int k = j - 1; k >= 0; --k) {
        if (mapa[i][k] == '#' or (i - 1 >= 0 and mapa[i - 1][k] == '#')) {
          if (mapa[i][k] == '.') ares.PB(PP(j - k - 1, P(st, tonum(i - 1, k))));
          else ares.PB(PP(j - k - 1, P(st, tonum(i, k))));
          break;
        }
      }

      for (int k = j + 1; k < C; ++k) {
        if (mapa[i][k] == '#' or (i + 1 < R and mapa[i + 1][k] == '#')) {
          if (mapa[i][k] == '.') ares.PB(PP(k - j - 1, P(st, tonum(i + 1, k))));
          else ares.PB(PP(k - j - 1, P(st, tonum(i, k))));
          break;
        }
      }

      for (int k = j + 1; k < C; ++k) {
        if (mapa[i][k] == '#' or (i - 1 >= 0 and mapa[i - 1][k] == '#')) {
          if (mapa[i][k] == '.') ares.PB(PP(k - j - 1, P(st, tonum(i - 1, k))));
          else ares.PB(PP(k - j - 1, P(st, tonum(i, k))));
          break;
        }
      }

      for (int k = i - 1; k >= 0; --k) {
        if (mapa[k][j] == '#' or (j + 1 < C and mapa[k][j + 1] == '#')) {
          if (mapa[k][j] == '.') ares.PB(PP(i - k - 1, P(st, tonum(k, j + 1))));
          else ares.PB(PP(i - k - 1, P(st, tonum(k, j))));
          break;
        }
      }

      for (int k = i - 1; k >= 0; --k) {
        if (mapa[k][j] == '#' or (j - 1 >= 0 and mapa[k][j - 1] == '#')) {
          if (mapa[k][j] == '.') ares.PB(PP(i - k - 1, P(st, tonum(k, j - 1))));
          else ares.PB(PP(i - k - 1, P(st, tonum(k, j))));
          break;
        }
      }

      for (int k = i + 1; k < R; ++k) {
        if (mapa[k][j] == '#' or (j + 1 < C and mapa[k][j + 1] == '#')) {
          if (mapa[k][j] == '.') ares.PB(PP(k - i - 1, P(st, tonum(k, j + 1))));
          else ares.PB(PP(k - i - 1, P(st, tonum(k, j))));
          break;
        }
      }

      for (int k = i + 1; k < R; ++k) {
        if (mapa[k][j] == '#' or (j - 1 >= 0 and mapa[k][j - 1] == '#')) {
          if (mapa[k][j] == '.') ares.PB(PP(k - i - 1, P(st, tonum(k, j - 1))));
          else ares.PB(PP(k - i - 1, P(st, tonum(k, j))));
          break;
        }
      }
    }

  sort(ares.begin(), ares.end());
  int m = ares.size();
```

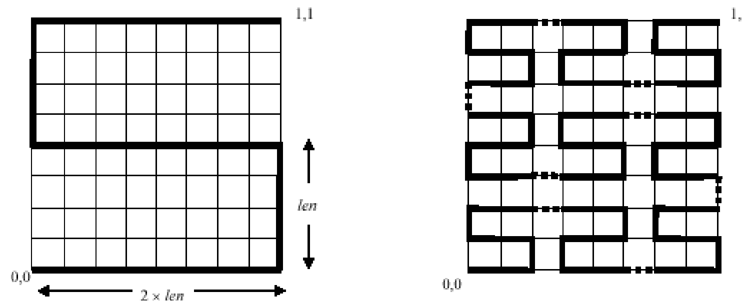
```
    tree = Vi(R*C, -1);
    int cost = 0, bridges = 0;
    for (int i = 0; i < m and comp > 1; ++i) {
      int a = ares[i].Y.X;
      int b = ares[i].Y.Y;
      int c = ares[i].X;
      int ra = root(a), rb = root(b);
      if (ra != rb) {
        tree[rb] = ra;
        cost += c;
        --comp;
        if (c) ++bridges;
      }
    }

    if (cas > 1) cout << endl;
    cout << "City " << cas++ << endl;
    if (bridges == 0) {
      if (comp <= 1) cout << "No bridges are needed." << endl;
      else {
        cout << "No bridges are possible." << endl;
        cout << comp << " disconnected groups" << endl;
      }
    }
    else {
      cout << bridges << " " << (bridges == 1 ? "bridge" : "bridges") << " of total
          length " << cost << endl;
      if (comp > 1) cout << comp << " disconnected groups" << endl;
    }
  }
}
```

UVa 1020. Riding the Bus



The latest research in reconfigurable multiprocessor chips focuses on the use of a single bus that winds around the chip. Processor components, which can be anywhere on the chip, are attached to *connecting points* on the bus so that they can communicate with each other.

Some research involves bus layout that uses recursively-defined ``SZ" curves, also known as ``S-shaped Peano curves." Two examples of these curves are shown below. Each curve is drawn on the unit square. The order-1 curve, shown on the left, approximates the letter ``S" and consists of line segments connecting the points (0,0), (1,0), (1,0.5), (0,0.5), (0,1), and (1,1) in order. Each horizontal line in an ``S" or ``Z" curve is twice as long as each vertical line. For the order-1 curve, the length of a vertical line, *len*, is 0.5.



The order-2 curve, shown on the right, contains 9 smaller copies of the order-1 curve (4 of which are reversed left to right to yield ``Z" curves). These copies are connected by line segments of length *len*, shown as dotted lines. Since the width and height of the order-2 curve is 8×*len*, and the curve is drawn on the unit square, *len* = 0.125 for the order-2 curve.

The order-3 curve contains 9 smaller copies of the order-2 curve (with 4 reversed left to right), connected by line segments, as described for the order-2 curve. Higher order curves are drawn in a similar manner. The *connecting points* to which processor components attach are evenly spaced every *len* units along the bus. The first connecting point is at (0,0) and the last is at (1,1). There are 9^k connecting points along the order-*k* curve, and the total bus length is $(9^k - 1)\times len$ units.

You must write a program to determine the total distance that signals must travel between two processor components. Each component's coordinates are given as an *x*, *y* pair, $0 \le x \le 1$ and $0 \le y \le 1$, where *x* is the distance from the left side of the chip, and *y* is the distance from the lower edge of the chip. Each component is attached to the closest connecting point by a straight line. If multiple connecting points are equidistant from a component, the one with the smallest *x* coordinate and smallest *y* coordinate is used. The total distance a signal must travel between two components is the sum of the length of the lines connecting the components to the bus, and the length of the bus between the two connecting points. For example, the distance between components located at (0.5, 0.25) and (1.0, 0.875) on a chip using the order-1 curve is 3.8750 units.

Input

The input contains several cases. For each case, the input consists of an integer that gives the order of the SZ curve used as the bus (no larger than 8), and then four real numbers x_1, y_1, x_2, y_2 that give the coordinates of the processor components to be connected. While each processor component should actually be in a unique location not on the bus, your program must correctly handle all possible locations.

The last case in the input is followed by a single zero.

Output

For each case, display the case number (starting with 1 for the first case) and the distance between the processor components when they are connected as described. Display the distance with 4 digits to the right of the decimal point.

Use the same format as that shown in the sample output shown below. Leave a blank line between the output lines for consecutive cases.

Sample Input

```
1 0.5 .25 1 .875
1 0 0 1 1
2 .3 .3 .7 .7
2 0 0 1 1
0
```

Sample Output

```
Case 1.  Distance is 3.8750

Case 2.  Distance is 4.0000

Case 3.  Distance is 8.1414

Case 4.  Distance is 10.0000
```

Beverly Hills 2002-2003

```
/*
UVa 1020. Riding the Bus
Despite the fact that the component location is given as real coordinates,
find the closest grid point and work with that one. Use integer numbers
for the coordinates inside the grid. Find, for each point, the distance
from the origin to that point, following the path defined by the bus
(find this distance as the number of grid edges, and scale at the end
if necessary). The case of the order-1 curve is pretty easy to solve.
In order to solve the case of an order-k curve (k>1), first find in
which of the 9 order-(k-1) sub-curves the point is located, find
recursively the distance travelled inside that sub-curve (adapting the
coordinates properly), and add the length of the path from the origin
to the beginning of that sub-curve to the result.
*/
#include <iostream>
#include <cmath>
using namespace std;

#define X first
#define Y second

typedef long long ll;
typedef long double ld;
typedef pair<ll, ll> P;

const ld EPS = 1e-9;

const int res1[3][3] = {
  { 0, 5, 6 },
  { 1, 4, 7 },
  { 2, 3, 8 }
};

ll pow3[20], pow9[20];

ll fun(int k, ll x, ll y) {
  if (k == 1) return res1[x][y];

  ll t = pow3[k - 1] - 1;
  ll p = pow9[k - 1] - 1;

  if (y <= t) {
    if (x <= t) return fun(k - 1, x, y); //1
    else if (x <= 2*t + 1) return p + 1 + fun(k - 1, x - t - 1, t - y); //2
    else return 2*p + 2 + fun(k - 1, x - 2*t - 2, y); //3
  }
  else if (y <= 2*t + 1) {
    if (x >= 2*t + 2) return 3*p + 3 + fun(k - 1, 3*t + 2 - x, y - t - 1); //4
    else if (x >= t + 1) return 4*p + 4 + fun(k - 1, 2*t + 1 - x, 2*t + 1 - y); //5
    else return 5*p + 5 + fun(k - 1, t - x, y - t - 1); //6
  }
  else {
    if (x <= t) return 6*p + 6 + fun(k - 1, x, y - 2*t - 2); //7
    else if (x <= 2*t + 1) return 7*p + 7 + fun(k - 1, x - t - 1, 3*t + 2 - y); //8
    else return 8*p + 8 + fun(k - 1, x - 2*t - 2, y - 2*t - 2); //9
  }
}

ld dist2(ld x1, ld y1, ld x2, ld y2) {
  ld x = x1 - x2;
  ld y = y1 - y2;
```

```
  return x*x + y*y;
}

ld dist(ld x1, ld y1, ld x2, ld y2) {
  return sqrt(dist2(x1, y1, x2, y2));
}

P puntproper(int k, ld x, ld y) {
  ll t = pow3[k] - 1;
  ll nx = x, ny = y;

  ld d2 = dist2(x, y, nx, ny);
  P res(nx, ny);

  P v[3];
  int s = 0;
  if (ny + 1 <= t) v[s++] = P(nx, ny + 1);
  if (nx + 1 <= t) v[s++] = P(nx + 1, ny);
  if (nx + 1 <= t and ny + 1 <= t) v[s++] = P(nx + 1, ny + 1);

  for (int i = 0; i < s; ++i) {
    ld tmp = dist2(x, y, v[i].X, v[i].Y);
    if (tmp + EPS < d2) {
      d2 = tmp;
      res = v[i];
    }
  }
  return res;
}

int main() {
  cout.setf(ios::fixed);
  cout.precision(4);

  pow3[0] = pow9[0] = 1;
  for (int i = 1; i < 20; ++i) pow3[i] = 3*pow3[i - 1];
  for (int i = 1; i < 20; ++i) pow9[i] = 9*pow9[i - 1];

  int cas = 1;

  int k;
  while (cin >> k and k > 0) {
    ld x1, y1, x2, y2;
    cin >> x1 >> y1 >> x2 >> y2;

    ll t = pow3[k] - 1;
    x1 *= t;
    y1 *= t;
    x2 *= t;
    y2 *= t;

    P p1 = puntproper(k, x1, y1);
    P p2 = puntproper(k, x2, y2);

    ll d1 = fun(k, p1.X, p1.Y);
    ll d2 = fun(k, p2.X, p2.Y);

    ld t1 = dist(x1, y1, p1.X, p1.Y);
    ld t2 = dist(x2, y2, p2.X, p2.Y);

    if (cas > 1) cout << endl;
```

```
    cout << "Case " << cas++ << ".  Distance is " << (t1 + t2 + abs(d1 - d2))/ld(t)
        << endl;
  }
}
```

UVa 1022. Covering Whole Holes

2725 - Covering Whole Holes

World Finals - Beverly Hills - 2002/2003

Can you cover a round hole with a square cover? You can, as long as the square cover is big enough. It obviously will not be an exact fit, but it is still possible to cover the hole completely.

The Association of Cover Manufacturers (ACM) is a group of companies that produce covers for all kinds of holes - manholes, holes on streets, wells, ditches, cave entrances, holes in backyards dug by dogs to bury bones, to name only a few. ACM wants a program that determines whether a given cover can be used to completely cover a specified hole. At this time, they are interested only in covers and holes that are rectangular polygons (that is, polygons with interior angles of only 90 or 270 degrees). Moreover, both cover and hole are aligned along the same coordinate axes, and are not supposed to be rotated against each other - just translated relative to each other.

Input

The input consists of several descriptions of covers and holes. The first line of each description contains two integers h and c ($4 \le h \le 50$ and $4 \le c \le 50$), the number of points of the polygon describing the hole and the cover respectively. Each of the following h lines contains two integers x and y, which are the vertices of the hole's polygon in the order they would be visited in a trip around the polygon. The next c lines give a corresponding description of the cover. Both polygons are rectangular, and the sides of the polygons are aligned with the coordinate axes. The polygons have positive area and do not intersect themselves.

The last description is followed by a line containing two zeros.

Output

For each problem description, print its number in the sequence of descriptions. If the hole can be completely covered by moving the cover (without rotating it), print `Yes' otherwise print `No'. Recall that the cover may extend beyond the boundaries of the hole as long as no part of the hole is uncovered. Follow the output format in the example given below.

Sample Input

```
4 4
0 0
0 10
10 10
10 0
0 0
0 20
20 20
20 0
4 6
0 0
0 10
10 10
10 0
0 0
0 10
10 10
```

```
10 1
9 1
9 0
0 0
```

Sample Output

```
Hole 1: Yes
Hole 2: No
```

Beverly Hills 2002-2003

```
/*
UVa 1022. Covering Whole Holes
First of all, note that if the hole can be completely covered by the cover, we
can move the cover downwards until some horizontal edge of the cover overlaps
with some horizontal edge of the hole. Similarly, we can move the cover to the
right until some vertical edge of the cover overlaps with some horizontal edge
of the hole. This leads to a solution that consists in iterating over all pairs
of horizontal sides overlapping and for each of these pairs iterate over all
pairs of vertical sides overlapping, and for each of them check if the cover in
its current position covers the whole hole.

In order to check if, given a fixed vertical and horizontal translation, the
hole is covered by the cover, we start by compressing the coordinates of the
cover. Consider the matrix that, if we look at the plane as a grid made of unit
squares, contains 1s in the cells that are inside the cover, and 0s in the cells
that are outside the cover. Compressing coordinates means that, for every
segment of consecutive equal rows we only store one row and we keep track of
the interval of Y coordinates that have rows of this kind. Thus, the number of
rows we end up storing is about half the number of vertices of the poligon.
We perform the same compression in the other coordinate axis (only store one
column of each type). Now in each cell we store the sum of all the cells in the
original matrix that it represents. Note that every cell in the compressed
matrix will either be completely inside the cover or completely outside the
cover, which means that the number stored in a cell will be either 0 (outside)
or the area it represents (inside).

Next, we take the matrix with compressed coordinates C and build another matrix
SC with the same dimensions, which contains in cell (i, j) the sum of all the
cells (i', j') of C with i' <= i and j' <= j. Using this matrix, we can compute
the sum of any rectangle very efficiently using the inclusion-exclusion
principle. This also means that we can check very efficiently if a given
rectangle is completely inside the cover or not.

Finally, in order to check if a translated hole is covered by the cover we can
divide the hole into a set of rectangles such that their union equals the hole,
and check if every one of them is covered by the cover. One possible way to
divide the hole into rectangles is to compress its coordinates and then divide
the compressed matrix H into rectangles, for example by repeatedly taking the
topmost and leftmost unused corner, extending the rectangle vertically until
it touches an horizontal edge and then extending the rectangle horizontally
until it touches a vertical edge.
*/
#include <algorithm>
#include <iostream>
#include <map>
#include <set>
#include <vector>

#define X first
#define Y second
#define FR first
#define SC second

using namespace std;

typedef pair<int, int> PII;

const int dy[] = {1, 0, -1, 0};
const int dx[] = {0, 1, 0, -1};

inline int area_sum(vector<vector<int> >& sum, int x0, int y0, int x1, int y1) {
```

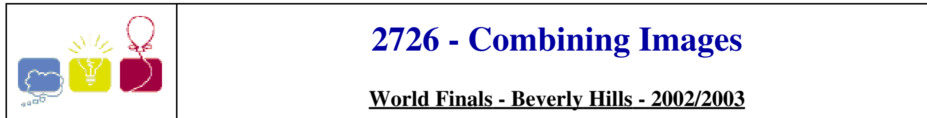
```
  return sum[y1+1][x1+1]-sum[y0][x1+1]-sum[y1+1][x0]+sum[y0][x0];
}

void flood(vector<vector<int> >& conn, vector<vector<bool> >& inside, int y, int x)
    {
  inside[y][x] = false;
  for (int dir = 0; dir < 4; ++dir) if ((conn[y][x]>>dir)&1) {
    int ny = y+dy[dir], nx = x+dx[dir];
    if (ny >= 0 && ny < int(inside.size()) && nx >= 0 && nx < int(inside[ny].size())
        ) {
      if (inside[ny][nx]) flood(conn, inside, ny, nx);
    }
  }
}

vector<vector<bool> > compress(vector<PII>& p, map<int, int>& mx, map<int, int>& my)
     {
  set<int> sx, sy;
  for (int i = 0; i < int(p.size()); ++i) {
    sx.insert(p[i].X);
    sy.insert(p[i].Y);
  }
  sx.insert(*sx.begin() - 1);
  sy.insert(*sy.begin() - 1);

  mx.clear();
  int xcnt = 0;
  for (set<int>::iterator it = sx.begin(); it != sx.end(); ++it) {
    mx[*it] = xcnt;
    ++xcnt;
  }
  my.clear();
  int ycnt = 0;
  for (set<int>::iterator it = sy.begin(); it != sy.end(); ++it) {
    my[*it] = ycnt;
    ++ycnt;
  }

  vector<vector<int> > conn(my.size(), vector<int>(mx.size(), 15));
  for (int k = 0; k < int(p.size()); ++k) {
    PII cur = p[k], nxt = p[(k+1)%int(p.size())];
    if (cur > nxt) swap(cur, nxt);
    if (cur.X == nxt.X) {
      for (int i = my[cur.Y]; i < my[nxt.Y]; ++i) {
        conn[i][mx[cur.X]-1] &= ~(1<<1);
        conn[i][mx[cur.X]] &= ~(1<<3);
      }
    }
    else { // cur.Y == nxt.Y
      for (int j = mx[cur.X]; j < mx[nxt.X]; ++j) {
        conn[my[cur.Y]-1][j] &= ~(1<<0);
        conn[my[cur.Y]][j] &= ~(1<<2);
      }
    }
  }
  vector<vector<bool> > inside(my.size(), vector<bool>(mx.size(), true));
  flood(conn, inside, 0, 0);
  return inside;
}

inline bool contained(vector<vector<int> >& csum, vector<int>& vmx, vector<int>& vmy
```

```
    , int x0, int y0, int x1, int y1) {
  int x0c = int(upper_bound(vmx.begin(), vmx.end(), x0)-vmx.begin())-1;
  if (x0c < 0) return false;

  int y0c = int(upper_bound(vmy.begin(), vmy.end(), y0)-vmy.begin())-1;
  if (y0c < 0) return false;

  int x1c = int(upper_bound(vmx.begin(), vmx.end(), x1)-vmx.begin())-1;

  int y1c = int(upper_bound(vmy.begin(), vmy.end(), y1)-vmy.begin())-1;

  return area_sum(csum, x0c, y0c, x1c, y1c) == (y1c-y0c+1)*(x1c-x0c+1);
}

int main() {
  for (int ca = 1, h, c; cin >> h >> c && (h > 0 || c > 0); ++ca) {

    vector<PII> ph(h);
    set<int> sxh, syh;
    for (int i = 0; i < h; ++i) {
      cin >> ph[i].X >> ph[i].Y;
      sxh.insert(ph[i].X);
      syh.insert(ph[i].Y);
    }
    vector<int> vxh(sxh.begin(), sxh.end()); // Xs at which the hole has some vertex
    vector<int> vyh(syh.begin(), syh.end()); // Ys at which the hole has some vertex

    vector<PII> pc(c);
    set<int> sxc, syc;
    for (int i = 0; i < c; ++i) {
      cin >> pc[i].X >> pc[i].Y;
      sxc.insert(pc[i].X);
      syc.insert(pc[i].Y);
    }
    vector<int> vxc(sxc.begin(), sxc.end()); // Xs at which the cover has some
        vertex
    vector<int> vyc(syc.begin(), syc.end()); // Ys at which the cover has some
        vertex

    map<int, int> mxh, myh, mxc, myc; // coordinate compression
    vector<vector<bool> > ch = compress(ph, mxh, myh), cc = compress(pc, mxc, myc);

    vector<vector<int> > chsum(ch.size()+1, vector<int>(ch[0].size()+1, 0)); // area
         sum of ch
    for (int y = 0; y < int(ch.size()); ++y) {
      for (int x = 0; x < int(ch[y].size()); ++x) {
        chsum[y+1][x+1] = chsum[y][x+1]+chsum[y+1][x]-chsum[y][x];
        if (ch[y][x]) ++chsum[y+1][x+1];
      }
    }

    vector<vector<int> > ccsum(cc.size()+1, vector<int>(cc[0].size()+1, 0)); // area
         sum of cc
    for (int y = 0; y < int(cc.size()); ++y) {
      for (int x = 0; x < int(cc[y].size()); ++x) {
        ccsum[y+1][x+1] = ccsum[y][x+1]+ccsum[y+1][x]-ccsum[y][x];
        if (cc[y][x]) ++ccsum[y+1][x+1];
      }
    }

    set<int> sty; // possible Y translation values
```

```
  for (int ih = 0; ih < int(vyh.size()); ++ih) {
    bool both = false, toph = false;
    for (int j = 0; j < int(ch[myh[vyh[ih]]].size()); ++j) {
      if (!ch[myh[vyh[ih]]-1][j] && ch[myh[vyh[ih]]][j]) {
        both = true;
      }
      if (ch[myh[vyh[ih]]-1][j] && !ch[myh[vyh[ih]]][j]) {
        toph = true;
      }
    }
    for (int ic = 0; ic < int(vyc.size()); ++ic) {
      bool botc = false, topc = false;
      for (int j = 0; j < int(cc[myc[vyc[ic]]].size()); ++j) {
        if (!cc[myc[vyc[ic]]-1][j] && cc[myc[vyc[ic]]][j]) {
          botc = true;
        }
        if (cc[myc[vyc[ic]]-1][j] && !cc[myc[vyc[ic]]][j]) {
          topc = true;
        }
      }
      if ((toph && topc) || (both && botc)) {
        sty.insert(vyc[ic]-vyh[ih]);
      }
    }
  }
  vector<int> vty(sty.begin(), sty.end());

  set<int> stx; // possible X translation values
  for (int jh = 0; jh < int(vxh.size()); ++jh) {
    bool lefth = false, righth = false;
    for (int i = 0; i < int(ch.size()); ++i) {
      if (!ch[i][mxh[vxh[jh]]-1] && ch[i][mxh[vxh[jh]]]) {
        lefth = true;
      }
      if (ch[i][mxh[vxh[jh]]-1] && !ch[i][mxh[vxh[jh]]]) {
        righth = true;
      }
    }
    for (int jc = 0; jc < int(vxc.size()); ++jc) {
      bool leftc = false, rightc = false;
      for (int i = 0; i < int(cc.size()); ++i) {
        if (!cc[i][mxc[vxc[jc]]-1] && cc[i][mxc[vxc[jc]]]) {
          leftc = true;
        }
        if (cc[i][mxc[vxc[jc]]-1] && !cc[i][mxc[vxc[jc]]]) {
          rightc = true;
        }
      }
      if ((lefth && leftc) || (righth && rightc)) {
        stx.insert(vxc[jc]-vxh[jh]);
      }
    }
  }
  vector<int> vtx(stx.begin(), stx.end());

  vector<int> rmxh, rmyh, rmxc, rmyc; // reverse coordinate compression
  for (map<int, int>::iterator it = mxh.begin(); it != mxh.end(); ++it) {
    rmxh.push_back(it->first);
  }
  for (map<int, int>::iterator it = myh.begin(); it != myh.end(); ++it) {
    rmyh.push_back(it->first);
```

```
    }
    for (map<int, int>::iterator it = mxc.begin(); it != mxc.end(); ++it) {
      rmxc.push_back(it->first);
    }
    for (map<int, int>::iterator it = myc.begin(); it != myc.end(); ++it) {
      rmyc.push_back(it->first);
    }

    vector<pair<PII, PII> > rect; // division of the hole into rectangles
    vector<int> seen_bm(ch.size(), 0); // 32-bit int used for each row (<= 25+1)
    for (int i = 0; i < int(ch.size()); ++i) {
      for (int j = 0; j < int(ch[i].size()); ++j) if (ch[i][j] && ((seen_bm[i]>>j)
          &1) == 0) {
        int width = 1;
        for (; area_sum(chsum, j, i, j+(width+1)-1, i) == width+1;) ++width;
        int height = 1;
        for (; area_sum(chsum, j, i, j+width-1, i+(height+1)-1) == width*(height+1)
            ;) ++height;
        for (int di = 0; di < height; ++di) {
          seen_bm[i+di] |= (((1<<width)-1)<<j);
        }
        rect.push_back(pair<PII, PII>(PII(rmxh[j], rmyh[i]), PII(rmxh[j+width]-1,
            rmyh[i+height]-1)));
      }
    }

    vector<int> vmyc;
    for (map<int, int>::iterator it = myc.begin(); it != myc.end(); ++it) {
      vmyc.push_back(it->first);
    }
    vector<int> vmxc;
    for (map<int, int>::iterator it = mxc.begin(); it != mxc.end(); ++it) {
      vmxc.push_back(it->first);
    }
    bool covered = false;
    for (int i = 0; i < int(vty.size()) && !covered; ++i) {
      for (int j = 0; j < int(vtx.size()) && !covered; ++j) {
        int ty = vty[i];
        int tx = vtx[j];
        covered = true;
        for (int k = 0; k < int(rect.size()) && covered; ++k) {
          if (!contained(ccsum, vmxc, vmyc, rect[k].FR.X+tx, rect[k].FR.Y+ty, rect[k
              ].SC.X+tx, rect[k].SC.Y+ty)) {
            covered = false;
          }
        }
      }
    }
    cout << "Hole " << ca << ": ";
    if (covered) cout << "Yes" << endl;
    else cout << "No" << endl;
  }
}
```

UVa 1023. Combining Images



As the exchange of images over computer networks becomes more common, the problem of image compression takes on increasing importance. Image compression algorithms are used to represent images using a relatively small number of bits.

One image compression algorithm is based on an encoding called a ``Quad Tree." An image has a Quad Tree encoding if it is a square array of binary pixels (the value of each pixel is 0 or 1, called the ``color" of the pixel), and the number of pixels on the side of the square is a power of two.

If an image is homogeneous (all its pixels are of the same color), the Quad Tree encoding of the image is 1 followed by the color of the pixels. For example, the Quad Tree encoding of an image that contains pixels of color 1 only is 11, regardless of the size of the image.

If an image is heterogeneous (it contains pixels of both colors), the Quad Tree encoding of the image is 0 followed by the Quad Tree encodings of its upper-left quadrant, its upper-right quadrant, its lower-left quadrant, and its lower-right quadrant, in order.

The Quad Tree encoding of an image is a string of binary digits. For easier printing, a Quad Tree encoding can be converted to a Hex Quad Tree encoding by the following steps:

a. Prepend a 1 digit as a delimiter on the left of the Quad Tree encoding.

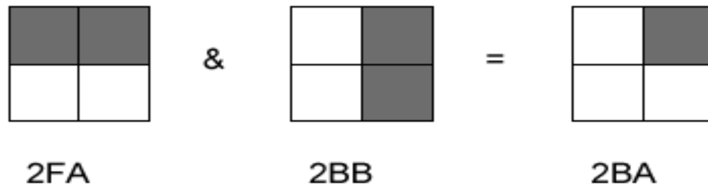
b. Prepend 0 digits on the left as necessary until the number of digits is a multiple of four.

c. Convert each sequence of four binary digits into a hexadecimal digit, using the digits 0 to 9 and capital A through F to represent binary patterns from 0000 to 1111.

For example, the Hex Quad Tree encoding of an image that contains pixels of color 1 only is 7, which corresponds to the binary string 0111.

You must write a program that reads the Hex Quad Tree encoding of two images, computes a new image that is the intersection of those two images, and prints its Hex Quad Tree encoding. Assume that both input images are square and contain the same number of pixels (although the lengths of their encodings may differ). If two images A and B have the same size and shape, their intersection (written as A & B) also has the same size and shape. By definition, a pixel of A & B is equal to 1 if and only if the corresponding pixels of image A and image B are both equal to 1.

The following figure illustrates two input images and their intersection, together with the Hex Quad Tree encodings of each image. In the illustration, shaded squares represent pixels of color 1.



Input

The input data set contains a sequence of test cases, each of which is represented by two lines of input. In each test case, the first input line contains the Hex Quad Tree encoding of the first image and the second line contains the Hex Quad Tree encoding of the second image. For each input image, the number of hexadecimal digits in its Hex Quad Tree encoding will not exceed 100.

The last test case is followed by two input lines, each containing a single zero.

Output

For each test case, print `Image' followed by its sequence number. On the next line, print the Hex Quad Tree encoding of the intersection of the two images for that test case. Separate the output for consecutive test cases with a blank line.

Sample Input

```
2FA
2BB
2FB
2EF
7
2FA
0
0
```

Sample Output

```
Image 1:
2BA

Image 2:
2EB

Image 3:
2FA
```

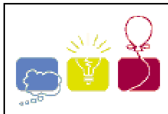
Beverly Hills 2002-2003

```
/*
UVa 1023. Combining Images
Use a tree structure to save and handle the Quad-Trees.
First parse the input and save both trees. Then, merge
the two structures recursively, having in mind the following
case: When, at some point, all the four children of a given
node define homogenic white images, then that node must not
have any children and define the white image itself.
After merging the two trees, output the result in the
specified format.
*/
#include <iostream>
#include <map>
#include <string>
using namespace std;

#define X first
#define Y second

typedef map<string, string> MAP;
typedef MAP::iterator Mit;

struct ST {
  int homo, cont;
  ST *a, *b, *c, *d;
};

MAP tobin, tohex;
string S;
int M;

ST* parse() {
  ST *res = new ST;
  if (S[M] == '0') {
    ++M;
    res->homo = 0;
    res->a = parse();
    res->b = parse();
    res->c = parse();
    res->d = parse();
  }
  else {
    ++M;
    res->homo = 1;
    res->cont = S[M] - '0';
    ++M;
  }
  return res;
}

ST* fun(ST *t1, ST *t2) {
  if (t1->homo) {
    if (t1->cont == 0) return t1;
    return t2;
  }
  if (t2->homo) {
    if (t2->cont == 0) return t2;
    return t1;
  }
  ST *res = new ST;
  res->homo = 0;
```

```
  res->a = fun(t1->a, t2->a);
  res->b = fun(t1->b, t2->b);
  res->c = fun(t1->c, t2->c);
  res->d = fun(t1->d, t2->d);
  if (res->a->homo and res->a->cont == 0 and
      res->b->homo and res->b->cont == 0 and
      res->c->homo and res->c->cont == 0 and
      res->d->homo and res->d->cont == 0)
    res = res->a;
  return res;
}

void tostring(ST *t) {
  if (t->homo) {
    S += "1";
    S += char('0' + t->cont);
    return;
  }
  S += "0";
  tostring(t->a);
  tostring(t->b);
  tostring(t->c);
  tostring(t->d);
}

int main() {
  tobin["0"] = "0000";
  tobin["1"] = "0001";
  tobin["2"] = "0010";
  tobin["3"] = "0011";
  tobin["4"] = "0100";
  tobin["5"] = "0101";
  tobin["6"] = "0110";
  tobin["7"] = "0111";
  tobin["8"] = "1000";
  tobin["9"] = "1001";
  tobin["A"] = "1010";
  tobin["B"] = "1011";
  tobin["C"] = "1100";
  tobin["D"] = "1101";
  tobin["E"] = "1110";
  tobin["F"] = "1111";
  for (Mit it = tobin.begin(); it != tobin.end(); ++it)
    tohex[it->Y] = it->X;

  int cas = 1;

  string a, b;
  while (cin >> a >> b and a != "0") {
    int na = a.size(), nb = b.size();
    string sa, sb;
    for (int i = 0; i < na; ++i) sa += tobin[a.substr(i, 1)];
    for (int i = 0; i < nb; ++i) sb += tobin[b.substr(i, 1)];

    S = sa;
    M = 0;
    while (S[M] == '0') ++M;
    ++M;
    ST *t1 = parse();

    S = sb;
```

```
    M = 0;
    while (S[M] == '0') ++M;
    ++M;
    ST *t2 = parse();

    ST *t3 = fun(t1, t2);
    S.clear();
    tostring(t3);

    S = "1" + S;
    while (S.size()%4 != 0) S = "0" + S;

    if (cas > 1) cout << endl;
    cout << "Image " << cas++ << ":" << endl;

    int n = S.size();
    for (int i = 0; i < n; i += 4)
      cout << tohex[S.substr(i, 4)];
    cout << endl;
  }
}
```

UVa 1025. A Spy in the Metro



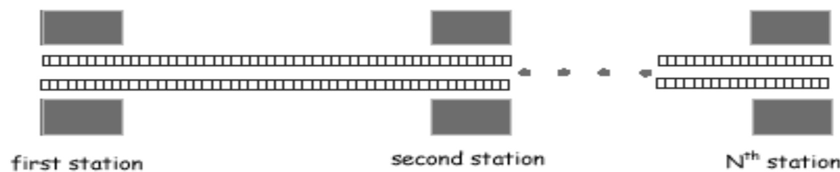
2728 - A Spy in the Metro

World Finals - Beverly Hills - 2002/2003

Secret agent Maria was sent to Algorithms City to carry out an especially dangerous mission. After several thrilling events we find her in the first station of Algorithms City Metro, examining the time table. The Algorithms City Metro consists of a single line with trains running both ways, so its time table is not complicated.

Maria has an appointment with a local spy at the last station of Algorithms City Metro. Maria knows that a powerful organization is after her. She also knows that while waiting at a station, she is at great risk of being caught. To hide in a running train is much safer, so she decides to stay in running trains as much as possible, even if this means traveling backward and forward. Maria needs to know a schedule with minimal waiting time at the stations that gets her to the last station in time for her appointment. You must write a program that finds the total waiting time in a best schedule for Maria.

The Algorithms City Metro system has N stations, consecutively numbered from 1 to N. Trains move in both directions: from the first station to the last station and from the last station back to the first station. The time required for a train to travel between two consecutive stations is fixed since all trains move at the same speed. Trains make a very short stop at each station, which you can ignore for simplicity. Since she is a very fast agent, Maria can always change trains at a station even if the trains involved stop in that station at the same time.



Input

The input file contains several test cases. Each test case consists of seven lines with information as follows.

Line 1.
The integer N ($2 \le N \le 50$), which is the number of stations.

Line 2.
The integer T ($0 \le T \le 200$), which is the time of the appointment.

Line 3.
N - 1 integers: $t_1, t_2, \ldots, t_{N-1}$ ($1 \le t_i \le 70$), representing the travel times for the trains between two consecutive stations: t_1 represents the travel time between the first two stations, t_2 the time between the second and the third station, and so on.

Line 4.
The integer $M1$ ($1 \le M1 \le 50$), representing the number of trains departing from the first station.

Line 5.

$M1$ integers: $d_1, d_2, \dots, d_{M1}$ ($0 \le d_i \le 250$ and $d_i < d_{i+1}$), representing the times at which trains depart from the first station.

Line 6.

The integer $M2$ ($1 \le M2 \le 50$), representing the number of trains departing from the N-th station.

Line 7.

$M2$ integers: $e_1, e_2, \dots, e_{M2}$ ($0 \le e_i \le 250$ and $e_i < e_{i+1}$) representing the times at which trains depart from the N-th station.

The last case is followed by a line containing a single zero.

Output

For each test case, print a line containing the case number (starting with 1) and an integer representing the total waiting time in the stations for a best schedule, or the word `impossible' in case Maria is unable to make the appointment. Use the format of the sample output.

Sample Input

```
4
55
5 10 15
4
0 5 10 20
4
0 5 10 15
4
18
1 2 3
5
0 3 6 10 12
6
0 3 5 7 12 15
2
30
20
1
20
7
1 3 5 7 11 13 17
0
```

Sample Output

```
Case Number 1: 5
Case Number 2: 0
Case Number 3: impossible
```

Beverly Hills 2002-2003

```
/*
UVa 1025. A Spy in the Metro
This problem can be solved using Dynamic Programming, defining a state (i, j)
which is the minimum waiting time to be at station i at time j. From a state
(i, j) the reachable states are (i, j+1), which corresponds to waiting one
time unit, (i-1, j+t[i-1]) if there is a train going to the left departing from
station i at time j which takes t[i-1] time units to reach the next station,
and (i+1, j+t[i]) if there is a train going to the right departing from station
i at time j which takes t[i] time units to reach the next station.
*/
#include <iostream>
#include <vector>

using namespace std;

const int INF = 1000000000;

int main() {
  for (int ca = 1, n; cin >> n && n > 0; ++ca) {
    int T;
    cin >> T;
    vector<int> t(n-1, INF);
    for (int i = 0; i < n-1; ++i) {
      cin >> t[i];
    }
    int m1;
    cin >> m1;
    vector<int> d(m1);
    for (int i = 0; i < m1; ++i) {
      cin >> d[i];
    }
    int m2;
    cin >> m2;
    vector<int> e(m2);
    for (int i = 0; i < m2; ++i) {
      cin >> e[i];
    }
    vector<vector<bool> > right(n, vector<bool>(T+1, false));
    for (int i = 0, dist = 0; i < n-1; ++i) {
      for (int j = 0; j < m1 && dist+d[j] <= T; ++j) {
        right[i][dist+d[j]] = true;
      }
      dist += t[i];
    }
    vector<vector<bool> > left(n, vector<bool>(T+1, false));
    for (int i = n-1, dist = 0; i > 0; --i) {
      for (int j = 0; j < m2 && dist+e[j] <= T; ++j) {
        left[i][dist+e[j]] = true;
      }
      dist += t[i-1];
    }

    vector<vector<int> > wait(T+1, vector<int>(n, INF));
    wait[0][0] = 0;
    for (int i = 0; i < T; ++i) {
      for (int j = 0; j < n; ++j) {
        if (i+1 <= T) {
          wait[i+1][j] = min(wait[i+1][j], wait[i][j]+1);
        }
        if (left[j][i]) {
          if (i+t[j-1] <= T) {
```

```
          wait[i+t[j-1]][j-1] = min(wait[i+t[j-1]][j-1], wait[i][j]);
        }
      }
      if (right[j][i]) {
        if (i+t[j] <= T) {
          wait[i+t[j]][j+1] = min(wait[i+t[j]][j+1], wait[i][j]);
        }
      }
    }
  }
  cout << "Case Number " << ca << ": ";
  if (wait[T][n-1] == INF) cout << "impossible" << endl;
  else cout << wait[T][n-1] << endl;
 }
}
```

UVa 1030. Image Is Everything

2995 - Image Is Everything

World Finals - Prague - 2003/2004

Your new company is building a robot that can hold small lightweight objects. The robot will have the intelligence to determine if an object is light enough to hold. It does this by taking pictures of the object from the 6 cardinal directions, and then inferring an upper limit on the object's weight based on those images. You must write a program to do that for the robot.

You can assume that each object is formed from an $N \times N \times N$ lattice of cubes, some of which may be missing. Each $1 \times 1 \times 1$ cube weighs 1 gram, and each cube is painted a single solid color. The object is not necessarily connected.

Input

The input for this problem consists of several test cases representing different objects. Every case begins with a line containing N, which is the size of the object ($1 \leq N \leq 10$). The next N lines are the different $N \times N$ views of the object, in the order front, left, back, right, top, bottom. Each view will be separated by a single space from the view that follows it. The bottom edge of the top view corresponds to the top edge of the front view. Similarly, the top edge of the bottom view corresponds to the bottom edge of the front view. In each view, colors are represented by single, unique capital letters, while a period (.) indicates that the object can be seen through at that location.

Input for the last test case is followed by a line consisting of the number 0.

Output

For each test case, print a line containing the maximum possible weight of the object, using the format shown below.

Sample Input

```
3
.R. YYR .Y. RYY .Y. .R.
GRB YGR BYG RBY GYB GRB
.R. YRR .Y. RRY .R. .Y.
2
ZZ ZZ ZZ ZZ ZZ ZZ
ZZ ZZ ZZ ZZ ZZ ZZ
0
```

Sample Output

```
Maximum weight: 11 gram(s)
Maximum weight: 8 gram(s)
```

Prague 2003-2004

Tests-Setter: Rujia Liu
Special Thanks: Yao Guan

```
/*
UVa 1030. Image Is Everything
Start with a NxNxN cube and apply the following process repeatedly
until no further change is made:
For each 1x1x1 cube remaining in the large one, look if the existence
of that 1x1x1 cube is contradicting any information about the 6
outside faces and, if so, remove that 1x1x1 cube. Look at the
function "fun()" for accurate details about this process.
At the end, the result is a maximal figure satisfying the constraints
given by the outside faces.
*/
#include <iostream>
using namespace std;

int N;
int cube[20][20][20];
int front[20][20], leftt[20][20], back[20][20], rightt[20][20], top[20][20], bottom
    [20][20];

bool check(int i, int j, int k, int di, int dj, int dk) {
  i += di; j += dj; k += dk;
  while (0 <= i and i < N and 0 <= j and j < N and 0 <= k and k < N) {
    if (cube[i][j][k]) return false;
    i += di; j += dj; k += dk;
  }
  return true;
}

int fun_front(int i, int j, int k) {
  if (check(i, j, k, 0, 0, -1)) return front[N - i - 1][j];
  return -2;
}

int fun_left(int i, int j, int k) {
  if (check(i, j, k, 0, -1, 0)) return leftt[N - i - 1][N - k - 1];
  return -2;
}

int fun_back(int i, int j, int k) {
  if (check(i, j, k, 0, 0, 1)) return back[N - i - 1][N - j - 1];
  return -2;
}

int fun_right(int i, int j, int k) {
  if (check(i, j, k, 0, 1, 0)) return rightt[N - i - 1][k];
  return -2;
}

int fun_top(int i, int j, int k) {
  if (check(i, j, k, 1, 0, 0)) return top[N - k - 1][j];
  return -2;
}

int fun_bottom(int i, int j, int k) {
  if (check(i, j, k, -1, 0, 0)) return bottom[k][j];
  return -2;
}

// retorna true si la info es consistent deixant el bloc viu
bool fun(int i, int j, int k) {
  int tmp[6];
```

```
  tmp[0] = fun_front(i, j, k);
  tmp[1] = fun_left(i, j, k);
  tmp[2] = fun_back(i, j, k);
  tmp[3] = fun_right(i, j, k);
  tmp[4] = fun_top(i, j, k);
  tmp[5] = fun_bottom(i, j, k);
  int q = -1;
  for (int x = 0; x < 6; ++x)
    if (tmp[x] != -2) {
      q = x;
      break;
    }
  if (q == -1) return true;
  for (int x = q + 1; x < 6; ++x) {
    if (tmp[x] != -2 and tmp[q] != tmp[x]) {
      return false;
    }
  }
  return tmp[q] != -1;
}

int main() {
  while (cin >> N and N > 0) {
    for (int i = 0; i < N; ++i) {
      // front
      for (int j = 0; j < N; ++j) {
        char c;
        cin >> c;
        if (c == '.') front[i][j] = -1;
        else front[i][j] = c - 'A';
      }

      // left
      for (int j = 0; j < N; ++j) {
        char c;
        cin >> c;
        if (c == '.') leftt[i][j] = -1;
        else leftt[i][j] = c - 'A';
      }

      // back
      for (int j = 0; j < N; ++j) {
        char c;
        cin >> c;
        if (c == '.') back[i][j] = -1;
        else back[i][j] = c - 'A';
      }

      // right
      for (int j = 0; j < N; ++j) {
        char c;
        cin >> c;
        if (c == '.') rightt[i][j] = -1;
        else rightt[i][j] = c - 'A';
      }

      // top
      for (int j = 0; j < N; ++j) {
        char c;
        cin >> c;
        if (c == '.') top[i][j] = -1;
```

```
        else top[i][j] = c - 'A';
      }

      // bottom
      for (int j = 0; j < N; ++j) {
        char c;
        cin >> c;
        if (c == '.') bottom[i][j] = -1;
        else bottom[i][j] = c - 'A';
      }
    }
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j)
        for (int k = 0; k < N; ++k)
          cube[i][j][k] = 1;

    bool ok = true;
    while (ok) {
      ok = false;
      for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
          for (int k = 0; k < N; ++k)
            if (cube[i][j][k] and not fun(i, j, k)) {
              cube[i][j][k] = 0;
              ok = true;
            }
    }

    int res = 0;
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j)
        for (int k = 0; k < N; ++k)
          res += cube[i][j][k];
    cout << "Maximum weight: " << res << " gram(s)" << endl;
  }
}
```

UVa 1032. Intersecting Dates

2997 - Intersecting Dates

World Finals - Prague - 2003/2004

A research group is developing a computer program that will fetch historical stock market quotes from a service that charges a fixed fee for each day's quotes that it delivers. The group has examined the collection of previously-requested quotes and discovered a lot of duplication, resulting in wasted money. So the new program will maintain a list of all past quotes requested by members of the group. When additional quotes are required, only quotes for those dates not previously obtained will be fetched from the service, thus minimizing the cost.

You are to write a program that d etermines when new quotes are required. Input for the program consists of the date ranges for which quotes have been requested in the past and the date ranges for which quotes are required. The program will then determine the date ranges for which quotes must be fetched from the service.

Input

There will be multiple input cases. The input for each case begins with two non-negative integers *NX* and *NR*, ($0 \le NX, NR \le 100$). *NX* is the number of existing date ranges for quotes requested in the past. *NR* is the number of date ranges in the incoming requests for quotes. Following these are *NX* + *NR* pairs of dates. The first date in each pair will be less than or equal to the second date in the pair. The first *NX* pairs specify the date ranges of quotes which have been requested and obtained in the past, and the next *NR* pairs specify the date ranges for which quotes are required.

Two zeroes will follow the input data for the last case.

Each input date will be given in the form *YYYYMMDD*. *YYYY* is the year (1700 to 2100), *MM* is the month (01 to 12), and *DD* is the day (in the allowed range for the given month and year). Recall that months 04, 06, 09, and 11 have 30 days, months 01, 03, 05, 07, 08, 10, and 12 have 31 days, and month 02 has 28 days except in leap years, when it has 29 days. A year is a leap year if it is evenly divisible by 4 and is not a century year (a multiple of 100), or if it is divisible by 400.

Output

For each input case, display the case number (1, 2, ...) followed by a list of any date ranges for which quotes must be fetched from the service, one date range per output line. Use the American date format shown in the sample output below. Explicitly indicate (as shown) if no additional quotes must be fetched. If two date ranges are contiguous or overlap, then merge them into a single date range. If a date range consists of a single date, print it as a single date, not as a range consisting of two identical dates. Display the date ranges in chronological order, starting with the earliest date range.

Sample Input

```
1 1
19900101 19901231
19901201 20000131
0 3
19720101 19720131
19720201 19720228
19720301 19720301
1 1
```

```
20010101 20011231
20010515 20010901
0 0
```

Sample Output

```
Case 1:
    1/1/1991 to 1/31/2000

Case 2:
    1/1/1972 to 2/28/1972
    3/1/1972

Case 3:
    No additional quotes are required.
```

Prague 2003-2004

Tests-Setter: Rujia Liu
Special Thanks: Yao Guan

```
/*
UVa 1032. Intersecting Dates
In order to solve this problem it is very useful to assign an integer number to
each day in such a way that if day A comes after day B, the integer assigned to
A is larger than the integer assigned to B. After this transformation, the
operations that have to be performed (interval merging, intersection and
removal) become fairly simple. Finally, reverse the transformation and output
dates in the appropiate format.
*/
#include <algorithm>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

#define ini first
#define fin second

using namespace std;

typedef pair<int, int> PII;

const int FIRST_DATE = 17000101;

vector<int> id, rid;

inline bool is_leap_year(int y) {
  return (y%4 == 0 && y%100 != 0) || y%400 == 0;
}

int days_in_month(int m, int y) {
  if (m == 1 || m == 3 || m == 5 || m == 7 || m == 8 || m == 10 || m == 12) {
    return 31;
  }
  if (m == 4 || m == 6 || m == 9 || m == 11) {
    return 30;
  }
  if (m == 2) {
    if (is_leap_year(y)) return 29;
    return 28;
  }
  return -1;
}

int next_date(int date) {
  int y = date/10000;
  int m = (date/100)%100;
  int d = date%100;
  ++d;
  if (d > days_in_month(m, y)) {
    d = 1;
    ++m;
    if (m == 13) {
      m = 1;
      ++y;
    }
  }
  return y*10000+m*100+d;
}

int get_id(int date) {
```

```
  return id[date-FIRST_DATE];
}

void merge_adjacent(vector<PII>& ranges) {
  vector<PII> res;
  for (int i = 0; i < int(ranges.size()); ++i) {
    if (res.size() == 0 || ranges[i].ini > res[int(res.size())-1].fin) {
      res.push_back(ranges[i]);
    }
    else {
      res[int(res.size())-1].fin = max(res[int(res.size())-1].fin, ranges[i].fin);
    }
  }
  ranges = res;
}

vector<PII> remove_existing(const vector<PII>& exist, vector<PII> required, int a,
    int b) {
  if (required.size() == 0) return required;
  if (a == b) return required;
  PII pivot = exist[(a+b)/2];
  vector<PII> left, right;
  for (int i = 0; i < int(required.size()); ++i) {
    PII cur = required[i];
    if (cur.ini < pivot.ini) {
      cur.fin = min(cur.fin, pivot.ini);
      left.push_back(cur);
    }
    cur = required[i];
    if (cur.fin > pivot.fin) {
      cur.ini = max(cur.ini, pivot.fin);
      right.push_back(cur);
    }
  }
  left = remove_existing(exist, left, a, (a+b)/2);
  right = remove_existing(exist, right, (a+b)/2+1, b);
  for (int i = 0; i < int(right.size()); ++i) {
    left.push_back(right[i]);
  }
  return left;
}

string date_to_american(int date) {
  int y = date/10000;
  int m = (date/100)%100;
  int d = date%100;
  stringstream ss;
  ss << m << "/" << d << "/" << y << endl;
  string ret;
  ss >> ret;
  return ret;
}

int main() {
  id = rid = vector<int>(5000000, -1);
  for (int date = FIRST_DATE, i = 0; date < 21010101; date = next_date(date), ++i) {
    id[date-FIRST_DATE] = i;
    rid[i] = date;
  }
  for (int nx, nr, ca = 1; cin >> nx >> nr && (nx > 0 || nr > 0); ++ca) {
    vector<PII> exist(nx), required(nr);
```

```
    for (int i = 0; i < nx; ++i) {
      cin >> exist[i].ini >> exist[i].fin;
      exist[i].ini = get_id(exist[i].ini);
      exist[i].fin = get_id(exist[i].fin);
      ++exist[i].fin;
    }
    for (int i = 0; i < nr; ++i) {
      cin >> required[i].ini >> required[i].fin;
      required[i].ini = get_id(required[i].ini);
      required[i].fin = get_id(required[i].fin);
      ++required[i].fin;
    }
    sort(exist.begin(), exist.end());
    merge_adjacent(exist);
    sort(required.begin(), required.end());
    merge_adjacent(required);
    vector<PII> ans = remove_existing(exist, required, 0, int(exist.size()));
    if (ca > 1) cout << endl;
    cout << "Case " << ca << ":" << endl;
    if (ans.size() == 0) {
      cout << "    No additional quotes are required." << endl;
    }
    else for (int i = 0; i < int(ans.size()); ++i) {
      cout << "    ";
      if (ans[i].fin-ans[i].ini == 1) {
        cout << date_to_american(rid[ans[i].ini]) << endl;
      }
      else {
        cout << date_to_american(rid[ans[i].ini]) << " to " << date_to_american(rid[
            ans[i].fin-1]) << endl;
      }
    }
  }
}
```

UVa 1059. Jacquard Circuits

2395 - Jacquard Circuits

World Finals - Tokyo - 2006/2007

The eccentric sculptor Albrecht Caravaggio Mondrian has been inspired by the history of the computer to create works of art that he calls ``Jacquard circuits." Each of his works of art consists of a series of polygonal circuit boards (defined below), all having the same shape but at different scales, joined together with wire or string into a three-dimensional tiered structure. Figure 1 below shows an example with two levels and a pentagonal shape.

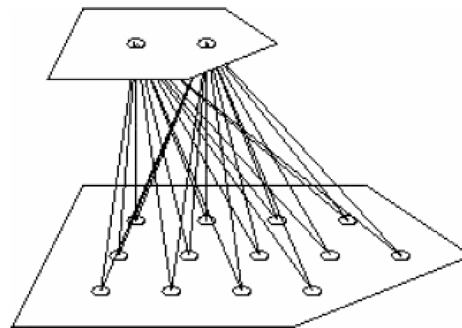


Figure 1

A.C.M. (as he is known to his friends) bases his art upon the punched hole cards of the Jacquard loom (that were later adapted by Charles Babbage for his analytical engine) and upon the regular grid layout approach often used in circuit interfacing (for instance, Pin Grid Arrays).

The circuit boards used in the sculptures are in the shapes of lattice polygons. A lattice polygon is defined as any closed, non-self-intersecting cycle of straight line segments that join points with integer coordinates, with no two consecutive line segments parallel. For any given lattice polygon P , there is a smallest lattice polygon with the same shape and orientation as P -- call this the origin. Smaller polygons of the same shape and orientation as P are called its predecessors and polygons larger than P are its successors. (See Figure 2 on the following page.)

To build one of his sculptures, A.C.M. begins by randomly selecting N lattice points and then drawing a pattern by connecting these points. (Note that not all of the N points are necessarily vertices of a lattice polygon. This may happen, for example, if three or more consecutive points are collinear.) Let P be the lattice polygon determined by the pattern. Then he determines the origin corresponding to P , selects the number of levels, M , he wishes to use, and constructs the first M polygonal circuit boards in the series (that is, the origin and its first M - 1 successors). Each lattice point lying strictly within the boundary of any of these polygons is a hole where strings or wires meet.

The hard part of creating the sculpture is tying together all the strings or wires that meet at a given hole. Furthermore, some of A.C.M.'s famous miniaturized sculptures, built using nano-engineering techniques, involve hundreds of thousands of levels. Mondrian would like a way to determine, given a polygonal shape and the number of levels, how many holes there will be in the final sculpture. You must write a program to help him. (For example, the sculpture in Figure 1 above has 15 holes.) Assume holes have zero diameter.

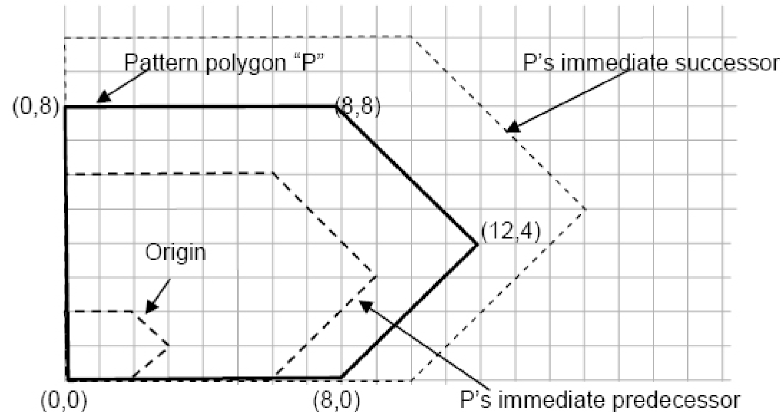


Figure 2

Input

The input consists of one or more test cases. Each case begins with a line containing two positive integers N ($3 \le N \le 1000$) and M ($1 \le M \le 1000000$) . N is the number of lattice points Mondrian selected and M is the number of levels in the completed sculpture. Each of the next N lines contains two integers x , y ($|x|,|y| \le 1000000$) which denote the coordinates of one of Mondrian's points. Points are listed in either clockwise or counterclockwise order.

The last test case is followed by a line containing two zeroes.

Output

For each test case, print a line containing the test case number (beginning with 1) followed by the number of holes in an M -level sculpture starting from the origin polygon of the given pattern. In no case will this value exceed the maximum possible value of a 64-bit signed integer.

Sample Input

```
5 2
0 0
8 0
12 4
8 8
0 8
3 2
-1 -1
3 1
5 -1
0 0
```

Sample Output

```
Case 1: 15
Case 2: 2
```

Tokyo 2006-2007

Tests-Setter: Shahriar Manzoor

```cpp
/*
UVa 1059. Jacquard Circuits
Choose an arbitrary point of the polygon, let's call it the origin O. Now, for
each point P_i different than the origin, consider the vector from the origin to
that point: P_i - O = (a_i, b_i). Calculate the value
g = gcd(a_1, ..., a_N, b_1, ..., b_N). Now scale the polygon by a factor of 1/g,
this way we are finding the smallest polygon such that every point is at integer
coordinates. This is the first polygon. The second polygon can be obtained by
scaling the first polygon by a factor of 2. The third, scaling the first one by
a factor of 3, and so on.
The number of integer interior points can be obtained with Pick's formula,
which is: i = A - b/2 + 1, where 'i' is the number of interior lattice points,
'b' is the number of boundary lattice points, and 'A' is the area of the polygon.
Both the boundary points and the area can be easily calculated.
Find the formula which gives the number of interior points of every polygon (from
the first to the M-th). Then, find the closed formula of the sum of the points
for the first M polygons.
Be careful while handling the given polygon, it may have aligned points.
Also, you must be very careful while computing the numeric result to avoid
overflows.
*/
#include <iostream>
#include <vector>
using namespace std;

#define PB push_back
#define X first
#define Y second

typedef long long ll;
typedef pair<ll, ll> P;
typedef vector<P> Vp;

ll gcd(ll a, ll b) {
  if (b == 0) return a;
  return gcd(b, a%b);
}

ll cross(P a, P b, P c) {
  return (c.X - b.X)*(a.Y - b.Y) - (c.Y - b.Y)*(a.X - b.X);
}

Vp demanzoorize(Vp v) {
  int n = v.size();
  int p = -1;
  for (int i = 0; p == -1 and i < n; ++i) {
    int i1 = (i + 1)%n;
    int i2 = (i + 2)%n;
    if (cross(v[i], v[i1], v[i2]) != 0) p = i1;
  }

  Vp u;
  int i = 0;
  while (i < n) {
    u.PB(v[i]);
    int q = 1;
    while (cross(v[i], v[(i + q)%n], v[(i + q + 1)%n]) == 0) ++q;
    i += q;
  }
  return u;
}
```

```
int main() {
  int cas = 1;
  int n;
  ll m;
  while (cin >> n >> m and n > 0) {
    Vp v(n);
    for (int i = 0; i < n; ++i) cin >> v[i].X >> v[i].Y;

    v = demanzoorize(v);
    n = v.size();

    ll a = 0;
    ll s = 0;
    ll g = 0;
    for (int i = 0; i < n; ++i) {
      int j = (i + 1 < n ? i + 1 : 0);
      ll t = gcd(abs(v[j].X - v[i].X), abs(v[j].Y - v[i].Y));
      g = gcd(g, t);
      s += t;
      a += v[i].X*v[j].Y - v[j].X*v[i].Y;
    }
    a = abs(a);

    a /= g*g;
    s /= g;

    ll m1 = m*(m + 1)/2;
    ll m2 = (2*m + 1);


    ll t1 = (a*m2 - 3*s);
    ll t2 = m1;
    if (t1%2 == 0) t1 /= 2;
    else t2 /= 2;
    if (t1%3 == 0) t1 /= 3;
    else t2 /= 3;
    ll r = t1*t2 + m;

    cout << "Case " << cas++ << ": " << r << endl;
  }
}
```

UVa 1061. Consanguine Calculations

3736 - Consanguine Calculations

World Finals - Tokyo - 2006/2007

Every person's blood has 2 markers called ABO alleles. Each of the markers is represented by one of three letters: A, B, or O. This gives six possible combinations of these alleles that a person can have, each of them resulting in a particular ABO blood type for that person.

Combination	ABO Blood Type
AA	A
AB	AB
AO	A
BB	B
BO	B
OO	O

Likewise, every person has two alleles for the blood Rh factor, represented by the characters + and −. Someone who is ``Rh positive" or ``Rh+" has at least one + allele, but could have two. Someone who is ``Rh negative" always has two − alleles.

The blood type of a person is a combination of ABO blood type and Rh factor. The blood type is written by suffixing the ABO blood type with the + or − representing the Rh factor. Examples include A+, AB−, and O−.

Blood types are inherited: each biological parent donates one ABO allele (randomly chosen from their two) and one Rh factor allele to their child. Therefore 2 ABO alleles and 2 Rh factor alleles of the parents determine the child's blood type. For example, if both parents of a child have blood type A−, then the child could have either type A− or type O− blood. A child of parents with blood types A+ and B+ could have any blood type.

In this problem, you will be given the blood type of either both parents or one parent and a child; you will then determine the (possibly empty) set of blood types that might characterize the child or the other parent.

Note: an uppercase letter ``Oh" is used in this problem to denote blood types, not a digit (zero).

Input

The input consists of multiple test cases. Each test case is on a single line in the format: the blood type of one parent, the blood type of the other parent, and finally the blood type of the child, except that the blood type of one parent or the child will be replaced by a question mark. To improve readability, whitespace may be included anywhere on the line except inside a single blood type specification.

The last test case is followed by a line containing the letters E, N, and D separated by whitespace.

Output

For each test case in the input, print the case number (beginning with 1) and the blood type of the parents and the child. If no blood type for a parent is possible, print ``IMPOSSIBLE". If multiple blood types for parents or child are possible, print all possible values in a comma-separated list enclosed in curly braces. The order of the blood types inside the curly braces does not matter.

The sample output illustrates multiple output formats. Your output format should be similar.

Sample Input

```
O+  O-  ?
O+  ?  O-
AB-  AB+  ?
AB+  ?  O+
E N D
```

Sample Output

```
Case 1: O+ O- {O+, O-}
Case 2: O+ {A-, A+, B-, B+, O-, O+} O-
Case 3: AB- AB+ {A+, A-, B+, B-, AB+, AB-}
Case 4: AB+ IMPOSSIBLE O+
```

Tokyo 2006-2007

Tests-Setter: Derek Kisman

```
/*
UVa 1061. Consanguine Calculations
This problem is quite straightforward, its only difficulty
lies in writing the output in the proper format, and making
a good interpretation of the rules specified in the statemnt.
*/
#include <iostream>
#include <map>
#include <set>
#include <vector>
using namespace std;

#define PB push_back

typedef map<string, string> MAP;
typedef set<string> SET;
typedef SET::iterator Sit;
typedef vector<string> Vs;

MAP comb;
MAP blood;

string getlet(string s) {
  int n = s.size();
  string t;
  for (int i = 0; i < n; ++i)
    if (s[i] != '+' and s[i] != '-')
      t += s[i];
  return t;
}

string getsig(string s) {
  int n = s.size();
  string t;
  for (int i = 0; i < n; ++i)
    if (s[i] == '+' or s[i] == '-')
      t += s[i];
  return t;
}

Vs fun(string a, string b) {
  string ta = getlet(a), tb = getlet(b);
  string sa = getsig(a), sb = getsig(b);

  Vs signs;
  if (sa == "-" and sb == "-") signs.PB("-");
  else  { signs.PB("+"); signs.PB("-"); }

  string aa;
  if (ta == "A") aa = "AO";
  else if (ta == "B") aa = "BO";
  else if (ta == "AB") aa = "AB";
  else aa = "O";

  string bb;
  if (tb == "A") bb = "AO";
  else if (tb == "B") bb = "BO";
  else if (tb == "AB") bb = "AB";
  else bb = "O";

  SET bloods;
```

```
  for (int i = 0; i < int(aa.size()); ++i)
    for (int j = 0; j < int(bb.size()); ++j) {
      string c(1, aa[i]);
      c += bb[j];
      bloods.insert(blood[c]);
    }

  Vs result;
  for (Sit it = bloods.begin(); it != bloods.end(); ++it)
    for (int i = 0; i < int(signs.size()); ++i)
      result.PB((*it) + signs[i]);
  return result;
}

int main() {
  comb["A"] = "AO";
  comb["B"] = "BO";
  comb["AB"] = "AB";
  comb["O"] = "O";

  blood["AA"] = "A";
  blood["AB"] = "AB";
  blood["AO"] = "A";
  blood["BA"] = "AB";
  blood["BB"] = "B";
  blood["BO"] = "B";
  blood["OA"] = "A";
  blood["OB"] = "B";
  blood["OO"] = "O";

  Vs tipus;
  tipus.PB("A+");
  tipus.PB("A-");
  tipus.PB("B+");
  tipus.PB("B-");
  tipus.PB("AB+");
  tipus.PB("AB-");
  tipus.PB("O+");
  tipus.PB("O-");

  int cas = 1;
  string a, b, c;
  while (cin >> a >> b >> c and a != "E") {
    int quin = 0;
    if (a == "?") quin = 1;
    else if (b == "?") quin = 2;
    else if (c == "?") quin = 3;

    cout << "Case " << cas++ << ": ";

    if (quin == 0) {
      cout << a << " " << b << " " << c << endl;
    }
    else if (quin == 3) {
      Vs res = fun(a, b);
      cout << a << " " << b << " ";
      if (res.size() > 1)  cout << "{";
      for (int i = 0; i < int(res.size()); ++i) {
        if (i) cout << ", ";
        cout << res[i];
      }
```

```
      if (res.size() > 1) cout << "}";
      cout << endl;
    }
    else {
      if (quin == 1) swap(a, b);
      Vs res;
      for (int i = 0; i < int(tipus.size()); ++i) {
        Vs tmp = fun(a, tipus[i]);
        bool ok = false;
        for (int j = 0; j < int(tmp.size()); ++j)
          if (tmp[j] == c) {
            ok = true;
            break;
          }
        if (ok) res.PB(tipus[i]);
      }
      if (quin == 1) swap(a, b);

      if (quin == 2) cout << a << " ";
      if (res.size() == 0) cout << "IMPOSSIBLE";
      else {
        if (res.size() > 1) cout << "{";
        for (int i = 0; i < int(res.size()); ++i) {
          if (i) cout << ", ";
          cout << res[i];
        }
        if (res.size() > 1) cout << "}";
      }
      cout << " ";
      if (quin == 1) cout << b << " ";
      cout << c << endl;
    }
  }
}
```

UVa 1062. Containers

3752 - Containers

World Finals - Tokyo - 2006/2007

A seaport container terminal stores large containers that are eventually loaded on seagoing ships for transport abroad. Containers coming to the terminal by road and rail are stacked at the terminal as they arrive.

Seagoing ships carry large numbers of containers. The time to load a ship depends in part on the locations of its containers. The loading time increases when the containers are not on the top of the stacks, but can be fetched only after removing other containers that are on top of them.

The container terminal needs a plan for stacking containers in order to decrease loading time. The plan must allow each ship to be loaded by accessing only topmost containers on the stacks, and minimizing the total number of stacks needed.

For this problem, we know the order in which ships must be loaded and the order in which containers arrive. Each ship is represented by a capital letter between A and Z (inclusive), and the ships will be loaded in alphabetical order. Each container is labeled with a capital letter representing the ship onto which it needs to be loaded. There is no limit on the number of containers that can be placed in a single stack.

Input

The input file contains multiple test cases. Each test case consists of a single line containing from 1 to 1000 capital letters representing the order of arrival of a set of containers. For example, the line ABAC means consecutive containers arrive to be loaded onto ships A, B, A, and C, respectively. When all containers have arrived, the ships are loaded in strictly increasing order: first ship A, then ship B, and so on.

A line containing the word `end` follows the last test case.

Output

For each input case, print the case number (beginning with 1) and the minimum number of stacks needed to store the containers before loading starts. Your output format should be similar to the one shown here.

Sample Input

```
A
CBACBACBACBACBA
CCCCBBBBAAAA
ACMICPC
end
```

Sample Output

```
Case 1: 1
Case 2: 3
Case 3: 1
Case 4: 4
```

Tokyo 2006-2007

```
/*
UVa 1062. Containers
This problem asks to place into stacks as they arrive in such a way that all
the stacks are sorted increasingly from top to bottom, and the goal is to
minimize the number of stacks used. The following greedy algorithm works:
process the letters in order of arrival, and place the letter in the stack
which has at the top the smallest symbol that is greater or equal to the symbol
currently being processed, or in a new stack if that is not possible.
*/
#include <iostream>
#include <string>
#include <vector>

using namespace std;

int main() {
  int ca = 1;
  for (string s; cin >> s && s != "end"; ++ca) {
    vector<char> top(26, 'Z'+1);
    int nstacks = 0;
    for (int i = 0; i < int(s.size()); ++i) {
      int st = -1;
      for (int j = 0; j < nstacks; ++j) {
        if (s[i] <= top[j]) {
          if (st == -1 || top[j] < top[st]) {
            st = j;
          }
        }
      }
      if (st == -1) {
        st = nstacks;
        ++nstacks;
      }
      top[st] = s[i];
    }
    cout << "Case " << ca << ": " << nstacks << endl;
  }
}
```

UVa 1064. Network

3808 - Network

World Finals - Tokyo - 2006/2007

A packet-switching network handles information in small units, breaking long messages into multiple packets before routing. Although each packet may travel along a different path, and the packets comprising a message may arrive at different times or out of order, the receiving computer reassembles the original message correctly.

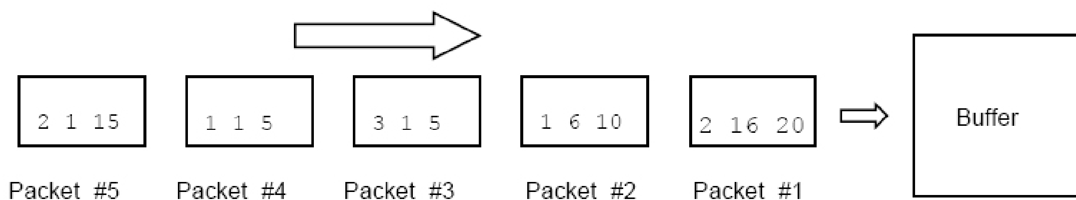
The receiving computer uses a buffer memory to hold packets that arrive out of order. You must write a program that calculates the minimum buffer size in bytes needed to reassemble the incoming messages when the number of messages (N), the number of packets (M), the part of the messages in each packet, the size of each message, and the order of the incoming packets are given.

When each packet arrives, it may be placed into the buffer or moved directly to the output area. All packets that are held in the buffer are available to be moved to the output area at any time. A packet is said to ``pass the buffer'' when it is moved to the output area. A message is said to ``pass the buffer'' when all of its packets have passed the buffer.

The packets of any message must be ordered so the data in the sequence of packets that pass the buffer is in order. For example, the packet containing bytes 3 through 5 of a message must pass the buffer before the packet containing bytes 6 through 10 of the same message. Messages can pass the buffer in any order, but all packets from a single message must pass the buffer consecutively and in order (but not necessarily at the same time). Note that unlike actual buffering systems, the process for this problem can look ahead at all incoming packets to make its decisions.

The packets consist of data and header. The header contains three numbers: the message number, the starting byte number of data in the packet, and the ending byte number of data in the packet respectively. The first byte number in any message is 1.

For example, the figure below shows three messages (with sizes of 10, 20, and 5 bytes) and five packets. The minimum buffer size for this example is 10 bytes. As they arrive, packet #1 and packet #2 are stored in the buffer. They occupy 10 bytes. Then packet #3 passes the buffer directly. Packet #4 passes the buffer directly and then packet #2 exits the buffer. Finally, packet #5 passes the buffer directly and packet #1 exits the buffer.



Input

The input file contains several test cases. The first line of each test case contains two integers N ($1 \le N \le 5$) and M ($1 \le M \le 1000$) . The second line contains N integers that are the sizes of messages in bytes; the first

number denotes the size of message #1, the second number denotes the size of message #2, and so on. Each of the following M lines describes a packet with three integers: the message number and the starting and ending byte numbers of data in the packet. No packet contains more 64 bytes of data.

The last test case is followed by a line containing two zeroes.

Output

For each test case, print a line containing the test case number (beginning with 1) followed by the minimum buffer size in bytes required to reassemble the original messages. Put a blank line after the output for each test case. Use the format of the sample output.

Sample Input

```
3 3
5 5 5
1 1 5
2 1 5
3 1 5
3 5
10 20 5
2 16 20
1 6 10
3 1 5
1 1 5
2 1 15
0 0
```

Sample Output

```
Case 1: 0

Case 2: 10
```

Tokyo 2006-2007

Tests-Setter: Derek Kisman

```
/*
UVa 1064. Network
For each of the N! orders of letting messages pass the buffer, simulate the
process and keep track of the number of bytes used simultaneously. Since the
process for this problem can look ahead at all incoming packets to make its
decisions, the buffer size required is the minimum number of bytes used
simultaneously among all the possible orders of letting messages pass the
buffer.
*/
#include <algorithm>
#include <iostream>
#include <map>
#include <set>
#include <vector>

#define FR first
#define SC second

using namespace std;

typedef pair<int, int> PII;

const int INF = 1000000000;

int main() {
  for (int ca = 1, n, m; cin >> n >> m && (n > 0 || m > 0); ++ca) {
    vector<int> sizes(n);
    for (int i = 0; i < n; ++i) {
      cin >> sizes[i];
    }
    vector<pair<int, PII> > packets(m);
    vector<vector<PII> > mpackets(n);
    for (int i = 0; i < m; ++i) {
      cin >> packets[i].FR >> packets[i].SC.FR >> packets[i].SC.SC;
      --packets[i].FR;
      mpackets[packets[i].FR].push_back(packets[i].SC);
    }
    vector<map<PII, int> > ind(n);
    for (int i = 0; i < n; ++i) {
      sort(mpackets[i].begin(), mpackets[i].end());
      for (int j = 0; j < int(mpackets[i].size()); ++j) {
        ind[i][mpackets[i][j]] = j;
      }
    }
    for (int i = 0; i < m; ++i) {
      int mind = ind[packets[i].FR][packets[i].SC];
      packets[i].SC.SC = packets[i].SC.SC-packets[i].SC.FR+1; // size
      packets[i].SC.FR = mind; // index in the message after sorting
    }
    vector<int> perm(n);
    for (int i = 0; i < n; ++i) {
      perm[i] = i;
    }
    int ans = INF;
    do {
      int cur = 0, cur_ans = 0, buffer_size = 0;
      vector<set<int> > buffer(n);
      vector<int> nxt_packet(n, 0);
      for (int i = 0; i < m; ++i) {
        buffer[packets[i].FR].insert(packets[i].SC.FR);
        buffer_size += packets[i].SC.SC;
```

```
        for (; cur < n && buffer[perm[cur]].count(nxt_packet[perm[cur]]) > 0;) {
          buffer[perm[cur]].erase(nxt_packet[perm[cur]]);
          buffer_size -= mpackets[perm[cur]][nxt_packet[perm[cur]]].SC-mpackets[perm
              [cur]][nxt_packet[perm[cur]]].FR+1;
          ++nxt_packet[perm[cur]];
          if (nxt_packet[perm[cur]] == int(mpackets[perm[cur]].size())) {
            ++cur;
          }
        }
        cur_ans = max(cur_ans, buffer_size);
      }
      ans = min(ans, cur_ans);
    } while(next_permutation(perm.begin(), perm.end()));
    cout << "Case " << ca << ": " << ans << endl << endl;
  }
}
```

1068 - Air Conditioning Machinery

You are a technician for the Air Conditioning Machinery company (ACM). Unfortunately, when you arrive at a customer site to install some air conditioning ducts, you discover that you are running low on supplies. You have only six duct segments, and they are all of the same kind, called an ``elbow."

You must install a duct in a confined space: a rectangular prism whose sides are multiples of a unit length. Think of the confined space as consisting of an array of unit cubes. Each elbow occupies exactly four unit cubes, as shown in Figure 1 below. A unit cube can be occupied by at most one elbow. Each elbow has exactly two openings, as indicated by the gray squares in the elbow shown in Figure 1. You may assemble the elbows into longer ducts, but your duct must be completely contained inside the given space. One way to connect two elbows is shown in Figure 2. Your task is to connect an inflow to an outflow. The inflow and the outflow are located on the exterior surface of the confined space, aligned with the unit cubes, as shown in Figure 3. To keep expenses down, you must accomplish this task while using the minimum number of elbows. out

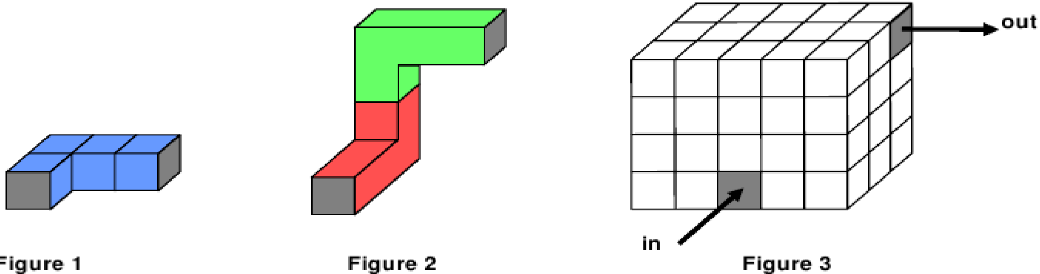


Figure 1 Figure 2 Figure 3

Input

The input consists of several test cases, each of which consists of a single line containing eleven input values separated by blanks. The input values for each test case are as follows.

The first three input values are integers (x_{max} , y_{max} , and z_{max}) that indicate the size of the confined space in the x , y , and z dimensions, respectively. Each unit cube in the confined space can be identified by coordinates (x, y, z) where $1 \le x \le x_{max}$, $1 \le y \le y_{max}$, and $1 \le z \le z_{max}$. x_{max} , y_{max} , and z_{max} are all positive and not greater than 20.

The next three input values are integers that indicate the location of the inflow by identifying the x , y , and z coordinates of the unit cube that connects to the inflow.

The next input value is a two-character string that indicates the direction of the inward flow, using one of the following codes: + x, - x, + y, - y, + z, - z . The inflow connection is on the face of the unit cube that receives this inward flow. For example, if the data specifies an inflow direction of + y , the inflow connection is on the face of the unit cube that faces in the negative y direction.

The next three input values are integers that indicate the location of the outflow by identifying the x , y , and z coordinates of the unit cube that connects to the outflow.

The last input value is a two-character string that indicates the direction of the outward flow, using the same codes described above. The outflow connection is on the face of the unit cube that generates this outward flow. For example, if the data specifies an outflow direction of + y , the outflow connection is on the face of the unit cube that faces in the positive y direction.

The last line of the input file consists of a single zero to indicate end of input.

Output

For each test case, print the case number (starting with 1) followed by the minimum number of elbows that are required to connect the inflow to the outflow without going outside the confined space. If the task cannot be accomplished with your supply of six elbow segments, print the word `Impossible' instead. Use the format in the sample data.

Sample Input

```
5 4 3 3 1 1 +z 5 4 3 +x
5 4 3 3 1 1 +z 1 2 3 -x
0
```

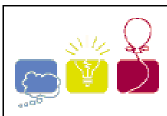
Sample Output

```
Case 1: 2
Case 2: Impossible
```

```
/*
UVa 1068. Air Conditioning Machinery
This problem can be solved with a method known as Iterative Deeping.
It consists of a backtracking with limited deep, that is initially
launched with limit=1, then with limit=2, and so on. When we finally
find a solution, we know the minimum distance to the closest solution
is exactly the current limit.
In this problem, we don't want solutions more than 6 steps away, so we
won't set a limit greater than 6. If even with limit=6 the backtracking
doesn't find a solution, it means that the solution doesn't exist, or it
is more than 6 steps away, and in this case we must print "Impossible".
*/
#include <iostream>
using namespace std;

struct P {
  int x, y, z;
  P() {}
  P(int xx, int yy, int zz) : x(xx), y(yy), z(zz) {}
  void operator+=(P p) { x += p.x; y += p.y; z += p.z; }
};

const int INF = 1000000000;
const P dir[] = { P(1, 0, 0), P(0, 1, 0), P(0, 0, 1), P(-1, 0, 0), P(0, -1, 0), P(0,
     0, -1) };

int mat[30][30][30];
int res, lim;
P dest;

istream& operator>>(istream& in, P& p) {
  in >> p.x >> p.y >> p.z;
  return in;
}

ostream& operator<<(ostream& out, const P& p) {
  out << "(" << p.x << ", " << p.y << ", " << p.z << ")";
  return out;
}

bool operator==(const P& a, const P& b) {
  return a.x == b.x and a.y == b.y and a.z == b.z;
}

P operator+(const P& a, const P& b) {
  return P(a.x + b.x, a.y + b.y, a.z + b.z);
}

P operator*(int t, const P& p) {
  return P(t*p.x, t*p.y, t*p.z);
}

bool vist(const P& p) {
  return mat[p.x][p.y][p.z];
}

void marca(const P& p) {
  mat[p.x][p.y][p.z] = 1;
}

void desmarca(const P& p) {
```

```
  mat[p.x][p.y][p.z] = 0;
}

bool back(P p, int d, int t) {
  if (p == dest) return true;
  if (t >= lim) return false;
  if (vist(p) or vist(p + dir[d])) return false;

  marca(p);
  marca(p + dir[d]);

  for (int k = 0; k < 6; ++k) {
    if ((k - d)%3 == 0) continue;
    if (vist(p + dir[d] + dir[k]) or vist(p + dir[d] + 2*dir[k])) continue;
    marca(p + dir[d] + dir[k]);
    marca(p + dir[d] + 2*dir[k]);
    if (back(p + dir[d] + 3*dir[k], k, t + 1)) return true;
    desmarca(p + dir[d] + dir[k]);
    desmarca(p + dir[d] + 2*dir[k]);
  }

  if (not vist(p + 2*dir[d])) {
    marca(p + 2*dir[d]);
    for (int k = 0; k < 6; ++k) {
      if ((k - d)%3 == 0) continue;
      if (vist(p + 2*dir[d] + dir[k])) continue;
      marca(p + 2*dir[d] + dir[k]);
      if (back(p + 2*dir[d] + 2*dir[k], k, t + 1)) return true;
      desmarca(p + 2*dir[d] + dir[k]);
    }
    desmarca(p + 2*dir[d]);
  }

  desmarca(p);
  desmarca(p + dir[d]);
  return false;
}

int main() {
  int cas = 1;
  int xmax, ymax, zmax;
  while (cin >> xmax >> ymax >> zmax and xmax > 0) {
    for (int y = 0; y <= ymax + 1; ++y)
      for (int z = 0; z <= zmax + 1; ++z)
        mat[0][y][z] = mat[xmax + 1][y][z] = 1;
    for (int x = 0; x <= xmax + 1; ++x)
      for (int z = 0; z <= zmax + 1; ++z)
        mat[x][0][z] = mat[x][ymax + 1][z] = 1;
    for (int x = 0; x <= xmax + 1; ++x)
      for (int y = 0; y <= ymax + 1; ++y)
        mat[x][y][0] = mat[x][y][zmax + 1] = 1;
    for (int x = 1; x <= xmax; ++x)
      for (int y = 1; y <= ymax; ++y)
        for (int z = 1; z <= zmax; ++z)
          mat[x][y][z] = 0;

    P ini;
    string s1, s2;
    cin >> ini >> s1 >> dest >> s2;
    int d1 = (s1[0] == '+' ? 0 : 3) + s1[1] - 'x';
    int d2 = (s2[0] == '+' ? 0 : 3) + s2[1] - 'x';
```

```
    dest += dir[d2];

    lim = 1;
    while (lim <= 6 and not back(ini, d1, 0)) ++lim;

    cout << "Case " << cas++ << ": ";
    if (lim <= 6) cout << lim << endl;
    else cout << "Impossible" << endl;
  }
}
```



4119 - Always an integer

World Finals - Banff Springs, Alberta - 2007/2008

Combinatorics is a branch of mathematics chiefly concerned with counting discrete objects. For instance, how many ways can you pick two people out of a crowd of n people? Into how many regions can you divide a circular disk by connecting n points on its boundary with one another? How many cubes are in a pyramid with square layers ranging from 1×1 to n×n cubes?

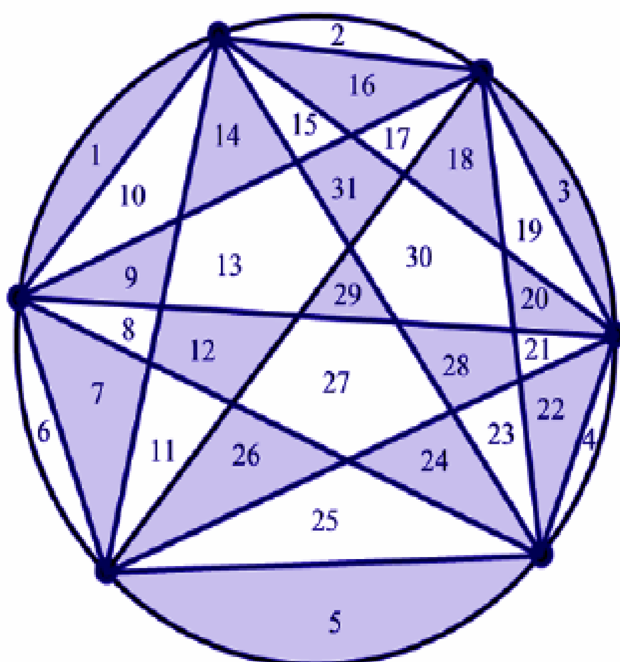


Figure 1: If we connect six points on the boundary of a circle, at most 31 regions are created.

Many questions like these have answers that can be reduced to simple polynomials in n . The answer to the first question above is $n(n - 1)/2$, or $(n \wedge 2 - n)/2$. The answer to the second is $(n \wedge 4 - 6n \wedge 3 + 23n \wedge 2 - 18n + 24)/24$. The answer to the third is $n(n + 1)(2n + 1)/6$, or $(2n \wedge 3 + 3n \wedge 2 + n)/6$. We write these polynomials in a standard form, as a polynomial with integer coefficients divided by a positive integer denominator.

These polynomials are answers to questions that can have integer answers only. But since they have fractional coefficients, they look as if they could produce non-integer results! Of course, evaluating these particular polynomials on a positive integer always results in an integer. For other polynomials of similar form, this is not necessarily true. It can be hard to tell the two cases apart. So that, naturally, is your task.

Input

The input consists of multiple test cases, each on a separate line. Each test case is an expression in the form $(P)/D$, where P is a polynomial with integer coefficients and D is a positive integer denominator. P is a sum of terms of the form $Cn \wedge E$, where the coefficient C and the exponent E satisfy the following conditions:

1. E is an integer satisfying $0 \le E \le 100$. If E is 0, then $Cn \wedge E$ is expressed as C . If E is 1, then $Cn \wedge E$ is expressed as Cn , unless C is 1 or -1. In those instances, $Cn \wedge E$ is expressed as n or - n .
2. C is an integer. If C is 1 or -1 and E is not 0 or 1, then the $Cn \wedge E$ will appear as $n \wedge E$ or - $n \wedge E$.
3. Only non-negative C values that are not part of the first term in the polynomial are preceded by +.
4. Exponents in consecutive terms are strictly decreasing.
5. C and D fit in a 32-bit signed integer.

See the sample input for details.

Input is terminated by a line containing a single period.

Output

For each test case, print the case number (starting with 1). Then print `Always an integer' if the test case polynomial evaluates to an integer for every positive integer $n$ . Print `Not always an integer' otherwise. Print the output for separate test cases on separate lines. Your output should follow the same format as the sample output.

Sample Input

```
(n^2-n)/2
(2n^3+3n^2+n)/6
(-n^14-11n+1)/3
.
```

Sample Output

```
Case 1: Always an integer
Case 2: Always an integer
Case 3: Not always an integer
```

Banff Springs, Alberta 2007-2008

Tests-Setter: Derek Kisman, Shahriar Manzoor

```
/*
UVa 1069. Always an integer
Use the following conjecture:
The P is divisible by the given number D when
evaluated at any of the integers: 0, 1, ..., deg(P)
IFF
It is always divisible by D when evaluated at any integer.

Be careful while reading the input, the format may be
tricky.
*/
#include <cstdlib>
#include <cstring>
#include <iostream>
using namespace std;

typedef long long ll;

const int DEG = 100;
const int NTEST = 2000;

ll TEST[NTEST];
ll C[DEG + 1];

int main() {
  for (int i = 0; i < 1000; ++i) TEST[i] = i + 1;
  for (int i = 0; i < 1000; ++i) TEST[1000 + i] = rand();

  int cas = 1;

  cin >> ws;
  while (cin.peek() != '.') {
    for (int i = 0; i <= DEG; ++i) C[i] = 0;

    char spam;
    cin >> spam;  // (
    while (cin.peek() != ')') {
      ll s = 1;
      cin >> ws;
      if (cin.peek() == '+') cin >> spam; // +
      else if (cin.peek() == '-') {
        s = -1;
        cin >> spam;  // -
      }

      ll c = 1;
      cin >> ws;
      if (cin.peek() != 'n') cin >> c;

      ll e = 0;
      cin >> ws;
      if (cin.peek() == 'n') {
        cin >> spam;  // n
        cin >> ws;
        if (cin.peek() == '^') {
          cin >> spam;  // ^
          cin >> e;
        }
        else e = 1;
      }
```

```
      C[e] += c*s;

      cin >> ws;
    }

    cin >> spam >> spam;  // )/

    ll m;
    cin >> m;

    int deg = 0;
    for (int i = 0; i <= DEG; ++i)
      if (C[i]) deg = i;

    bool ok = true;
    for (int i = 0; ok and i <= deg; ++i) {
      ll x = TEST[i];
      ll t = 0, p = 1;
      for (int j = 0; j <= DEG; ++j) {
        t = (t + p*C[j])%m;
        p = (p*x)%m;
      }
      if (t) ok = false;
    }

    cout << "Case " << cas++ << ": ";
    if (ok) cout << "Always an integer" << endl;
    else cout << "Not always an integer" << endl;

    cin >> ws;
  }
}
```

1073 - Glenbow Museum

The famous Glenbow Museum in Calgary is Western Canada's largest museum, with exhibits ranging from art to cultural history to mineralogy. A brand new section is being planned, devoted to brilliant computer programmers just like you. Unfortunately, due to lack of space, the museum is going to have to build a brand new building and relocate into it.

The size and capacity of the new building differ from those of the original building. But the floor plans of both buildings are orthogonal polygons. An orthogonal polygon is a polygon whose internal angles are either 90Â° or 270Â°. If 90Â° angles are denoted as R (Right) and 270Â° angles are denoted as O (Obtuse) then a string containing only R and O can roughly describe an orthogonal polygon. For example, a rectangle (Figure 1) is the simplest orthogonal polygon and it can be described as RRRR (the angles are listed in counter-clockwise order, starting from any corner). Similarly, a cross-shaped orthogonal polygon (Figure 2) can be described by the sequence RRORRORRORRO, RORRORRORROR, or ORRORRORRORR. These sequences are called angle strings.

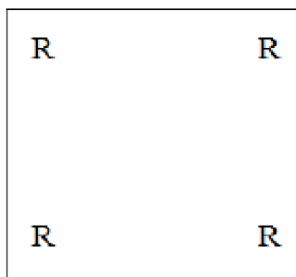


Figure 1: A rectangle

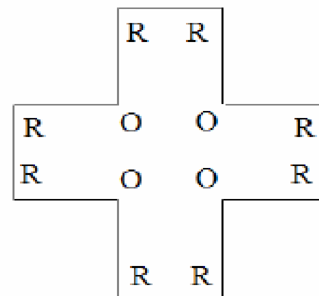


Figure 2: A cross-shaped polygon

Of course, an angle string does not completely specify the shape of a polygon -- it says nothing about the length of the sides. And some angle strings cannot possibly describe a valid orthogonal polygon (RRROR, for example).

To complicate things further, not all orthogonal polygons are acceptable floor plans for the museum. A museum contains many valuable objects, and these objects must be guarded. Due to cost considerations, no floor can have more than one guard. So a floor plan is acceptable only if there is a place within the floor from which one guard can see the entire floor. Similarly, an angle string is acceptable only if it describes at least one acceptable polygon. Note that the cross-shaped polygon in Figure 2 can be guarded by someone standing in the center, so it is acceptable. Thus the angle string RRORRORRORRO is acceptable, even though it also describes other polygons that cannot be properly guarded by a single guard.

Help the designers of the new building determine how many acceptable angle strings there are of a given length.

Input

The input file contains several test cases. Each test case consists of a line containing a positive integer L ($1 \leq L \leq 1000$) , which is the desired length of an angle string. The input will end with a line containing a single zero.

Output

For each test case, print a line containing the test case number (beginning with 1) followed by the number of acceptable angle strings of the given length. Follow the format of the sample output.

Sample Input

```
4
6
0
```

Sample Output

```
Case 1: 1
Case 2: 6
```

```cpp
/*
UVa 1073. Glenbow Museum
Let's look at orthogonal polygons as paths starting at some vertex with some
direction, and repeatedly advance some distance and turn right or left until
the starting point is reached again. Notice that if the number of turns is odd
it cannot possibly be a closed orthogonal polygon.Also, the number of turns has
to be at least 4, because at least 4 turns are required to close an orthogonal
polygon.

If the number of turns L is even and at least 4, we know that 4 of its turns
have to be R so that the orthogonal polygon is closed. Among the remaining
L-4 turns, the difference between the number of Rs and the number of Os has to
be a multiple of 4, so that the final direction is the same as the initial
direction. However, a difference other than 4 would require the polygon to
self-intersect, but that would not be valid.

There cannot be 2 consecutive O turns, because that would make it impossible
to see the whole polygon from one point. There are no other restrictions
concerning the order of turns.

Assuming L is even and at least 4, we want to count the number of angle strings
that have (L+4)/2 Rs and (L-4)/2 Os such that there are no two consecutive Os:
C((L+4)/2 + 1, (L-4)/2)
where C(n, k) is the binomial coefficient.
But the first symbol is consecutive to the last symbol, therefore when there are
at least 2 Os we have to substract the number of previously considered valid
strings that start and end with O:
C((L+4)/2 - 1, (L-4)/2 - 2)

Binomial coefficients can be computed with Dynamic Programming using the
following identity:
C(n, k) = C(n-1, k) + C(n-1, k-1)
*/
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;

int main() {
  vector<vector<ll> > comb(1002, vector<ll>(1002, 0));
  comb[0][0] = 1;
  for (int i = 1; i < int(comb.size()); ++i) {
    comb[i][0] = comb[i][i] = 1;
    for (int j = 1; j < i; ++j) {
      comb[i][j] = comb[i-1][j-1]+comb[i-1][j];
    }
  }

  for (int ca = 1, l; cin >> l && l > 0; ++ca) {
    cout << "Case " << ca << ": ";
    if (l%2 == 1 || l == 2) cout << 0 << endl;
    else {
      int o = (l-4)/2;
      int r = l-o;
      cout << comb[r+1][o] - (o >= 2? comb[r-1][o-2] : 0) << endl;
    }
  }
}
```

UVa 1076. Password Suspects

4126 - Password Suspects

World Finals - Banff Springs, Alberta - 2007/2008

You are the computer whiz for the secret organization known as the Sneaky Underground Smug Perpetrators of Evil Crimes and Thefts. The target for SUSPECT's latest evil crime is their greatest foe, the Indescribably Clever Policemen's Club, and everything is prepared. Everything, except for one small thing: the secret password for ICPC's main computer system.

The password is known to consist only of lowercase letters `a'-`z'. Furthermore, through various sneaky observations, you have been able to determine the length of the password, as well as a few (possibly overlapping) substrings of the password, though you do not know exactly where in the password they occur.

For instance, say that you know that the password is 10 characters long, and that you have observed the substrings ``hello" and ``world". Then the password must be either ``helloworld" or ``worldhello".

The question is whether this information is enough to reduce the number of possible passwords to a reasonable amount. To answer this, your task is to write a program that determines the number of possible passwords and, if there are at most 42 of them, prints them all.

Input

The input file contains several test cases. Each test case begins with a line containing two integers N and M ($1 \le N \le 25$, $0 \le M \le 10$) , giving the length of the password and the number of known substrings respectively. This is followed by M lines, each containing a known substring. Each known substring consists of between 1 and 10 lowercase letters `a'-`z'.

The last test case is followed by a line containing two zeroes.

Output

For each test case, print the case number (beginning with 1) followed by `Y suspects', where Y is the number of possible passwords for this case. If the number of passwords is at most 42, then output all possible passwords in alphabetical order, one per line.

The input will be such that the number of possible passwords at most 10^{15} .

Sample Input

```
10 2
hello
world
10 0
4 1
icpc
0 0
```

Sample Output

```
Case 1: 2 suspects
helloworld
worldhello
Case 2: 141167095653376 suspects
Case 3: 1 suspects
icpc
```

Banff Springs, Alberta 2007-2008

Tests-Setter: Derek Kisman

```
/*
UVa 1076. Password Suspects
Construct an automata that recognizes whether all the words in the input
have appeared as substrings in a given string.
Thanks to that automata, the following Dynamic Programming algorithm can
be used:
fun(n,m) is the number of different strings of length 'n' that will be
recognized by the automata, assuming the automata is at the state 'm' at
the beginning. This value is calculated recursively.
The part of printing the at most 42 strings when necessary should not be
hard to implement.
*/
#include <iostream>
#include <map>
#include <string>
#include <vector>
using namespace std;

typedef long long ll;
typedef map<ll, ll> MAP;
typedef vector<string> Vs;

int N, M;
ll F;
Vs V;
MAP dp[30];
int fail[20][20][26];

inline int get(ll mask, int i) {
  return (mask>>(4*i))&15;
}

inline ll set(ll mask, int i, ll v) {
  mask &= ~(15LL<<(4*i));
  mask |= v<<(4*i);
  return mask;
}

ll next(ll m, char c) {
  for (int i = 0; i < M; ++i) {
    int t = get(m, i);
    m = set(m, i, fail[i][t][c - 'a']);
  }
  return m;
}

ll fun(int n, ll m) {
  if (n == 0) {
    if (m == F) return 1;
    return 0;
  }
  if (dp[n].count(m)) return dp[n][m];
  ll res = 0;
  for (char c = 'a'; c <= 'z'; ++c)
    res += fun(n - 1, next(m, c));
  return dp[n][m] = res;
}

void print(int n, ll m, string s) {
  if (fun(n, m) == 0) return;
  if (n == 0) {
```

```
    cout << s << endl;
    return;
  }
  for (char c = 'a'; c <= 'z'; ++c)
    print(n - 1, next(m, c), s + c);
}

// a.size() <= b.size()
int common(string a, string b) {
  int na = a.size();
  for (int i = 0; i < na; ++i)
    if (a.substr(i, na - i) == b.substr(0, na - i))
      return na - i;
  return 0;
}

int main() {
  int cas = 1;
  while (cin >> N >> M and N > 0) {
    for (int i = 0; i < 30; ++i) dp[i].clear();

    V = Vs(M);
    for (int i = 0; i < M; ++i) cin >> V[i];

    F = 0;
    for (int i = 0; i < M; ++i) F = set(F, i, V[i].size());

    for (int i = 0; i < M; ++i) {
      for (int j = 0; j <= V[i].size(); ++j) {
        for (int k = 0; k < 26; ++k) {
          if (j == V[i].size()) fail[i][j][k] = V[i].size();
          else {
            string a = V[i].substr(0, j) + char('a' + k);
            fail[i][j][k] = common(a, V[i]);
          }
        }
      }
    }

    ll f = fun(N, 0);

    cout << "Case " << cas++ << ": " << f << " suspects" << endl;
    if (f <= 42) print(N, 0, "");
  }
}
```

UVa 1078. Steam Roller

4128 - Steam Roller

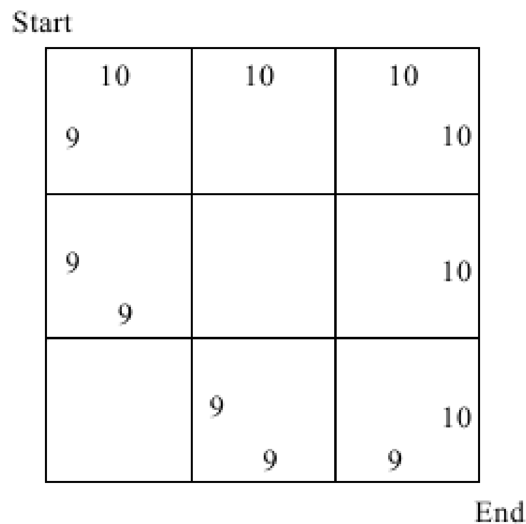
World Finals - Banff Springs, Alberta - 2007/2008

Johnny drives a steam roller, which like all steam rollers is slow and takes a relatively long time to start moving, change direction, and brake to a full stop. Johnny has just finished his day's work and is driving his steam roller home to see his wife. Your task is to find the fastest path for him and his steam roller.

The city where Johnny lives has a regular structure (the streets form an orthogonal system). The city streets are laid out on a rectangular grid of intersections. Each intersection is connected to its neighbors (up to four of them) by a street. Each street is exactly one block long. When Johnny enters a street, he must always travel to the other end (continue to the next intersection). From that point, he can continue in any of the four possible directions to another intersection, and so on.

By studying the road conditions of the streets, Johnny has calculated the time needed to go from one end to the other of every street in town. The time is the same for both directions. However, Johnny's calculations hold only under the ideal condition that the steam roller is already in motion when it enters a street and does not need to accelerate or brake. Whenever the steam roller changes direction at a intersection directly before or after a street, the estimated ideal time for that street must be doubled. The same holds if the roller begins moving from a full stop (for example at the beginning of Johnny's trip) or comes to a full stop (for example at the end of his trip).

The following picture shows an example. The numbers show the ``ideal" times needed to drive through the corresponding streets. Streets with missing numbers are unusable for steam rollers. Johnny wants to go from the top-left corner to the bottom-right one.



The path consisting of streets labeled with 9's seems to be faster at the first sight. However, due to the braking and accelerating restrictions, it takes double the estimated time for every street on the path, making the total time 108. The path along the streets labeled with 10's is faster because Johnny can drive two of the streets at

4128 - Steam Roller 1/2

the full speed, giving a total time of 100.

Input

The input consists of several test cases. Each test case starts with six positive integer numbers: R , C , r_1 , c_1 , r_2 , and c_2 . R and C describe the size of the city, r_1 , c_1 are the starting coordinates, and r_2 , c_2 are the coordinates of Johnny's home. The starting coordinates are different from the coordinates of Johnny's home. The numbers satisfy the following condition: $1 \le r_1, r_2 \le R \le 100$, $1 \le c_1, c_2 \le C \le 100$.

After the six numbers, there are C - 1 non-negative integers describing the time needed to drive on streets between intersections (1,1) and (1,2), (1,2) and (1,3), (1,3) and (1,4), and so on. Then there are C non-negative integers describing the time need to drive on streets between intersections (1,1) and (2,1), (1,2) and (2,2), and so on. After that, another C - 1 non-negative integers describe the next row of streets across the width of the city. The input continues in this way to describe all streets in the city. Each integer specifies the time needed to drive through the corresponding street (not higher than 10000), provided the steam roller proceeds straight through without starting, stopping, or turning at either end of the street. If any combination of one or more of these events occurs, the time is multiplied by two. Any of these integers may be zero, in which case the corresponding street cannot be used at all.

The last test case is followed by six zeroes.

All numbers are separated with at least one whitespace character (space, tab, or newline), but any amount of additional whitespace (including empty lines) may be present to improve readability.

Output

For each test case, print the case number (beginning with 1) followed by the minimal time needed to go from intersection r_1, c_1 to r_2, c_2 . If the trip cannot be accomplished (due to unusable streets), print the word \``Impossible`' instead.

Sample Input

```
4  4  1  1  4  4
 10   10   10
9   0   0   10
  0   0   0
9   0   0   10
  9   0   0
0   9   0   10
  0   9   9

2  2  1  1  2  2  0  1  1  0

0  0  0  0  0  0
```

Sample Output

```
Case 1: 100
Case 2: Impossible
```

Banff Springs, Alberta 2007-2008

Tests-Setter: Derek Kisman

```
/*
UVa 1078. Steam Roller
We can solve this problem using Dijkstra's algorithm for shortest path in a
graph where the nodes are (row, column, direction, speed during the last move).
There are 4 possible directions and 2 possible speed values. There is an edge
between two nodes if their positions (row, column) are neighbours and, in case
of change of direction, the conditions about speed described in the problem
statement are satisfied.
*/
#include <iostream>
#include <queue>
#include <vector>

#define Y first
#define X second
#define FR first
#define SC second

using namespace std;

typedef long long ll;
typedef pair<int, int> PII;
typedef vector<ll> VI;
typedef vector<VI> V2I;
typedef vector<V2I> V3I;
typedef vector<V3I> V4I;

const ll INF = 1000000000000000LL;
const int DY[] = {-1, 0, 1, 0};
const int DX[] = {0, 1, 0, -1};

struct State {
  int row, col, dir, prev_slow;

  bool operator<(const State& b) const {
    return row < b.row;
  }

  State() {};
  State(int a, int b, int c, int d) : row(a), col(b), dir(c), prev_slow(d) {};
};

inline bool inside(int y, int x, int r, int c) {
  return y >= 0 && y < r && x >= 0 && x < c;
}

int main() {
  for (int ca = 1, r, c, r1, c1, r2, c2; cin >> r >> c >> r1 >> c1 >> r2 >> c2 && (r
       > 0 || c > 0 || r1 > 0 || c1 > 0 || r2 > 0 || c2 > 0); ++ca) {
    --r1;
    --c1;
    --r2;
    --c2;
    PII start(r1, c1), dest(r2, c2);
    vector<vector<ll> > cost(2*r-1, vector<ll>(2*c-1, 0));
    for (int i = 0; i < int(cost.size()); i += 2) {
      for (int j = 1; j < int(cost[i].size()); j += 2) {
        cin >> cost[i][j];
      }
      if (i+1 < int(cost.size())) {
        for (int j = 0; j < int(cost[i+1].size()); j += 2) {
```

```
        cin >> cost[i+1][j];
      }
    }
  }
  for (int i = 0; i < int(cost.size()); ++i) {
    for (int j = 0; j < int(cost[i].size()); ++j) {
      if (cost[i][j] == 0) {
        cost[i][j] = INF;
      }
    }
  }
  V4I dist(r, V3I(c, V2I(4, VI(2, INF))));
  priority_queue<pair<ll, State>, vector<pair<ll, State> >, greater<pair<ll, State
      > > > pq;
  for (int dir = 0; dir < 4; ++dir) {
    State cur(start.Y+DY[dir], start.X+DX[dir], dir, 1);
    if (inside(cur.row, cur.col, r, c)) {
      pq.push(pair<ll, State>(2*cost[2*cur.row-DY[cur.dir]][2*cur.col-DX[cur.dir
          ]], cur));
    }
  }
  for (; !pq.empty();) {
    ll dis = pq.top().FR;
    State cur = pq.top().SC;
    pq.pop();

    if (dis >= dist[cur.row][cur.col][cur.dir][cur.prev_slow]) {
      continue;
    }
    dist[cur.row][cur.col][cur.dir][cur.prev_slow] = dis;
    State nxt;

    // keep direction
    nxt = State(cur.row+DY[cur.dir], cur.col+DX[cur.dir], cur.dir, 0);
    if (inside(nxt.row, nxt.col, r, c)) {
      ll nxt_dis = dis+cost[2*cur.row+DY[nxt.dir]][2*cur.col+DX[nxt.dir]];
      if (nxt_dis < dist[nxt.row][nxt.col][nxt.dir][nxt.prev_slow]) {
        pq.push(pair<ll, State>(nxt_dis, nxt));
      }
    }

    // keep direction but slow down
    nxt = State(cur.row+DY[cur.dir], cur.col+DX[cur.dir], cur.dir, 1);
    if (inside(nxt.row, nxt.col, r, c)) {
      ll nxt_dis = dis+2*cost[2*cur.row+DY[nxt.dir]][2*cur.col+DX[nxt.dir]];
      if (nxt_dis < dist[nxt.row][nxt.col][nxt.dir][nxt.prev_slow]) {
        pq.push(pair<ll, State>(nxt_dis, nxt));
      }
    }

    // turn right
    nxt = State(cur.row+DY[(cur.dir+1)%4], cur.col+DX[(cur.dir+1)%4], (cur.dir+1)
        %4, 1);
    if (inside(nxt.row, nxt.col, r, c) && cur.prev_slow == 1) {
      ll nxt_dis = dis+2*cost[2*cur.row+DY[nxt.dir]][2*cur.col+DX[nxt.dir]];
      if (nxt_dis < dist[nxt.row][nxt.col][nxt.dir][nxt.prev_slow]) {
        pq.push(pair<ll, State>(nxt_dis, nxt));
      }
    }

    // turn around
```

```
      nxt = State(cur.row+DY[(cur.dir+2)%4], cur.col+DX[(cur.dir+2)%4], (cur.dir+2)
          %4, 1);
      if (inside(nxt.row, nxt.col, r, c) && cur.prev_slow == 1) {
        ll nxt_dis = dis+2*cost[2*cur.row+DY[nxt.dir]][2*cur.col+DX[nxt.dir]];
        if (nxt_dis < dist[nxt.row][nxt.col][nxt.dir][nxt.prev_slow]) {
          pq.push(pair<ll, State>(nxt_dis, nxt));
        }
      }

      // turn left
      nxt = State(cur.row+DY[(cur.dir+3)%4], cur.col+DX[(cur.dir+3)%4], (cur.dir+3)
          %4, 1);

      if (inside(nxt.row, nxt.col, r, c) && cur.prev_slow == 1) {
        ll nxt_dis = dis+2*cost[2*cur.row+DY[nxt.dir]][2*cur.col+DX[nxt.dir]];
        if (nxt_dis < dist[nxt.row][nxt.col][nxt.dir][nxt.prev_slow]) {
          pq.push(pair<ll, State>(nxt_dis, nxt));
        }
      }
    }
    ll ans = INF;
    for (int dir = 0; dir < 4; ++dir) {
      ll cur_ans = dist[dest.Y][dest.X][dir][1];
      ans = min(ans, cur_ans);
    }
    cout << "Case " << ca << ": ";
    if (start == dest) cout << 0 << endl;
    else if (ans == INF) cout << "Impossible" << endl;
    else cout << ans << endl;
  }
}
```

UVa 1079. A Careful Approach

4445 - A Careful Approach

World Finals - Stockholm - 2008/2009

If you think participating in a programming contest is stressful, imagine being an air traffic controller. With human lives at stake, an air traffic controller has to focus on tasks while working under constantly changing conditions as well as dealing with unforeseen events.

Consider the task of scheduling the airplanes that are landing at an airport. Incoming airplanes report their positions, directions, and speeds, and then the controller has to devise a landing schedule that brings all airplanes safely to the ground. Generally, the more time there is between successive landings, the ``safer" a landing schedule is. This extra time gives pilots the opportunity to react to changing weather and other surprises.

Luckily, part of this scheduling task can be automated - this is where you come in. You will be given scenarios of airplane landings. Each airplane has a time window during which it can safely land. You must compute an order for landing all airplanes that respects these time windows. Furthermore, the airplane landings should be stretched out as much as possible so that the minimum time gap between successive landings is as large as possible. For example, if three airplanes land at 10:00am, 10:05am, and 10:15am, then the smallest gap is five minutes, which occurs between the first two airplanes. Not all gaps have to be the same, but the smallest gap should be as large as possible.

Input

The input file contains several test cases consisting of descriptions of landing scenarios. Each test case starts with a line containing a single integer n ($2 \le n \le 8$), which is the number of airplanes in the scenario. This is

followed by n lines, each containing two integers a_i, b_i, which give the beginning and end of the closed interval $[a_i, b_i]$ during which the i-th plane can land safely. The numbers a_i and b_i are specified in minutes and satisfy $0 \le a_i \le b_i \le 1440$.

The input is terminated with a line containing the single integer zero.

Output

For each test case in the input, print its case number (starting with 1) followed by the minimum achievable time gap between successive landings. Print the time split into minutes and seconds, rounded to the closest second. Follow the format of the sample output.

Sample Input

```
3
0 10
5 15
10 15
2
0 10
10 20
0
```

Sample Output

```
Case 1: 7:30
Case 2: 20:00
```

Stockholm 2008-2009

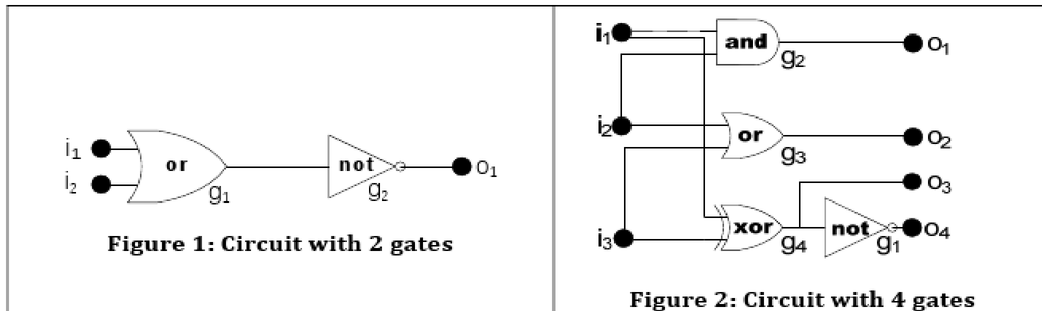
```
/*
UVa 1079. A Careful Approach
Given the order in which the planes have to land, and a fixed size of gap,
it can be easily checked whether a possible arrangement exists that leaves
at least that gap between planes. This can be checked with a Greedy algorithm.
So, in order to find the maximum gap we can do a binary search over its size,
using the Greedy algorithm to check whether a guess is too small or too large.
Given that the number of planes is really small, perform the aforesaid process
for each possible order of the planes (N! permutations).
*/
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

#define X first
#define Y second

typedef pair<int, int> P;
typedef vector<P> Vp;

bool test(const Vp& v, double g) {
  int n = v.size();
  double t = v[0].X;
  for (int i = 1; i < n; ++i) {
    if (t + g > v[i].Y) return false;
    t = max(double(v[i].X), t + g);
  }
  return true;
}

int main() {
  int cas = 1;

  int n;
  while (cin >> n and n > 0) {
    Vp v(n);
    for (int i = 0; i < n; ++i) cin >> v[i].X >> v[i].Y;

    sort(v.begin(), v.end());

    double res = 0;

    do {
      double e = 0, d = 1440;
      for (int step = 0; step <= 50; ++step) {
        double m = 0.5*e + 0.5*d;
        if (test(v, m)) e = m;
        else d = m;
      }

      res = max(res, d);
    }
    while (next_permutation(v.begin(), v.end()));

    int sec = int(60*res + 0.5);

    cout << "Case " << cas++ << ": ";
    cout << sec/60 << ":";
    if (sec%60 < 10) cout << 0;
    cout << sec%60 << endl;
```

```
    }
}
```

UVa 1080. My Bad

1080 - My Bad

A logic circuit maps its input through various gates to its output with no feedback loops in the circuit. The input and output are an ordered set of logical values, represented here by ones and zeros. The circuits we consider are comprised of and gates (which output 1 only when their two inputs are both 1), or gates (which output 1 when one or both of their inputs are 1), exclusive or (xor) gates (which output 1 only when exactly one of the two inputs is 1), and not gates (which output the complement of their single input). The figures below show two circuits.



Unfortunately, real gates sometimes fail. Although the failures may occur in many different ways, this problem limits attention to gates that fail in one of three ways: 1) always inverting the correct output, 2) always yielding 0, and 3) always yielding 1. In the circuits for this problem, at most one gate will fail.

You must write a program that analyzes a circuit and a number of observations of its input and output to see if the circuit is performing correctly or incorrectly. If at least one set of inputs produces the wrong output, your program must also attempt to determine the unique failing gate and the way in which this gate is failing. This may not always be possible.

Input

The input consists of multiple test cases, each representing a circuit with input and output descriptions. Each test case has the following parts in order.

- A line containing three positive integers giving the number of inputs ($N \leq 8$), the number of gates ($G \leq 19$), and the number of outputs ($U \leq 19$) in the circuit.
- One line of input for each gate. The first line describes gate g_1. If there are several gates, the next line describes gate g_2, and so on. Each of these lines contains the gate type (`a` = and, `n` = not, `o` = or, and `x` = exclusive or), and identification of the input(s) to the gate. Gate input comes from the circuit inputs (`i1`, `i2`, ...) or the output of another gate (`g1`, `g2`, ...).
- A line containing the numbers of the gates connected to the U outputs u_1, u_2,.... For example, if there are three outputs, and u_1 comes from g_5, u_2 from g_1, and u_3 from g_4, then the line would contain: `5 1 4`
- A line containing an integer which is the number of observations of the circuit's behavior (B).
- Finally B lines, each containing N values (ones and zeros) giving the observed input values and U values giving the corresponding observed output values. No two observations have the same input values.

Consecutive entries on any line of the input are separated by a single space. The input is terminated with a line containing three zeros.

Output

For each circuit in the input, print its case number (starting with 1), followed by a colon and a blank, and then the circuit analysis, which will be one of the following (with `#` replaced by the appropriate gate number):

```
No faults detected
Gate # is failing; output inverted
Gate # is failing; output stuck at 0
Gate # is failing; output stuck at 1
Unable to totally classify the failure
```

The circuits pictured in Figure 1 and Figure 2 are used in the first and last sample test cases.

Sample Input

```
2 2 1
o i1 i2
n g1
2
2
1 0 0
0 0 1
2 1 1
```

```
a i1 i2
1
1
1 0 1
2 1 1
a i1 i2
1
2
1 0 1
1 1 1
1 1 1
n i1
1
2
1 1
0 0
3 4 4
n g4
a i1 i2
o i2 i3
x i3 i1
2 3 4 1
4
0 1 0 0 1 0 1
0 1 1 0 1 1 0
1 1 1 0 1 0 1
0 0 0 0 0 0 1
0 0 0
```

Sample Output

```
Case 1: No faults detected
Case 2: Unable to totally classify the failure
Case 3: Gate 1 is failing; output stuck at 1
Case 4: Gate 1 is failing; output inverted
Case 5: Gate 2 is failing; output stuck at 0
```

```
/*
UVa 1080. My Bad
First, you should implement a simulator for the circuit, so you can
calculate the output the circuit should give for each given input.
Simulate the circuit witch each of the given inputs and compare
the output given to the output the circuit should give, if they
coincide, then "No faults detected". Otherwise, force each
gate to have each possible failure, and check if the output
coincides then. If there's only one combination of {gate,
kind of failure} that makes the circuit behave correspondingly
to the given input/output, then print the failure in the proper
format, otherwise, if there are many possible combinations,
print "Unable to totally...".
*/
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

#define PB push_back

typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef queue<int> Q;

int input[40], inputs;
int gates;
int output[20], outputs;

int N; // gates + inputs
Mi net;
int topo[20];

char gate_type[20];
int gate_input[20][2];
int output_from[20];

int tests;
int test_input[300][20];
int test_output[300][20];

int fail_gate, fail_type;

void simula() {
  for (int i = 0; i < gates; ++i) {
    int n = topo[i];
    int g = n - inputs;
    if (gate_type[g] == 'a') {
      int a = input[gate_input[g][0]];
      int b = input[gate_input[g][1]];
      input[n] = min(a, b);
    }
    else if (gate_type[g] == 'o') {
      int a = input[gate_input[g][0]];
      int b = input[gate_input[g][1]];
      input[n] = max(a, b);
    }
    else if (gate_type[g] == 'x') {
      int a = input[gate_input[g][0]];
      int b = input[gate_input[g][1]];
      input[n] = max(a, b) - min(a, b);
```

```
    }
    else {
      int a = input[gate_input[g][0]];
      input[n] = 1 - a;
    }

    if (fail_gate == g) {
      if (fail_type == 0) input[n] = 1 - input[n];
      else if (fail_type == 1) input[n] = 0;
      else input[n] = 1;
    }
  }

  for (int i = 0; i < outputs; ++i)
    output[i] = input[inputs + output_from[i]];
}

void toposort() {
  Vi deg(N, 0);
  for (int i = 0; i < N; ++i)
    for (int j = 0; j < net[i].size(); ++j)
      ++deg[net[i][j]];

  Q q;
  for (int i = 0; i < N; ++i)
    if (deg[i] == 0)
      q.push(i);

  int p = 0;
  while (not q.empty()) {
    int n = q.front();
    q.pop();

    if (inputs <= n) topo[p++] = n;

    for (int i = 0; i < net[n].size(); ++i)
      if (--deg[net[n][i]] == 0)
        q.push(net[n][i]);
  }
}

int main() {
  int cas = 1;

  while (cin >> inputs >> gates >> outputs and inputs > 0) {
    N = inputs + gates;
    net = Mi(N);

    for (int i = 0; i < gates; ++i) {
      cin >> gate_type[i];

      char c;
      int t;
      cin >> c >> t;
      --t;
      gate_input[i][0] = (c == 'g' ? inputs : 0) + t;
      net[gate_input[i][0]].PB(inputs + i);

      if (gate_type[i] != 'n') {
        cin >> c >> t;
        --t;
```

```
      gate_input[i][1] = (c == 'g' ? inputs : 0) + t;
      net[gate_input[i][1]].PB(inputs + i);
    }
  }

  for (int i = 0; i < outputs; ++i) {
    cin >> output_from[i];
    --output_from[i];
  }

  toposort();

  cin >> tests;
  for (int i = 0; i < tests; ++i) {
    for (int j = 0; j < inputs; ++j) cin >> test_input[i][j];
    for (int j = 0; j < outputs; ++j) cin >> test_output[i][j];
  }

  bool ok = true;
  fail_gate = -1;
  for (int i = 0; i < tests; ++i) {
    for (int j = 0; j < inputs; ++j) input[j] = test_input[i][j];
    simula();
    for (int j = 0; j < inputs; ++j)
      if (output[j] != test_output[i][j])
        ok = false;
  }

  cout << "Case " << cas++ << ": ";

  if (ok) {
    cout << "No faults detected" << endl;
    continue;
  }

  int fail_g = -1, fail_t = -1;
  for (fail_gate = 0; fail_g != -2 and fail_gate < gates; ++fail_gate)
    for (fail_type = 0; fail_g != -2 and fail_type < 3; ++fail_type) {
      ok = true;
      for (int test = 0; ok and test < tests; ++test) {
        for (int i = 0; i < inputs; ++i) input[i] = test_input[test][i];
        simula();
        for (int i = 0; ok and i < outputs; ++i)
          if (output[i] != test_output[test][i])
            ok = false;
      }
      if (ok) {
        if (fail_g == -1) {
          fail_g = fail_gate;
          fail_t = fail_type;
        }
        else fail_g = -2;
      }
    }

  if (fail_g < 0) cout << "Unable to totally classify the failure" << endl;
  else {
    cout << "Gate " << fail_g + 1 << " is failing; ";
    if (fail_t == 0) cout << "output inverted" << endl;
    else if (fail_t == 1) cout << "output stuck at 0" << endl;
    else cout << "output stuck at 1" << endl;
```

```
        }
    }
}
```

1083 - Fare and Balanced

Handling traffic congestion is a difficult challenge for young urban planners. Millions of drivers, each with different goals and each making independent choices, combine to form a complex system with sometimes predictable, sometimes chaotic behavior. As a devoted civil servant, you have been tasked with optimizing rushhour traffic over collections of roads.

All the roads lie between a residential area and a downtown business district. In the morning, each person living in the residential area drives a route to the business district. The morning commuter traffic on any particular road travels in only one direction, and no route has cycles (morning drivers do not backtrack).

Each road takes a certain time to drive, so some routes are faster than others. Drivers are much more likely to choose the faster routes, leading to congestion on those roads. In order to balance the traffic as much as possible, you are to add tolls to some roads so that the perceived ``cost'' of every route ends up the same. However, to avoid annoying drivers too much, you must not levy a toll on any driver twice, no matter which route he or she takes.

Figure 5 shows a collection of five roads that form routes from the residential area (at intersection 1) to the downtown business district (at intersection 4). The driving cost of each road is written in large blue font. The dotted arrows show the three possible routes from 1 to 4. Initially the costs of the routes are 10, 8 and 12. After adding a toll of cost 2 to the road connecting 1 and 4 and a toll of cost 4 to the road connecting 3 and 4, the cost of each route becomes 12.

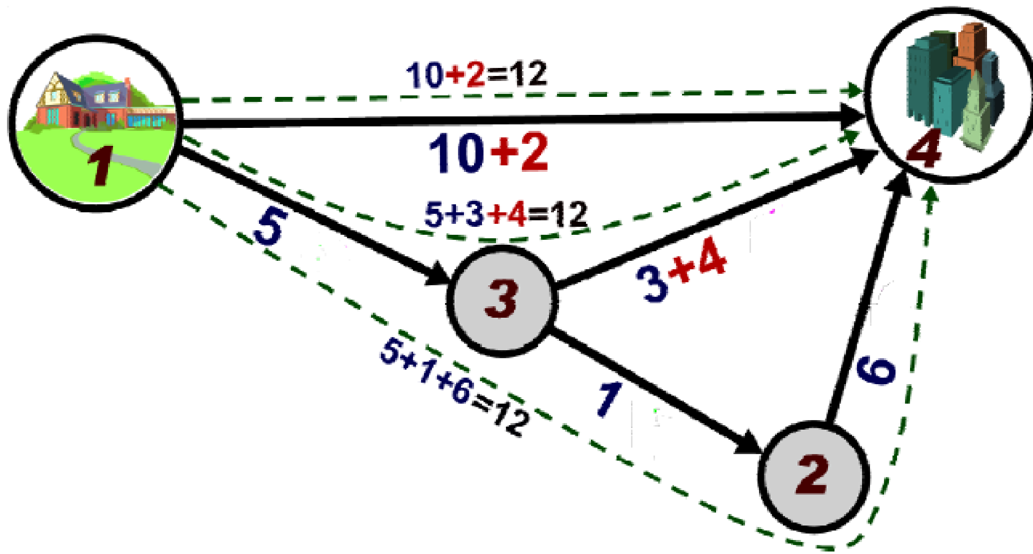


Figure 5: Roads connecting residential area at intersection 1 to business district at intersection 4

You must determine which roads should have tolls and how much each toll should be so that every route from start to finish has the same cost (driving time cost + possible toll) and no route contains more than one toll road. Additionally, the tolls should be chosen so as to minimize the final cost. In some settings, it might be impossible to impose tolls that satisfy the above conditions.

Input

Input consists of several test cases. A test case starts with a line containing an integer N ($2 \le N \le 50000$), which is the number of road intersections, and R ($1 \le R \le 50000$), which is the number of roads. Each of the next R lines contains three integers x_i, y_i, and c_i ($1 \le x_i, y_i \le N$, $1 \le c_i \le 1000$), indicating that morning traffic takes road i from intersection x_i to intersection y_i with a base driving time cost of c_i. Intersection 1 is the starting residential area, and intersection N is the goal business district. Roads are numbered from 1 to R in the given input order. Every intersection is part of a route from 1 to N, and there are no cycles.

The last test case is followed by a line containing two zeros.

Output

For each test case, print one line containing the case number (starting with 1), the number of roads to toll (T), and the final cost of every route. On the next T lines, print the road number i and the positive cost of the toll to apply to that road. If there are multiple minimal cost solutions, any will do. If there are none, print `No solution'. Follow the format of the sample output.

Sample Input

```
4 5
1 3 5
3 2 1
2 4 6
1 4 10
3 4 3
3 4
1 2 1
1 2 2
2 3 1
2 3 2
0 0
```

Sample Output

```
Case 1: 2 12
4 2
5 4
Case 2: No solution
```

```
/*
UVa 1083. Fare and Balanced
First of all, note that the given graph is always a DAG,
which allows us to run a DP algorithm without entering into
a cycle. Whenever you put a toll in a road, no one of the of
the roads reachable from there could have one, so all the paths
from there must be of the same length. Look at the code to
see what the algorithm does.
*/
#include <iostream>
#include <vector>
using namespace std;

#define PB push_back

typedef vector<int> Vi;
typedef vector<Vi> Mi;

const int INF = 1000000000;

int N, M;
Mi net, netr;
Mi net_cost, netr_cost;
Mi net_id, netr_id;

Vi dp, dpr, dpmax;

Vi vist, ares;

void aresta(int a, int b, int c, int id) {
  net[a].PB(b);
  net_cost[a].PB(c);
  net_id[a].PB(id);
  netr[b].PB(a);
  netr_cost[b].PB(c);
  netr_id[b].PB(id);
}

int calc_dp(int n) {
  if (dp[n] != -2) return dp[n];
  if (n == N - 1) return dp[n] = 0;
  int mini = INF, maxi = 0;
  for (int i = 0; i < net[n].size(); ++i) {
    int r = calc_dp(net[n][i]);
    if (r == -1) return dp[n] = -1;
    int t = net_cost[n][i] + r;
    mini = min(mini, t);
    maxi = max(maxi, t);
  }
  if (mini == maxi) return dp[n] = mini;
  return dp[n] = -1;
}

int calc_dpr(int n) {
  if (dpr[n] != -2) return dpr[n];
  if (n == 0) return dpr[n] = 0;
  int mini = INF, maxi = 0;
  for (int i = 0; i < netr[n].size(); ++i) {
    int r = calc_dpr(netr[n][i]);
    if (r == -1) return dpr[n] = -1;
    int t = netr_cost[n][i] + r;
```

```
    mini = min(mini, t);
    maxi = max(maxi, t);
  }
  if (mini == maxi) return dpr[n] = mini;
  return dpr[n] = -1;
}

int calc_dpmax(int n) {
  if (dpmax[n] != -2) return dp[n];
  int maxi = 0;
  for (int i = 0; i < net[n].size(); ++i)
    maxi = max(maxi, net_cost[n][i] + calc_dpmax(net[n][i]));
  return dpmax[n] = maxi;
}

void fun(int n) {
  if (vist[n]) return;
  vist[n] = 1;
  for (int i = 0; i < net[n].size(); ++i) {
    int m = net[n][i], c = net_cost[n][i], id = net_id[n][i];
    if (dp[m] == -1) fun(m);
    else if (dp[m] + c + dpr[n] < dpmax[0]) {
      ares[id] += dpmax[0] - (dp[m] + c + dpr[n]);
    }
  }
}

int main() {
  int cas = 1;
  while (cin >> N >> M and N > 0) {
    net = netr = Mi(N);
    net_cost = netr_cost = Mi(N);
    net_id = netr_id = Mi(N);
    for (int i = 0; i < M; ++i) {
      int a, b, c;
      cin >> a >> b >> c;
      --a; --b;
      aresta(a, b, c, i);
    }

    dp = dpr = dpmax = Vi(N, -2);
    for (int i = 0; i < N; ++i) calc_dp(i);
    for (int i = 0; i < N; ++i) calc_dpr(i);
    for (int i = 0; i < N; ++i) calc_dpmax(i);

    cout << "Case " << cas++ << ": ";

    bool ok = true;
    for (int i = 0; i < N; ++i) {
      if (dp[i] == -1 and dpr[i] == -1)
        ok = false;
    }

    if (not ok) {
      cout << "No solution" << endl;
      continue;
    }

    vist = Vi(N, 0);
    ares = Vi(M, 0);
    if (dp[0] == -1) fun(0);
```

```
    int ars = 0;
    for (int i = 0; i < M; ++i)
      if (ares[i]) ++ars;

    cout << ars << " " << dpmax[0] << endl;
    for (int i = 0; i < M; ++i)
      if (ares[i]) cout << i + 1 << " " << ares[i] << endl;
  }
}
```

UVa 1086. The Ministers' Major Mess

4452 - The Ministers' Major Mess

World Finals - Stockholm - 2008/2009

The ministers of the remote country of Stanistan are having severe problems with their decision making. It all started a few weeks ago when a new process for deciding which bills to pass was introduced. This process works as follows. During each voting session, there are several bills to be voted on. Each minister expresses an opinion by voting either ``yes" or ``no" for some of these bills. Because of limitations in the design of the technical solution used to evaluate the actual voting, each minister may vote on only at most four distinct bills (though this does not tend to be a problem, as most ministers only care about a handful of issues). Then, given these votes, the bills that are accepted are chosen in such a way that each minister gets more than half of his or her opinions satisfied.

As the astute reader has no doubt already realized, this process can lead to various problems. For instance, what if there are several possible choices satisfying all the ministers, or even worse, what if it is impossible to satisfy all the ministers? And even if the ministers' opinions lead to a unique choice, how is that choice found?

Your job is to write a program to help the ministers with some of these issues. Given the ministers' votes, the program must find out whether all the ministers can be satisfied, and if so, determine the decision on those bills for which, given the constraints, there is only one possible choice.

Input

Input consists of multiple test cases. Each test case starts with integers B ($1 \le B \le 100$), which is the number of distinct bills to vote on, and M ($1 \le M \le 500$), which is the number of ministers. The next M lines give the votes of the ministers. Each such line starts with an integer $1 \le k \le 4$, indicating the number of bills that the minister has voted on, followed by the k votes. Each vote is of the format < *bill* > < *vote* >, where < *bill* > is an integer between 1 and B identifying the bill that is voted on, and < *vote* > is either `y' or `n', indicating that the minister's opinion is ``yes" or ``no." No minister votes on the same bill more than once. The last test case is followed by a line containing two zeros.

Output

For each test case, print the test case number (starting with 1) followed by the result of the process. If it is impossible to satisfy all ministers, the result should be `impossible'. Otherwise, the result should be a string of length $B$, where the $i$-th character is `y', `n', or `?', depending on whether the decision on the i-th bill should be ``yes," whether it should be ``no," or whether the given votes do not determine the decision on this bill.

Sample Input

```
5 2
4 2 y 5 n 3 n 4 n
4 4 y 3 y 5 n 2 y
4 2
4 1 y 2 y 3 y 4 y
3 1 n 2 n 3 n
0 0
```

Sample Output

```
Case 1: ?y??n
Case 2: impossible
```

Stockholm 2008-2009

```
/*
UVa 1086. The Ministers' Major Mess
This problem can be solved as a 2-SAT, where each bill corresponds
to one variable, and the restrictions between variables are
imposed by the ministers' votes. Look at the code for the details.
In order to check whether a variable can be both false/true, force
that to be false and run the 2-SAT solver, then force it to be true
and run the solver again, if it was satisfiable in both cases,
then the value of that variable doesn't matter and you must print
an '?' in its place.
*/
#include <algorithm>
#include <iostream>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

#define PB push_back
#define X first
#define Y second

typedef pair<int, int> P;
typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef queue<int> Q;
typedef vector<P> Vp;

int N, M, C;
Mi net, netr;
Vi vist, comp, ord;

void dfsr(int n) {
  if (vist[n]) return;
  vist[n] = 1;
  for (int i = 0; i < netr[n].size(); ++i)
    dfsr(netr[n][i]);
  ord[--M] = n;
}

void dfs(int n) {
  if (vist[n]) return;
  vist[n] = 1;
  comp[n] = C;
  for (int i = 0; i < net[n].size(); ++i)
    dfs(net[n][i]);
}

void aresta(int a, int b) {
  net[a].PB(b);
  netr[b].PB(a);
}

bool sat() {
  int n = N/2;
  vist = Vi(N, 0);
  ord = Vi(N);
  M = N;
  for (int i = 0; i < N; ++i)
    dfsr(i);
```

```
  vist = Vi(N, 0);
  comp = Vi(N);
  C = 0;
  for (int i = 0; i < N; ++i)
    if (not vist[ord[i]]) {
      dfs(ord[i]);
      ++C;
    }

  for (int i = 0; i < n; ++i)
    if (comp[i] == comp[n + i])
      return false;
  return true;
}

int main() {
  int cas = 1;

  int n, m;
  while (cin >> n >> m and n > 0) {
    N = 2*n;
    net = netr = Mi(N);

    for (int i = 0; i < m; ++i) {
      int q;
      cin >> q;
      Vp v(q);
      for (int j = 0; j < q; ++j) {
        char c;
        cin >> v[j].X >> c;
        --v[j].X;
        v[j].Y = (c == 'y' ? 1 : 0);
      }

      if (q <= 2) {
        for (int j = 0; j < q; ++j) {
          if (v[j].Y == 0) aresta(v[j].X, n + v[j].X);
          else aresta(n + v[j].X, v[j].X);
        }
      }
      else {
        for (int j = 0; j < q; ++j) {
          int a = (v[j].Y == 0 ? v[j].X : n + v[j].X);
          for (int k = 0; k < q; ++k) {
            if (j == k) continue;
            int b = (v[k].Y == 0 ? n + v[k].X : v[k].X);
            aresta(a, b);
          }
        }
      }
    }

    cout << "Case " << cas++ << ": ";

    if (not sat()) cout << "impossible" << endl;
    else {
      string res(n, '?');
      for (int i = 0; i < n; ++i) {
        net[i].PB(n + i);
        netr[n + i].PB(i);
        bool no = sat();
```

```
        net[i].pop_back();
        netr[n + i].pop_back();

        net[n + i].PB(i);
        netr[i].PB(n + i);
        bool yes = sat();
        net[n + i].pop_back();
        netr[i].pop_back();

        if (no and not yes) res[i] = 'n';
        else if (not no and yes) res[i] = 'y';
      }
      cout << res << endl;
    }
  }
}
```

UVa 1087. Struts and Springs

4453 - Struts and Springs

World Finals - Stockholm - 2008/2009

Struts and springs are devices that determine the way in which rectangular windows on a screen are resized or repositioned when the enclosing window is resized. A window occupies a rectangular region of the screen, and it can enclose other windows to yield a hierarchy of windows. When the outermost window is resized, each of the immediately enclosed windows may change position or size (based on the placement of struts and springs); these changes may then affect the position and/or size of the windows they enclose.

A strut is conceptually a fixed length rod placed between the horizontal or vertical edges of a window, or between an edge of a window and the corresponding edge of the immediately enclosing window. When a strut connects the vertical or horizontal edges of a window, then the height or width of that window is fixed. Likewise, when a strut connects an edge of a window to the corresponding edge of the immediately enclosing window, then the distance between those edges is fixed. Springs, however, may be compressed or stretched, and may be used in place of struts.

Each window except the outermost has six struts or springs associated with it. One connects the vertical edges of the window, and another connects the horizontal edges of the window. Each of the other four struts or springs connects an edge of the window with the corresponding edge of the enclosing window. The sum of the lengths of the three vertical struts or springs equals the height of the enclosing window; similarly, the sum of the lengths of the three horizontal struts or springs equals the width of the enclosing window. When the enclosing window's width changes, any horizontal springs connected to the window are stretched or compressed in equal proportion, so that the new total length of struts and springs equals the new width. A similar action takes place for a change in the enclosing window's height. If all three vertical or horizontal components are struts, then the top or rightmost strut, respectively, is effectively replaced by a spring.

You must write a program that takes the initial sizes and positions of a set of windows (with one window guaranteed to enclose all others), the placement of struts and springs, and requests to resize the outermost window and then determines the modified size and position of each window for each size request.

Input

Input consists of multiple test cases corresponding to different sets of windows. Each test case begins with a line containing four integers *nwin*, *nresize*, *owidth*, and *oheight*. *nwin* is the number of windows (excluding the outer window which encloses all others), *nresize* is the number of outer window resize requests, and *owidth* and *oheight* are the initial width and height of the outer window.

Each of the next *nwin* lines contains 10 non-negative integers and describes a window. The first two integers give the initial *x* and *y* displacement of the upper left corner of the window with respect to the upper left corner of the outermost window. The next two integers give the initial width and height of the window. Each of the final six integers is either 0 (for a strut) or 1 (for a spring). The first two specify whether a strut or spring connects the vertical and horizontal edges of the window respectively, and the last four specify whether a strut or spring connects the tops, bottoms, left sides and right sides of the window and its immediately enclosing window.

Each of the last *nresize* lines in a test gives a new width and height for the outermost window - a resize operation. For each of these, your program must determine the size and placement of each of the *nwin* inner windows. The test data is such that, after every resizing operation, every strut and spring has a positive integral length, and different window boundaries do not touch. Also, resizing never causes one window to

jump inside another.

There are at most 100 windows and 100 resize operations in a test case, and the outermost window's width and height never exceed 1,000,000. The last test case is followed by a line with four zeros.

Output

For each resize operation in a test case, print the test case number (starting with 1) followed by the resize operation number (1, 2,...) on a line by itself. Then on each of the next *nwin* lines, print the window number, position (*x* and *y*) and size (width and height) of each of the inner windows of the test case as a result of resizing the outer window. Windows in each test case are numbered sequentially (1, 2,...) to match their position in the input, and should be output in that order. Follow the format shown in the sample output.

Sample Input

```
1 1 50 100
10 10 30 10 1 0 0 0 0 0
70 150
2 1 50 100
10 10 30 10 1 0 0 0 0 0
10 80 20 10 1 1 0 0 0 0
70 150
1 2 60 60
10 10 20 30 1 0 1 1 1 1
90 90
120 120
0 0 0 0
```

Sample Output

```
Case 1, resize operation 1:
    Window 1, x = 10, y = 60, width = 50, height = 10
Case 2, resize operation 1:
    Window 1, x = 10, y = 60, width = 50, height = 10
    Window 2, x = 10, y = 80, width = 40, height = 60
Case 3, resize operation 1:
    Window 1, x = 15, y = 20, width = 30, height = 30
Case 3, resize operation 2:
    Window 1, x = 20, y = 30, width = 40, height = 30
```

Stockholm 2008-2009

```
/*
UVa 1087. Struts and Springs
We can store the window structure as a tree where the outermost window is the
root, and every window is the child of the window that is its immediate
container. After the structure is built, for eqach resize operation we update
the width and height of the outermost window and recursively update all of its
children windows, according to the rules of springs and struts described in the
problem statement.
*/
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;

struct Window {
  ll x, y, width, height;
  vector<ll> hor, ver;
  vector<bool> hspr, vspr;
};

void rec_resize(const vector<vector<int> >& child, vector<Window>& win, int ind, ll
    nwidth, ll nheight) {
  win[ind].width = nwidth;
  win[ind].height = nheight;
  for (int i = 0; i < int(child[ind].size()); ++i) {
    Window& w = win[child[ind][i]];
    int xhave = 0, yhave = 0;
    int xsum = 0, ysum = 0;
    for (int j = 0; j < 3; ++j) {
      if (w.hspr[j]) {
        xsum += w.hor[j];
      }
      else {
        xhave += w.hor[j];
      }
      if (w.vspr[j]) {
        ysum += w.ver[j];
      }
      else {
        yhave += w.ver[j];
      }
    }
    for (int j = 0; j < 3; ++j) {
      if (w.hspr[j]) {
        w.hor[j] *= (nwidth-xhave);
        w.hor[j] /= xsum;
      }
      if (w.vspr[j]) {
        w.ver[j] *= (nheight-yhave);
        w.ver[j] /= ysum;
      }
    }
    w.x = win[ind].x + w.hor[0];
    w.y = win[ind].y + w.ver[0];
    rec_resize(child, win, child[ind][i], (win[ind].x+win[ind].width-w.hor[2])-w.x,
        (win[ind].y+win[ind].height-w.ver[2])-w.y);
  }
}
```

```
int main() {
  for (int nwin, nresize, owidth, oheight, ca = 1; cin >> nwin >> nresize >> owidth
      >> oheight && (nwin > 0 || nresize > 0 || owidth > 0 || oheight > 0); ++ca) {
    vector<Window> win(nwin+1);
    win[0].x = win[0].y = 0;
    win[0].width = owidth;
    win[0].height = oheight;
    for (int i = 1; i <= nwin; ++i) {
      cin >> win[i].x >> win[i].y >> win[i].width >> win[i].height;
      win[i].hspr = win[i].vspr = vector<bool>(3, false);
      int spr;
      cin >> spr;
      win[i].hspr[1] = (spr == 1);
      cin >> spr;
      win[i].vspr[1] = (spr == 1);
      cin >> spr;
      win[i].vspr[0] = (spr == 1);
      cin >> spr;
      win[i].vspr[2] = (spr == 1);
      cin >> spr;
      win[i].hspr[0] = (spr == 1);
      cin >> spr;
      win[i].hspr[2] = (spr == 1);
      int nhspr = 0, nvspr = 0;
      for (int j = 0; j < 3; ++j) {
        if (win[i].hspr[j]) {
          ++nhspr;
        }
        if (win[i].vspr[j]) {
          ++nvspr;
        }
      }
      if (nhspr == 0) {
        win[i].hspr[2] = true; // rightmost
      }
      if (nvspr == 0) {
        win[i].vspr[0] = true; // topmost
      }
    }
    vector<vector<int> > child(win.size());
    vector<int> fath(win.size(), -1);
    for (int i = 1; i <= nwin; ++i) {
      for (int j = 0; j <= nwin; ++j) if (j != i) {
        if (win[i].x >= win[j].x && win[i].x+win[i].width <= win[j].x+win[j].width
         && win[i].y >= win[j].y && win[i].y+win[i].height <= win[j].y+win[j].height
            ) {
          if (fath[i] == -1 || win[j].width*win[j].height < win[fath[i]].width*win[
              fath[i]].height) {
            fath[i] = j;
          }
        }
      }
      child[fath[i]].push_back(i);
    }
    for (int i = 1; i <= nwin; ++i) {
      win[i].hor = win[i].ver = vector<ll>(3);
      win[i].hor[0] = win[i].x-win[fath[i]].x;
      win[i].hor[1] = win[i].width;
      win[i].hor[2] = (win[fath[i]].x+win[fath[i]].width)-(win[i].x+win[i].width);

      win[i].ver[0] = win[i].y-win[fath[i]].y;
```

```
      win[i].ver[1] = win[i].height;
      win[i].ver[2] = (win[fath[i]].y+win[fath[i]].height)-(win[i].y+win[i].height);
    }
    for (int resiz = 1; resiz <= nresize; ++resiz) {
      int nwidth, nheight;
      cin >> nwidth >> nheight;
      rec_resize(child, win, 0, nwidth, nheight);
      cout << "Case " << ca << ", resize operation " << resiz << ":" << endl;
      for (int i = 1; i <= nwin; ++i) {
        cout << "    Window " << i << ", x = " << win[i].x << ", y = " << win[i].y
            << ", width = " << win[i].width << ", height = " << win[i].height <<
            endl;
      }
    }
  }
}
```

UVa 1089. Suffix-Replacement Grammars

4455 - Suffix-Replacement Grammars

World Finals - Stockholm - 2008/2009

As computer programmers, you have likely heard about regular expressions and context-free grammars. These are rich ways of generating sets of strings over a small alphabet (otherwise known as a formal language). There are other, more esoteric ways of generating languages, such as tree-adjoining grammars, context-sensitive grammars, and unrestricted grammars. This problem uses a new method for generating a language: a suffixreplacement grammar.

A suffix-replacement grammar consists of a starting string *S* and a set of suffix-replacement rules. Each rule is of the form $X \rightarrow Y$, where *X* and *Y* are equal-length strings of alphanumeric characters. This rule means that if the suffix (that is, the rightmost characters) of your current string is *X*, you can replace that suffix with *Y*. These rules may be applied arbitrarily many times.

For example, suppose there are 4 rules A $\rightarrow$ B, AB $\rightarrow$ BA, AA $\rightarrow$ CC, and CC $\rightarrow$ BB. You can then transform the string AA to BB using three rule applications: AA $\rightarrow$ AB (using the A $\rightarrow$ B rule), then AB $\rightarrow$ BA (using the AB $\rightarrow$ BA rule), and finally BA $\rightarrow$ BB (using the A $\rightarrow$ B rule again). But you can also do the transformation more quickly by applying only 2 rules: AA $\rightarrow$ CC and then CC $\rightarrow$ BB.

You must write a program that takes a suffix-replacement grammar and a string *T* and determines whether the grammar's starting string *S* can be transformed into the string *T*. If this is possible, the program must also find the minimal number of rule applications required to do the transformation.

Input

The input consists of several test cases. Each case starts with a line containing two equal-length alphanumeric strings *S* and *T* (each between 1 and 20 characters long, and separated by whitespace), and an integer *NR* ($0 \le NR \le 100$), which is the number of rules. Each of the next *NR* lines contains two equal-length

alphanumeric strings *X* and *Y* (each between 1 and 20 characters long, and separated by whitespace), indicating that $X \rightarrow Y$ is a rule of the grammar. All strings are case-sensitive. The last test case is followed by a line containing a period.

Output

For each test case, print the case number (beginning with 1) followed by the minimum number of rule applications required to transform *S* to *T*. If the transformation is not possible, print `` `No solution' ``. Follow the format of the sample output.

Sample Input

```
AA BB 4
A B
AB BA
AA CC
CC BB
A B 3
A C
B C
c B
.
```

Sample Output

```
Case 1: 2
Case 2: No solution
```

Stockholm 2008-2009

```
/*
UVa 1089. Suffix-Replacement Grammars
The idea here is: Run the Floyd-Warshall to find the minimum distance
between every pair of prefixes of length 1. Then, run a second
Floyd-Warshall, where some of the edges are determined by the result
of the first Floyd-Warshall, to determine the distance between every
pair of prefixes of length 2. The, run a third Floyd to determine
the distance between pairs of possible prefixes of length 3, and so
on. Until the last Floyd-Warshall determines the distance between all
pairs of words of maximum length, including the ones you needed to find.
*/
#include <iostream>
#include <map>
#include <string>
#include <vector>
using namespace std;

#define PB push_back
#define X first
#define Y second

typedef long long ll;
typedef map<string, int> MAP;
typedef MAP::iterator Mit;
typedef vector<ll> Vll;
typedef vector<Vll> Mll;
typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef vector<string> Vs;

int main() {
  int cas = 1;
  string A, B;
  int n;
  while (cin >> A >> B >> n and A != ".") {
    Vs va(n + 1), vb(n + 1);
    for (int i = 0; i < n; ++i) cin >> va[i] >> vb[i];
    va[n] = A;
    vb[n] = B;

    MAP mp;
    for (int i = 0; i <= n; ++i) {
      int m = va[i].size();
      for (int j = 0; j < m; ++j) {
        ++mp[va[i].substr(j, m - j)];
        ++mp[vb[i].substr(j, m - j)];
      }
    }

    int m = 0;
    for (Mit it = mp.begin(); it != mp.end(); ++it)
      it->Y = m++;

    Vs rev(m);
    for (Mit it = mp.begin(); it != mp.end(); ++it)
      rev[it->Y] = it->X;

    int ta = mp[A], tb = mp[B];
    Vi next(m);
    for (Mit it = mp.begin(); it != mp.end(); ++it) {
      string s = it->X;
```

```
      if (s.size() <= 1) next[it->Y] = -1;
      else next[it->Y] = mp[s.substr(1, s.size() - 1)];
    }

    Mi mat(21);
    for (int i = 0; i < m; ++i)
      mat[rev[i].size()].PB(i);

    Mll dist(m, Vll(m, -1));  // -1 means infinity
    for (int i = 0; i < n; ++i) dist[mp[va[i]]][mp[vb[i]]] = 1;
    for (int i = 0; i < m; ++i) dist[i][i] = 0;

    for (int sz = 1; sz <= A.size(); ++sz) {
      int s = mat[sz].size();

      for (int i = 0; i < s; ++i)
        for (int j = 0; j < s; ++j) {
          if (i == j) continue;
          int a = mat[sz][i], b = mat[sz][j];
          if (next[a] != -1 and next[b] != -1 and rev[a][0] == rev[b][0] and dist[
              next[a]][next[b]] != -1) {
            if (dist[a][b] == -1) dist[a][b] = dist[next[a]][next[b]];
            else dist[a][b] = min(dist[a][b], dist[next[a]][next[b]]);
          }
        }

      for (int k = 0; k < s; ++k)
        for (int i = 0; i < s; ++i)
          for (int j = 0; j < s; ++j) {
            if (i == j) continue;
            int a = mat[sz][i], b = mat[sz][j], c = mat[sz][k];
            if (dist[a][c] != -1 and dist[c][b] != -1) {
              if (dist[a][b] == -1) dist[a][b] = dist[a][c] + dist[c][b];
              else dist[a][b] = min(dist[a][b], dist[a][c] + dist[c][b]);
            }
          }

    }

    cout << "Case " << cas++ << ": ";
    if (dist[ta][tb] == -1) cout << "No solution" << endl;
    else cout << dist[ta][tb] << endl;
  }
}
```

1092 - Tracking Bio-bots

The researchers at International Bio-bot Makers (IBM) have invented a new kind of Bio-bot, a robot with behavior mimicking biological organisms. The development of the new robot is at a primitive stage; they now resemble simple four-wheeled rovers. And like most modern robots, Bio-bots are not very mobile. Their weak motors and limited turning capability put considerable limitations on their movement, even in simple, relatively obstacle-free environments.

Currently the Bio-bots operate in a room which can be described as an *m* x *n* grid. A Bio-bot occupies a full square on this grid. The exit is in the northeast corner, and the room slopes down towards it, which means the Bio-bots are only capable of moving north or east at any time. Some squares in the room are also occupied by walls, which completely block the robot. Figure 1, which corresponds to the sample input, shows an example of such a room.

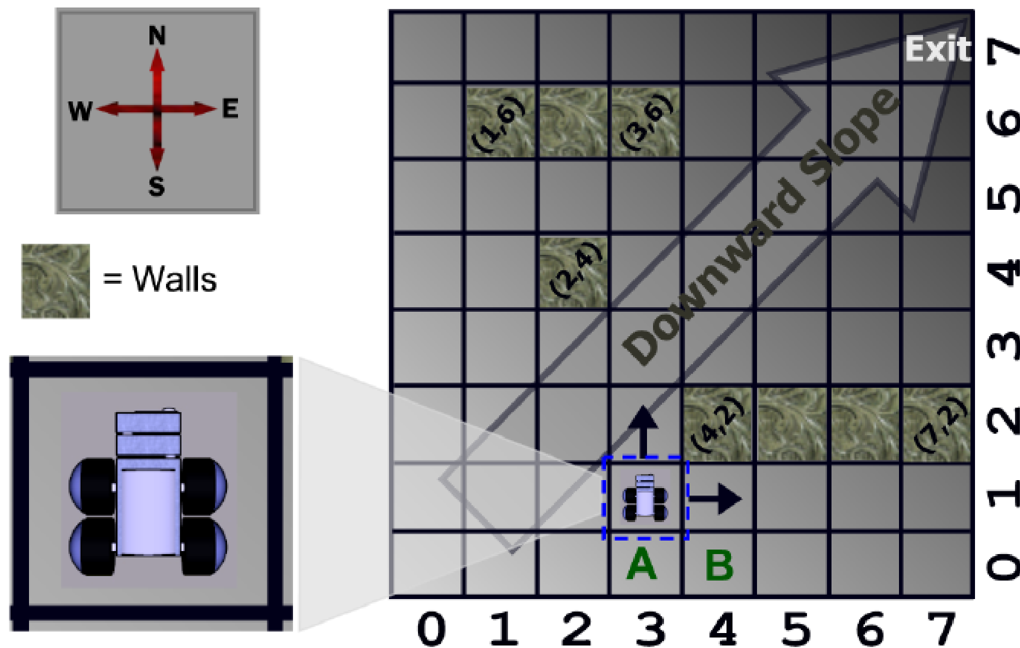


Figure 1

Clearly, a Bio-bot located on square A is capable of leaving the room, while one at square B is trapped inside it, no matter what it does. Locations like B are called ``stuck squares.'' (Walls do not count as stuck squares.) Given the description of a room, your job is to count the total number of stuck squares in the room.

Input

Input consists of multiple test cases, each describing one room. Each test case begins with a line containing three integers m, n, and w ($1 \le m, n \le 10^6$, $0 \le w \le 1000$). These indicate that the room contains m rows, n columns, and w horizontal walls.

Each of the next w lines contains four integers x_1, y_1, x_2, y_2, the coordinates of the squares delimiting one wall. All walls are aligned from west to east, so $0 \le x_1 \le x_2 < n$ and $0 \le y_1 = y_2 < m$. Walls do not overlap each other. The southwest corner of the room has coordinates (0,0) and the northeast corner has coordinates (n - 1, m - 1).

The last test case is followed by a line containing three zeros.

Output

For each test case, display one line of output containing the test case number followed by the number of stuck squares in the given room. Follow the format shown in the sample output.

Sample Input

```
8 8 3
1 6 3 6
2 4 2 4
4 2 7 2
0 0 0
```

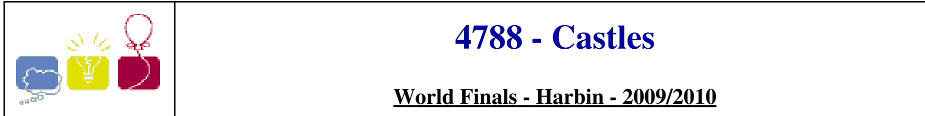
Sample Output

```
Case 1: 8
```

```
/*
UVa 1092. Tracking Bio-bots
First, compress the coordinates. That is, create a new matrix
where each cell represents a block of cells in the original
map. It is possible to create a matrix of O(dx*dy) cells (dx is
the number of different X's in the input, and dy the Y's), where
each cell represent a block of cells in the original map which
play the same role, so we can treat them as "the same".
Once the coordinates are compressed, a DP or BFS can be performed
over the compressed matrix to know what cells are reachable.
When computing the final answer, take into account the number
of cells in the original map that belong to each cell in the
compressed matrix.
*/
#include <iostream>
#include <map>
#include <set>
#include <vector>
using namespace std;

typedef long long ll;
typedef set<int> SET;
typedef map<int, int> MAP;
typedef vector<int> Vi;
typedef vector<Vi> Mi;

int main() {
  int cas = 1;
  int r, c, n;
  while (cin >> r >> c >> n and r > 0) {
    Vi wx1(n), wx2(n), wy(n);
    for (int i = 0; i < n; ++i)
      cin >> wx1[i] >> wy[i] >> wx2[i] >> wy[i];

    SET sx, sy;
    sx.insert(0);
    sx.insert(c);
    sy.insert(0);
    sy.insert(r);
    for (int i = 0; i < n; ++i) {
      sx.insert(wx1[i]);
      sx.insert(wx2[i] + 1);
      sy.insert(wy[i]);
      sy.insert(wy[i] + 1);
    }

    Vi vx(sx.begin(), sx.end()), vy(sy.begin(), sy.end());
    int nx = vx.size(), ny = vy.size();

    MAP mx, my;
    for (int i = 0; i < nx; ++i) mx[vx[i]] = i;
    for (int i = 0; i < ny; ++i) my[vy[i]] = i;

    Mi wall(ny, Vi(nx, 0));
    for (int i = 0; i < n; ++i) {
      int x1 = mx[wx1[i]], x2 = mx[wx2[i] + 1] - 1, y = my[wy[i]];
      for (int j = x1; j <= x2; ++j) wall[y][j] = 1;
    }

    Mi dp(ny, Vi(nx, 0));
    dp[ny - 2][nx - 2] = (wall[ny - 2][nx - 2] ? 0 : 1);
```

```
    for (int j = ny - 2; j >= 0; --j) {
      for (int i = nx - 2; i >= 0; --i)
        if (wall[j][i] == 0 and ((wall[j + 1][i] == 0 and dp[j + 1][i]) or (wall[j][
            i + 1] == 0 and dp[j][i + 1])))
          dp[j][i] = 1;
    }

    ll res = 0;
    for (int j = 0; j + 1 < ny; ++j)
      for (int i = 0; i + 1 < nx; ++i)
        if (wall[j][i] == 0 and dp[j][i] == 0)
          res += ll(vx[i + 1] - vx[i])*ll(vy[j + 1] - vy[j]);

    cout << "Case " << cas++ << ": " << res << endl;
  }
}
```

UVa 1093. Castles



Wars have played a significant role in world history. Unlike modern wars, armies in the middle ages were principally concerned with capturing and holding castles, the private fortified residences of lords and nobles. The size of the attacking army was an important factor in an army's ability to capture and hold one of these architectural masterpieces.

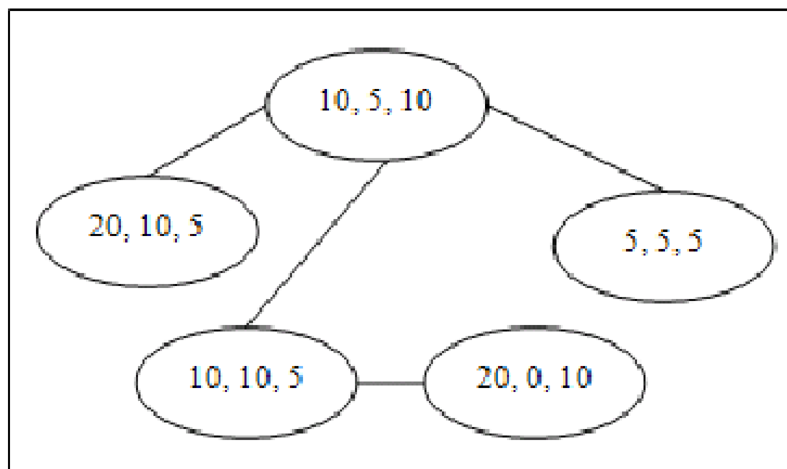


Figure 2

A certain minimum number of soldiers were required to capture a castle. Some soldiers were expected to die during the attack. After capturing the castle, some soldiers were required to remain in the castle to defend it against attacks from another enemy. Of course, those numbers were different for different castles. Commanders of the armies were obliged to consider the number of soldiers required for victory. For example, there are five castles in the region map shown in Figure 2. The castle at the lower right requires at least 20 soldiers to wage a winning attack. None are expected to perish during the attack, and 10 soldiers must be left in the castle when the army moves on.

In this problem you must determine the minimum size of an army needed to capture and hold all the castles in a particular region. For reasons of security, there is exactly one (bi-directional) route between any pair of castles in the region. Moving into the neighborhood of an uncaptured castle begins an attack on that castle. Any castle can serve as the first castle to be attacked, without regard for how the army got there. Once any castle has been captured, the requisite number of soldiers is left in the castle to defend it, and the remainder of the army moves on to do battle at another castle, if any remain uncaptured. The army may safely pass through the neighborhood of a castle that it has already captured. But because of the potential for attacks, the army may traverse the route between a pair of castles no more than twice (that is, at most once in each direction).

Input

The input contains multiple test cases corresponding to different regions. The description of the castles in each region occupies several lines. The first line contains an integer $n \leq 100$ that is the number of castles in the region. Each of the next n lines contains three integers a, m, and g ($1 \leq a \leq 1000$, $0 \leq m \leq a$, $1 \leq g \leq 1000$), that give the minimum number of soldiers required to successfully attack and capture a particular castle, the number of soldiers that are expected to die during the attack, and the number of soldiers that must be left at the castle to defend it. The castles are numbered 1 to n, and the input lines describing them are given in increasing order of castle numbers. Each of the remaining n - 1 lines in a test case has two integers that specify the castle numbers of a pair of castles that are connected by a direct route.

A line containing `0` follows the description of the last region.

Output

For each test case, display the case number and the minimum number of soldiers in the army needed to conquer all the castles in the region. Follow the format shown in the sample output.

Sample Input

```
3
5 5 5
10 5 5
5 1 1
1 3
2 3
5
10 5 10
20 10 5
10 10 5
5 5 5
20 0 10
1 2
1 3
1 4
3 5
0
```

Sample Output

```
Case 1: 22
Case 2: 65
```

Harbin 2009-2010

```cpp
/*
UVa 1093. Castles
There exists an optimal solutions in which a simple exploration of
the tree is made, that is, every edge is traversed at most 2 times
(one in each direction).
Run a DFS from each node (in other words, root the tree at every
node and run the DFS from the root each time) to calculate the number
of soldiers needed if one starts at that node. Calculate that number
for each subtree. The order of sending of conquering the children
in an optimal solution is to start first from the children that
return more soldiers back, and following an increasing order
for this value.
*/
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>
using namespace std;

#define PB push_back
#define X first
#define Y second

typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef pair<int, int> P;
typedef vector<P> Vp;

int N;
Vi A, M, G;
Mi net;

bool sortf(P a, P b) {
  return a.Y > b.Y;
}

P fun(int n, int m) {
  P res(max(A[n], M[n] + G[n]), max(0, A[n] - M[n] - G[n])); // (necessito, tornen)

  int s = net[n].size();
  Vp v(s);
  for (int i = 0; i < net[n].size(); ++i) {
    if (net[n][i] == m) continue;
    v[i] = fun(net[n][i], n);
  }
  sort(v.begin(), v.end(), sortf);

  for (int i = 0; i < s; ++i) {
    res.X += max(0, v[i].X - res.Y);
    res.Y = max(0, res.Y - v[i].X);
    res.Y += v[i].Y;
  }

  return res;
}

int main() {
  int cas = 1;
  while (cin >> N and N > 0) {
    A = M = G = Vi(N);
    for (int i = 0; i < N; ++i) cin >> A[i] >> M[i] >> G[i];
```

```
    net = Mi(N);
    for (int i = 0; i < N - 1; ++i) {
      int a, b;
      cin >> a >> b;
      --a; --b;
      net[a].PB(b);
      net[b].PB(a);
    }

    int res = fun(0, -1).X;
    for (int i = 1; i < N; ++i)
      res = min(res, fun(i, -1).X);
    cout << "Case " << cas++ << ": " << res << endl;
  }
}
```

UVa 1096. The Islands

4791 - The Islands

World Finals - Harbin - 2009/2010

Wen Chen is the captain of a rescue boat. One of his important tasks is to visit a group of islands once a day to check if everything is all right. Captain Wen starts from the west-most island, makes a pass to the east-most island visiting some of the islands, then makes a second pass from the east-most island back to the first one visiting the remaining islands. In each pass Captain Wen moves steadily east (in the first pass) or west (in the second pass), but moves as far north or south as he needs to reach the islands. The only complication is that there are two special islands where Wen gets fuel for his boat, so he must visit them in separate passes. Figure 7 shows the two special islands in pink (1 and 3) and one possible path Captain Wen could take.

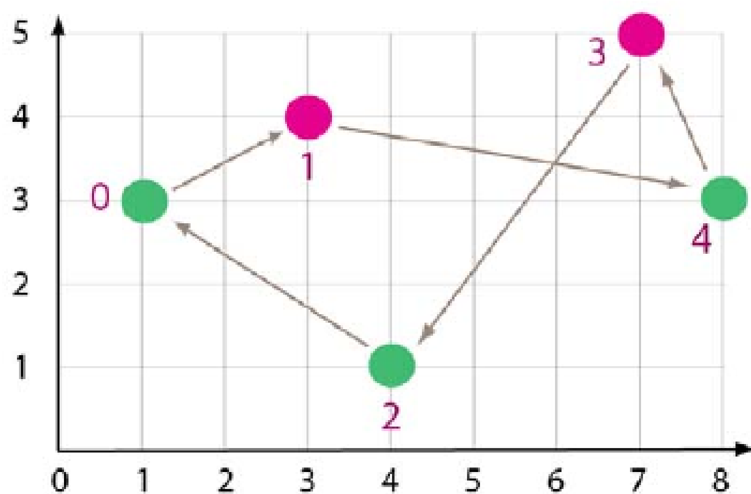


Figure 7

Calculate the length of the shortest path to visit all the islands in two passes when each island's location and the identification of the two special islands are given.

Input

The input consists of multiple test cases. The data for each case begins with a line containing 3 integers n $(4 \le n \le 100)$, b_1, and b_2 $(0 < b_1, b_2 < n - 1$ and $b_1 \ne b_2)$, where n is the number of islands (numbered 0 to n - 1) and b_1 and b_2 are the two special islands. Following this, there are n lines containing the integer x - and y -coordinates of each island $(0 \le x, y \le 2000)$, starting with island 0. No two islands have the same x -coordinate and they are in order from west-most to east-most (that is, minimum x -coordinate to maximum x -coordinate).

Input for the last case is followed by a line containing 3 zeroes.

Output

For each case, display two lines. The first line contains the case number and the length of the shortest tour Captain Wen can take to visit all the islands, rounded and displayed to the nearest hundredth. The second line contains a space-separated list of the islands in the order that they should be visited, starting with island 0 and 1, and ending with island 0. Each test case will have a unique solution. Follow the format in the sample output.

Sample Input

```
5 1 3
1 3
3 4
4 1
7 5
8 3
5 3 2
0 10
3 14
4 7
7 10
8 12
0 0 0
```

Sample Output

```
Case 1: 18.18
0 1 4 3 2 0
Case 2: 24.30
0 1 3 4 2 0
```

Harbin 2009-2010

```
/*
UVa 1096. The Islands
This problem can be solved with Dynamic Programming with
a time complexity of O(N^2), where N is the number of islands.
In the code below, dp[a][b] is the cost of a two disjoint paths
from island 'a' to the rightmost one, and from island 'b' to the
rightmost one, traversing only islands in a position greater (to
the right of) than the island at the max(a,b)-th position.
*/
#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;

#define PB push_back

typedef vector<int> Vi;

int N, N1, N2, VX[110], VY[110];

int tdp[110][110], T;
int que[110][110];
double dp[110][110];

double dist(int a, int b) {
  int x = VX[b] - VX[a];
  int y = VY[b] - VY[a];
  return sqrt(x*x + y*y);
}

double fun(int a, int b) {
  if (tdp[a][b] == T) return dp[a][b];
  tdp[a][b] = T;

  int n = max(a, b) + 1;
  if (n == N - 1) return dp[a][b] = dist(a, n) + dist(b, n);

  double ta = dist(a, n) + fun(n, b);
  double tb = dist(b, n) + fun(a, n);

  if (n == N1) {
    que[a][b] = 1;
    return dp[a][b] = ta;
  }

  if (n == N2) {
    que[a][b] = 2;
    return dp[a][b] = tb;
  }

  if (ta < tb) {
    que[a][b] = 1;
    return dp[a][b] = ta;
  }

  que[a][b] = 2;
  return dp[a][b] = tb;
}

int main() {
```

```
  cout.setf(ios::fixed);
  cout.precision(2);

  int cas = 1;
  while (cin >> N >> N1 >> N2 and N > 0) {
    for (int i = 0; i < N; ++i) cin >> VX[i] >> VY[i];

    ++T;
    cout << "Case " << cas++ << ": " << fun(0, 0) << endl;

    Vi va(1, 0), vb(1, 0);
    int a = 0, b = 0;
    while (max(a, b) + 1 < N - 1) {
      int t = max(a, b) + 1;
      if (que[a][b] == 1) {
        va.PB(t);
        a = t;
      }
      else {
        vb.PB(t);
        b = t;
      }
    }
    va.PB(N - 1);
    vb.PB(N - 1);

    int na = va.size(), nb = vb.size();
    Vi v;
    for (int i = 0; i < na; ++i) v.PB(va[i]);
    for (int i = nb - 2; i >= 0; --i) v.PB(vb[i]);

    int n = v.size();
    if (v[1] != 1) reverse(v.begin(), v.end());
    for (int i = 0; i < n; ++i) {
      if (i) cout << " ";
      cout << v[i];
    }
    cout << endl;
  }
}
```

1099 - Sharing Chocolate

Chocolate in its many forms is enjoyed by millions of people around the world every day. It is a truly universal candy available in virtually every country around the world.

You find that the only thing better than eating chocolate is to share it with friends. Unfortunately your friends are very picky and have different appetites: some would like more and others less of the chocolate that you offer them. You have found it increasingly difficult to determine whether their demands can be met. It is time to writte a program that solves the problem once and for all!

Your chocolate comes as a rectangular bar. The bar consists of same-sized rectangular pieces. To share the chocolate you may break one bar into two pieces along a division between rows or columns of the bar. You or the may then repeatedly break the resulting pieces in the same manner. Each of your friends insists on a getting a single rectangular portion of the chocolate that has a specified number of pieces. You are a little bit insistent as well: you will break up your bar only if all of it can be distributed to your friends, with none left over.

For exampla, Figure 9 shows one way that a chocolate bar consisting of 3 x 4 pieces can be split into 4 parts that contain 6, 3, 2, and 1 pieces respectively, by breanking it 3 times (This corresponds to the first sample input.)

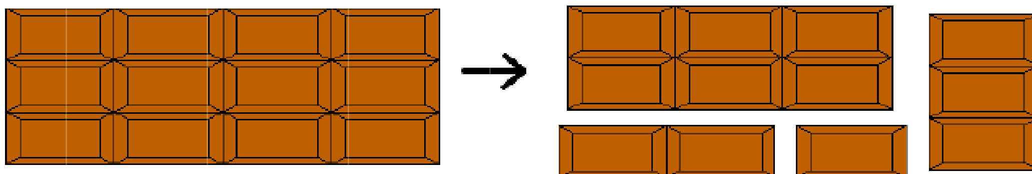


Figure 9

Input

The input consists of multiple test cases each describing a chocolate bar to share. Each description starts with a line containing a single integer n ($1 \leq n \leq 15$), the number of parts in which the bar is supposed to be split. This is followed by a line containing two integers x and y ($1 \leq x, y \leq 100$), the dimensions of the chocolate bar. The next line contains n positive integers, giving the number of pieces that are supposed to be in each of the n parts.

The input is terminated by a line containing the integer zero.

Output

For each test case, first display its case number. Then display whether it is possible to break the chocolate in the desired way: display ``Yes" if it is possible, and ``No" otherwise. Follow the format of the sample output.

Sample Input

```
4
3 4
6 3 2 1
2
2 3
1 5
0
```

Sample Output

```
Case 1: Yes
Case 2: No
```

```
/*
UVa 1099. Sharing Chocolate
This problem has been solved with Dynamic Programming.
The number of different states is O(M*2^N), where N is
the number of parts in which the chocolate bar has to
be splitted, and M is the lenth of the longest side of
the initial bar. The state of the DP dp[s][m] is
defined by the size of the current bar, and which
parts have to be made using that bar. 's' represents
the length of one of the sides of the bar, 'm' is a
mask representing which parts have to be obtained from
the bar, note that the length of the second side of the
bar is implicitly given by 's' and 'm'.
*/
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> Vi;

int N, R, C, V[15], S[1<<15];
bool dp[110][1<<15];
int tdp[110][1<<15], T;
Vi sub[1<<15];

int tmp[15];
void compute_sub(int m) {
  int n = 0;
  for (int i = 0; (1<<i) <= m; ++i)
    if ((m>>i)&1) tmp[n++] = (1<<i);
  sub[m] = Vi(1<<n, 0);
  for (int i = 0; i < (1<<n); ++i)
    for (int j = 0; j < n; ++j)
      if ((i>>j)&1) sub[m][i] |= tmp[j];
}

bool fun(int a, int m) {
  if (__builtin_popcount(m) == 1) return true;
  int b = S[m]/a;
  if (a > b) swap(a, b);
  if (tdp[a][m] == T) return dp[a][m];
  tdp[a][m] = T;
  if (sub[m].size() == 0) compute_sub(m);
  int s = sub[m].size();
  for (int i = 0; i < s; ++i) {
    int x = sub[m][i];
    if (x == 0 or x == m) continue;
    if (S[x]%a == 0 and S[m^x]%a == 0 and fun(a, x) and fun(a, m^x)) {
      return dp[a][m] = true;
    }
    if (S[x]%b == 0 and S[m^x]%b == 0 and fun(b, x) and fun(b, m^x)) {
      return dp[a][m] = true;
    }
  }
  return dp[a][m] = false;
}

int main() {
  int cas = 1;
  while (cin >> N and N > 0) {
    cin >> R >> C;
```

```
    for (int i = 0; i < N; ++i) cin >> V[i];

    cout << "Case " << cas++ << ": ";

    int suma = 0;
    for (int i = 0; i < N; ++i) suma += V[i];
    if (suma != R*C) {
      cout << "No" << endl;
      continue;
    }

    for (int i = 0; i < (1<<N); ++i) {
      S[i] = 0;
      for (int j = 0; j < N; ++j)
        if ((i>>j)&1) S[i] += V[j];
    }

    ++T;
    if (fun(R, (1<<N) - 1)) cout << "Yes" << endl;
    else cout << "No" << endl;
  }
}
```

UVa 1103. Ancient Messages

2011 World Finals
acm International Collegiate Programming Contest
IBM event sponsor

Problem C
Ancient Messages
Problem ID: ancient

In order to understand early civilizations, archaeologists often study texts written in ancient languages. One such language, used in Egypt more than 3000 years ago, is based on characters called hieroglyphs. Figure C.1 shows six hieroglyphs and their names. In this problem, you will write a program to recognize these six characters.

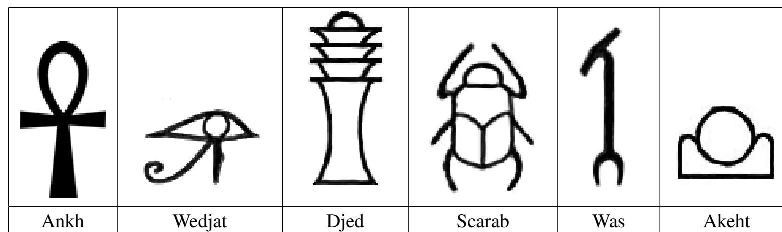


Figure C.1: Six hieroglyphs

Input

The input consists of several test cases, each of which describes an image containing one or more hieroglyphs chosen from among those shown in Figure C.1. The image is given in the form of a series of horizontal scan lines consisting of black pixels (represented by 1) and white pixels (represented by 0). In the input data, each scan line is encoded in hexadecimal notation. For example, the sequence of eight pixels `10011100` (one black pixel, followed by two white pixels, and so on) would be represented in hexadecimal notation as `9c`. Only digits and lowercase letters `a` through `f` are used in the hexadecimal encoding. The first line of each test case contains two integers, H and W. H ($0 < H \leq 200$) is the number of scan lines in the image. W ($0 < W \leq 50$) is the number of hexadecimal characters in each line. The next H lines contain the hexadecimal characters of the image, working from top to bottom. Input images conform to the following rules:

- The image contains only hieroglyphs shown in Figure C.1.
- Each image contains at least one valid hieroglyph.
- Each black pixel in the image is part of a valid hieroglyph.
- Each hieroglyph consists of a connected set of black pixels and each black pixel has at least one other black pixel on its top, bottom, left, or right side.
- The hieroglyphs do not touch and no hieroglyph is inside another hieroglyph.
- Two black pixels that touch diagonally will always have a common touching black pixel.
- The hieroglyphs may be distorted but each has a shape that is topologically equivalent to one of the symbols in Figure C.1[1].

The last test case is followed by a line containing two zeros.

[1] Two figures are topologically equivalent if each can be transformed into the other by stretching without tearing.

Output

For each test case, display its case number followed by a string containing one character for each hieroglyph recognized in the image, using the following code:

Ankh: `A`
Wedjat: `J`
Djed: `D`
Scarab: `S`
Was: `W`
Akhet: `K`

In each output string, print the codes in alphabetic order. Follow the format of the sample output.

The sample input contains descriptions of test cases shown in Figures C.2 and C.3. Due to space constraints not all of the sample input can be shown on this page.

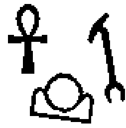


Figure C.2: AKW

Figure C.3: AAAAA

Sample input	Output for the Sample Input
100 25 0000000000000000000000000 0000000000000000000000000 ...(50 lines omitted)... 00001fe0000000000007c0000 00003fe0000000000007c0000 ...(44 lines omitted)... 0000000000000000000000000 0000000000000000000000000 150 38 00000000000000000000000000000000000000 00000000000000000000000000000000000000 ...(75 lines omitted)... 0000000003fffffffffffffffff00000000000 0000000003fffffffffffffffff00000000000 ...(69 lines omitted)... 00000000000000000000000000000000000000 00000000000000000000000000000000000000 0 0	Case 1: AKW Case 2: AAAAA

ICPC 2011 World Finals Problem C: Ancient Messages

```cpp
/*
UVa 1103. Ancient Messages
First of all, identify the regions of the figures. To do so, run a
BFS from the outside of the map and mark all the cells reachable
without trespassing any black cell. Each island of non-marked
connected cells corresponds to one figure. Now, inside each island,
count the number of islands of white connected cells. This number
identifies the figure in one of the 6 categories.
*/
#include <iostream>
#include <queue>
#include <string>
#include <utility>
#include <vector>
using namespace std;

#define X first
#define Y second

typedef pair<int, int> P;
typedef queue<P> Q;
typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef vector<bool> Vb;
typedef vector<Vb> Mb;

const int diri[4] = { -1, 0, 1, 0 };
const int dirj[4] = { 0, 1, 0, -1 };

int R, C;
Mb mapa;
Mb vist;
Mi color;

int todec(char c) {
  if ('0' <= c and c <= '9') return c - '0';
  if ('A' <= c and c <= 'Z') return 10 + c - 'A';
  return 10 + c - 'a';
}

void bfs_pinta(int sr, int sc, int col) {
  Q q;
  q.push(P(sr, sc));
  vist[sr][sc] = true;
  while (q.size()) {
    int r = q.front().X, c = q.front().Y;
    q.pop();
    color[r][c] = col;

    for (int k = 0; k < 4; ++k) {
      int i = r + diri[k], j = c + dirj[k];
      if (0 <= i and i < R and 0 <= j and j < C and mapa[i][j] and not vist[i][j]) {
        vist[i][j] = true;
        q.push(P(i, j));
      }
    }
  }
}

int bfs(int sr, int sc) {
  int res = -2;
```

```
  Q q;
  q.push(P(sr, sc));
  vist[sr][sc] = true;
  while (q.size()) {
    int r = q.front().X, c = q.front().Y;
    q.pop();

    for (int k = 0; k < 4; ++k) {
      int i = r + diri[k], j = c + dirj[k];
      if (0 <= i and i < R and 0 <= j and j < C) {
        if (mapa[i][j]) {
          if (res != -1) {
            if (res == -2) res = color[i][j];
            else if (res != color[i][j]) res = -1;
          }
        }
        else {
          if (not vist[i][j]) {
            vist[i][j] = true;
            q.push(P(i, j));
          }
        }
      }
      else res = -1;
    }
  }
  return res;
}

int main() {
  int cas = 1;
  int h, w;
  while (cin >> h >> w and h > 0) {
    R = h;
    C = 4*w;
    mapa = Mb(R, Vb(C));
    for (int i = 0; i < h; ++i) {
      string s;
      cin >> s;
      for (int j = 0; j < w; ++j) {
        int d = todec(s[j]);
        for (int k = 0; k < 4; ++k)
          mapa[i][4*j + 3 - k] = (d>>k)&1;
      }
    }

    int comps = 0;
    vist = Mb(R, Vb(C, false));
    color = Mi(R, Vi(C));
    for (int i = 0; i < R; ++i)
      for (int j = 0; j < C; ++j)
        if (mapa[i][j] and not vist[i][j])
          bfs_pinta(i, j, comps++);

    Vi holes(comps, 0);
    for (int i = 0; i < R; ++i)
      for (int j = 0; j < C; ++j)
        if (not mapa[i][j] and not vist[i][j]) {
          int t = bfs(i, j);
          if (t != -1) ++holes[t];
```

```
      }

    Vi type(6, 0);
    for (int i = 0; i < comps; ++i)
      ++type[holes[i]];

    cout << "Case " << cas++ << ": ";
    cout << string(type[1], 'A');
    cout << string(type[5], 'D');
    cout << string(type[3], 'J');
    cout << string(type[2], 'K');
    cout << string(type[4], 'S');
    cout << string(type[0], 'W');
    cout << endl;
  }
}
```

UVa 1105. Coffee Central

Problem E
Coffee Central
Problem ID: coffee

Is it just a fad or is it here to stay? You're not sure, but the steadily increasing number of coffee shops that are opening in your hometown has certainly become quite a draw. Apparently, people have become so addicted to coffee that apartments that are close to many coffee shops will actually fetch higher rents.

This has come to the attention of a local real-estate company. They are interested in identifying the most valuable locations in the city in terms of their proximity to large numbers of coffee shops. They have given you a map of the city, marked with the locations of coffee shops. Assuming that the average person is willing to walk only a fixed number of blocks for their morning coffee, you have to find the location from which one can reach the largest number of coffee shops. As you are probably aware, your hometown is built on a square grid layout, with blocks aligned on north-south and east-west axes. Since you have to walk along streets, the distance between intersections (a, b) and (c, d) is $|a - c| + |b - d|$.

Input

The input contains several test cases. Each test case describes a city. The first line of each test case contains four integers dx, dy, n, and q. These are the dimensions of the city grid $dx \times dy$ ($1 \le dx, dy \le 1000$), the number of coffee shops n ($0 \le n \le 5 \cdot 10^5$), and the number of queries q ($1 \le q \le 20$). Each of the next n lines contains two integers x_i and y_i ($1 \le x_i \le dx$, $1 \le y_i \le dy$); these specify the location of the ith coffee shop. There will be at most one coffee shop per intersection. Each of the next q lines contains a single integer m ($0 \le m \le 10^6$), the maximal distance that a person is willing to walk for a cup of coffee.

The last test case is followed by a line containing four zeros.

Output

For each test case in the input, display its case number. Then display one line per query in the test case. Each line displays the maximum number of coffee shops reachable for the given query distance m followed by the optimal location. For example, the sample output shows that 3 coffee shops are within query distance 1 of the optimal location $(3, 4)$, 4 shops are within query distance 2 of optimal location $(2, 2)$, and 5 shops are within query distance 4 of optimal location $(3, 1)$. If there are multiple optimal locations, pick the location that is furthest south (minimal positive integer y-coordinate). If there is still a tie, pick the location furthest west (minimal positive integer x-coordinate).

Follow the format of the sample output.

ICPC 2011 World Finals Problem E: Coffee Central

Sample input	Output for the Sample Input
4 4 5 3 1 1 1 2 3 3 4 4 2 4 1 2 4 0 0 0 0	Case 1: 3 (3,4) 4 (2,2) 5 (3,1)

ICPC 2011 World Finals Problem E: Coffee Central

```
/*
UVa 1105. Coffee Central
Consider this new coordinate system where a point (x,y) of the original system
becomes (x',y') in the following way: (x',y') = (x+y,x-y)
Fixed a point 'p' and a distance 'd' in the original sytem, the area of points
at distance 'd' of 'p' or closer according to the Manhattan distance form the
shape of a diamond. If we look at these points in the new coordinate system,
they form a square (with sides parallel to the axes).
So, for each query, we want to find the optimal position of a square of side 2*m
in the new coordinate system that maximizes the number of coffee shops contained
inside. We can calculte this number for every possible position of the square
in constant time. To do so, we have to precalculate the table sum[][], where
sum[x][y] = sum of mat[i][j] for 1<=i<=x and 1<=j<=y. This table filled properly
allows us to calculate the sum of cells of any given rectangle with sides parallel
to the axes.
*/
#include <iostream>
#include <utility>
#include <vector>
using namespace std;

#define X first
#define Y second

typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef pair<int, int> P;
typedef vector<P> Vp;

int N, DX, DY, M;
Vp V, U;

Mi mat, sum;

P fun(int x, int y) {
  return P(x + y, x - y + DY - 1);
}

P inv(int x, int y) {
  return P((x + y - (DY - 1))/2, (x - (y - (DY - 1)))/2);
}

void compute_sum() {
  sum = Mi(M, Vi(M, 0));
  sum[0][0] = mat[0][0];
  for (int i = 1; i < M; ++i) sum[0][i] = sum[0][i - 1] + mat[0][i];
  for (int i = 1; i < M; ++i) sum[i][0] = sum[i - 1][0] + mat[i][0];
  for (int i = 1; i < M; ++i)
    for (int j = 0; j < M; ++j)
      sum[i][j] = mat[i][j] + sum[i - 1][j] + sum[i][j - 1] - sum[i - 1][j - 1];
}

int suma(int x1, int y1, int x2, int y2) {
  if (x1 == 0 and y1 == 0) return sum[x2][y2];
  if (x1 == 0) return sum[x2][y2] - sum[x2][y1 - 1];
  if (y1 == 0) return sum[x2][y2] - sum[x1 - 1][y2];
  return sum[x2][y2] - sum[x1 - 1][y2] - sum[x2][y1 - 1] + sum[x1 - 1][y1 - 1];
}

int main() {
  int cas = 1;
```

```
  int q;
  while (cin >> DX >> DY >> N >> q and DX > 0) {
    M = DX + DY - 1;
    V = U = Vp(N);
    for (int i = 0; i < N; ++i) {
      cin >> V[i].X >> V[i].Y;
      --V[i].X;
      --V[i].Y;
    }

    mat = Mi(M, Vi(M, 0));
    for (int i = 0; i < N; ++i) {
      P p = fun(V[i].X, V[i].Y);
      mat[p.X][p.Y] = 1;
    }

    compute_sum();

    cout << "Case " << cas++ << ":" << endl;
    while (q--) {
      int m;
      cin >> m;

      int res = 0;
      int rx = 0, ry = 0;

      for (int i = 0; i < M; ++i)
        for (int j = 0; j < M; ++j) {
          if (((i + j)&1) != ((DY - 1)&1)) continue;

          int x1 = max(0, i - m);
          int y1 = max(0, j - m);
          int x2 = min(M - 1, i + m);
          int y2 = min(M - 1, j + m);

          int t = suma(x1, y1, x2, y2);
          P p = inv(i, j);

          if (p.X < 0 or DX <= p.X or p.Y < 0 or DY <= p.Y) continue;

          if (t > res or (t == res and (p.Y < ry or (p.Y == ry and p.X < rx)))) {
            res = t;
            rx = p.X;
            ry = p.Y;
          }
        }

      cout << res << " (" << rx + 1 << "," << ry + 1 << ")" << endl;
    }
  }
}
```

UVa 1107. Magic Sticks

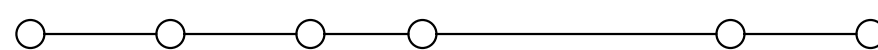
2011 World Finals
acm International Collegiate Programming Contest

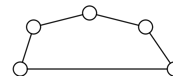
event sponsor

Problem G
Magic Sticks
Problem ID: magicsticks

Magic was accepted by all ancient peoples as a technique to compel the help of divine powers. In a well-known story, one group of sorcerers threw their walking sticks on the floor where they magically appeared to turn into live serpents. In opposition, another person threw his stick on the floor, where it turned into a serpent which then consumed the sorcerers' serpents!

The only magic required for this problem is its solution. You are given a magic stick that has several straight segments, with joints between the segments that allow the stick to be folded. Depending on the segment lengths and how they are folded, the segments of the stick can be arranged to produce a number of polygons. You are to determine the maximum area that could be enclosed by the polygons formed by folding the stick, using each segment in at most one polygon. Segments can touch only at their endpoints. For example, the stick shown below on the left has five segments and four joints. It can be folded to produce a polygon as shown on the right.





Input

The input contains several test cases. Each test case describes a magic stick. The first line in each test case contains an integer n ($1 \le n \le 500$) which indicates the number of the segments in the magic stick. The next line contains n integers $S_1, S_2, \ldots, S_n$ ($1 \le S_i \le 1000$) which indicate the lengths of the segments in the order they appear in the stick.

The last test case is followed by a line containing a single zero.

Output

For each case, display its case number followed by the maximum total enclosed area that can be obtained by folding the magic stick at the given points. Answers within an absolute or relative error of 10^{-4} will be accepted.

Follow the format of the sample output.

Sample input	Output for the Sample Input
4 1 2 3 4 8 3 4 5 33 3 4 3 5 0	Case 1: 4.898979 Case 2: 19.311

ICPC 2011 World Finals Problem G: Magic Sticks

```
/*
UVa 1107. Magic Sticks
The arrangement of a chain of edges that gives maximal area is always one of
the following:
(1) Form a polygon of maximal area using the whole chain of edges, if possible.
(2) Pick the longest edge of the chain and leave it out of any polygon, then
arrange optimally the chain of edges at each side of the longest edge
recursively. This process can be implemented with a Divide and Conquer
algorithm.
*/
#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> Vi;

double area(Vi v) {
  int n = v.size();
  if (n < 3) return 0;

  sort(v.begin(), v.end());

  int s = 0;
  for (int i = 0; i < n - 1; ++i) s += v[i];
  if (s <= v[n - 1]) return 0;

  double e = 0.5*v[n - 1], d = 1e+20;
  for (int step = 0; step < 100; ++step) {
    double m = 0.5*e + 0.5*d;
    double t = 0;
    for (int i = 0; i < n - 1; ++i)
      t += 2*asin(0.5*v[i]/m);
    if (t > 2*M_PI) e = m;
    else {
      double x = m*cos(t) - m, y = m*sin(t);
      double z = sqrt(x*x + y*y);
      if (z < v[n - 1]) e = m;
      else d = m;
    }
  }

  double a = 0, t = 0;
  for (int i = 0; i < n - 1; ++i) {
    t += 2*asin(0.5*v[i]/e);
    a += 0.5*v[i]*sqrt(e*e - 0.25*v[i]*v[i]);
  }
  if (t < M_PI) a -= 0.5*v[n - 1]*sqrt(e*e - 0.25*v[n - 1]*v[n - 1]);
  else a += 0.5*v[n - 1]*sqrt(e*e - 0.25*v[n - 1]*v[n - 1]);
  return a;
}

double dc(Vi v) {
  double res = area(v);

  int n = v.size();
  if (n < 3) return 0;

  int m = 0, q = 0;
  for (int i = 0; i < n; ++i)
```

```
    if (v[i] > m) {
      m = v[i];
      q = i;
    }

  res = max(res, dc(Vi(v.begin(), v.begin() + q)) + dc(Vi(v.begin() + q + 1, v.end()
      )));
  return res;
}

int main() {
  cout.setf(ios::fixed);
  cout.precision(10);

  int cas = 1;

  int n;
  while (cin >> n and n > 0) {
    Vi v(n);
    for (int i = 0; i < n; ++i) cin >> v[i];

    cout << "Case " << cas++ << ": " << dc(v) << endl;
  }
}
```

UVa 1110. Pyramids

event sponsor

Problem J
Pyramids
Problem ID: pyramids

It is not too hard to build a pyramid if you have a lot of identical cubes. On a flat foundation you lay, say, 10×10 cubes in a square. Centered on top of that square you lay a 9×9 square of cubes. Continuing this way you end up with a single cube, which is the top of the pyramid. The height of such a pyramid equals the length of its base, which in this case is 10. We call this a *high* pyramid.

If you think that a high pyramid is too steep, you can proceed as follows. On the 10×10 base square, lay an 8×8 square, then a 6×6 square, and so on, ending with a 2×2 top square (if you start with a base of odd length, you end up with a single cube on top, of course). The height of this pyramid is about half the length of its base. We call this a *low* pyramid.

Once upon a time (quite a long time ago, actually) there was a pharaoh who inherited a large number of stone cubes from his father. He ordered his architect to use all of these cubes to build a pyramid, not leaving a single one unused. The architect kindly explained that not every number of cubes can form a pyramid. With 10 cubes you can build a low pyramid with base 3. With 5 cubes you can build a high pyramid of base 2. But no pyramid can be built using exactly 7 cubes.

The pharaoh was not amused, but after some thinking he came up with new restrictions.

1. All cubes must be used.
2. You may build more than one pyramid, but you must build as few pyramids as possible.
3. All pyramids must be different.
4. Each pyramid must have a height of at least 2.
5. Satisfying the above, the largest of the pyramids must be as large as possible (i.e., containing the most cubes).
6. Satisfying the above, the next-to-largest pyramid must be as large as possible.
7. And so on...

Drawing figures and pictures in the sand, it took the architect quite some time to come up with the best solution.

Write a program that determines how to meet the restrictions of the pharaoh, given the number of cubes.

Input

The input consists of several test cases, each one on a single line. A test case is an integer c, where $1 \leq c \leq 10^6$, giving the number of cubes available.

The last test case is followed by a line containing a single zero.

Output

For each test case, display its case number followed by the pyramids to be built. The pyramids should be ordered with the largest first. Pyramids are specified by the length of their base followed by an `L` for low pyramids or an `H` for high pyramids. If two differenct pyramids have the same number of cubes, list the high pyramid first. Print "`impossible`" if it is not possible to meet the requirements of the pharaoh.

Follow the format of the sample output.

Sample input	Output for the Sample Input
29 28 0	Case 1: 3H 3L 2H Case 2: impossible

ICPC 2011 World Finals Problem J: Pyramids

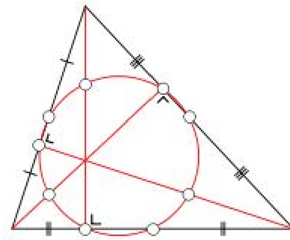
```cpp
/*
UVa 1110. Pyramids
This problem can be solved using Dynamic Programming, defining a state (i, j) as
the best solution to the problem using exactly i cubes and pyramids of at least
j cubes. Iterate over j decreasingly, computing (i, j) for every i by trying for
every k to build a pyramid of k cubes, checking if the solution resulting of
adding the new pyramid to (i-k, j+1) is better than the best so far.
*/
#include <algorithm>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

#define FR first
#define SC second

using namespace std;

typedef long long ll;

const int MAXC = 1000000;
const int INF = 1000000000;

bool by_pyramid(const pair<int, string>& a, const pair<int, string>& b) {
  if (a.FR != b.FR) {
    return a.FR < b.FR;
  }
  return a.SC[a.SC.size()-1] > b.SC[b.SC.size()-1];
}

string int_to_string(int n) {
  stringstream ss;
  ss << n << endl;
  string ret;
  ss >> ret;
  return ret;
}

int main() {
  vector<int> high;
  high.push_back(0);
  for (int b = int(high.size()); high[b-1]+b*b <= MAXC; ++b) {
    high.push_back(high[b-1]+b*b);
  }
  vector<int> low;
  low.push_back(0);
  low.push_back(1);
  for (int b = int(low.size()); low[b-2]+b*b <= MAXC; ++b) {
    low.push_back(low[b-2]+b*b);
  }

  vector<pair<int, string> > coins;
  for (int b = 2; b < int(high.size()); ++b) {
    coins.push_back(pair<int, string>(high[b], int_to_string(b)+"H"));
  }
  for (int b = 3; b < int(low.size()); ++b) {
    coins.push_back(pair<int, string>(low[b], int_to_string(b)+"L"));
  }
  sort(coins.begin(), coins.end(), by_pyramid);
  vector<int> len(MAXC+1, INF);
```

```
  vector<ll> sol(MAXC+1, -1);
  len[0] = 0;
  sol[0] = 0;
  for (int c = int(coins.size())-1; c >= 0; --c) {
    for (int i = int(sol.size())-1; i >= coins[c].FR; --i) {
      if (len[i-coins[c].FR] == INF) {
        continue;
      }
      if (len[i] == INF || len[i-coins[c].FR]+1 < len[i] || (len[i-coins[c].FR]+1 ==
           len[i] && ((sol[i-coins[c].FR]<<9)|c) > sol[i])) {
        sol[i] = (sol[i-coins[c].FR]<<9)|c;
        len[i] = len[i-coins[c].FR]+1;
      }
    }
  }

  for (int ca = 1, c; cin >> c && c > 0; ++ca) {
    cout << "Case " << ca << ":";
    if (len[c] == INF) {
      cout << " impossible" << endl;
    }
    else {
      for (int i = 0; i < len[c]; ++i) {
        cout << " " << coins[(sol[c]>>(9*(len[c]-1-i)))&((1<<9)-1)].SC;
      }
      cout << endl;
    }
  }
}
```

C. Nine-Point Circle

In geometry, the nine-point circle is a circle that can be constructed for any given triangle. It is so named because it passes through nine significant points defined from the triangle. These nine points are:

- The midpoint of each side of the triangle
- The foot of each altitude
- The midpoint of the line segment from each vertex of the triangle to the orthocenter (where the three altitudes meet; these line segments lie on their respective altitudes).



The nine-point circle is also known as Feuerbach's circle, Euler's circle, Terquem's circle, the six-point circle, the twelve-point circle, the n-point circle, the medioscribed circle, the mid circle or the circum-midcircle.

Given three non-collinear points A, B and C, you're to calculate the center position and radius of triangle ABC's nine-point circle.

Input

There will be at most 100 test cases. Each case contains 6 integers x1, y1, x2, y2, x3, y3 (0 <= x1, y1, x2, y2, x3, y3 <= 1000), the coordinates of A, B and C. The last test case is followed by a line with six -1, which should not be processed.

Output

For each test case, print three real numbers x, y, r, indicating that the nine point circle is centered at (x,y), with radius r. The numbers should be rounded to six decimal places.

Sample Input

```
0 0 10 0 3 4
-1 -1 -1 -1 -1 -1
```

Output for Sample Input

```
4.000000 2.312500 2.519456
```

```
/*
UVa 12302. Nine-Point Circle
An easy way to compute the center and radius of the Nine-Point Circle is to
compute the center and radius of the circle that goes through the midpoints
of the three sides of the triangle.
*/
#include <cmath>
#include <iostream>

#define x first
#define y second

using namespace std;

typedef pair<double, double> PDD;
typedef long double ld;

const double EPS = 1e-9, INF = 1e30;


bool int_rectes(ld x1, ld y1, ld vx1, ld vy1, ld x2, ld y2, ld vx2, ld vy2, ld& x,
    ld& y) {
  if (abs(vx2*vy1 - vy2*vx1) <= EPS) return false;
  ld t2 = (y2*vx1 - y1*vx1 - x2*vy1 + x1*vy1)/(vx2*vy1 - vy2*vx1);
  x = x2 + t2*vx2;
  y = y2 + t2*vy2;
  return true;
}

PDD circle_center(PDD a, PDD b, PDD c) {
  PDD m1((a.x+b.x)/2.0, (a.y+b.y)/2.0);
  PDD m2((b.x+c.x)/2.0, (b.y+c.y)/2.0);
  PDD v(-(b.y-a.y), b.x-a.x);
  PDD w(-(c.y-b.y), c.x-b.x);
  ld ox, oy;
  int_rectes(m1.x, m1.y, v.x, v.y, m2.x, m2.y, w.x, w.y, ox, oy);
  return PDD(ox, oy);
}

double dist(PDD a, PDD b) {
  return sqrt((a.x-b.x)*(a.x-b.x) + (a.y-b.y)*(a.y-b.y));
}

int main() {
  cout.setf(ios::fixed);
  cout.precision(6);
  for (PDD a, b, c; cin >> a.x >> a.y >> b.x >> b.y >> c.x >> c.y;) {
    if (a.x == -1.0 && a.y == -1.0 && b.x == -1.0 && b.y == -1.0 && c.x == -1.0 && c
        .y == -1.0) break;
    PDD m1, m2, m3;
    m1.x = (a.x+b.x)/2.0;
    m1.y = (a.y+b.y)/2.0;
    m2.x = (b.x+c.x)/2.0;
    m2.y = (b.y+c.y)/2.0;
    m3.x = (c.x+a.x)/2.0;
    m3.y = (c.y+a.y)/2.0;
    PDD o = circle_center(m1, m2, m3);
    cout << o.x << " " << o.y << " " << dist(o, m1) << endl;
  }
}
```

Problem H

Smallest Enclosing Rectangle

There are n points in 2D space. You're to find a smallest enclosing rectangle of these points. By "smallest" we mean either area or perimeter (yes, you have to solve both problems. The optimal rectangle for these two problems might be different). Note that the sides of the rectangle might not be parallel to the coordinate axes.

Input

There will be at most 10 test cases in the input. Each test case begins with a single integer n (3<=n<=100,000), the number of points. Each of the following n lines contains two real numbers x, y (-100,000<=x,y<=100,000), the coordinates of the points. The points will not be collinear. The last test case is followed by a line with n=0, which should not be processed.

Output

For each line, print the area of the minimum-area enclosing rectangle, and the perimeter of the minimum-perimeter enclosing rectangle, both rounded to two decimal places.

Sample Input

```
5
0 0
2 0
2 2
0 2
1 1
5
1 1
9 0
7 10
0 5
2 11
3
5 3
7 2
6 6
4
6 3
9 1
9 6
8 10
0
```

Output for the Sample Input

```
4.00 8.00
95.38 39.19
7.00 11.38
27.00 23.63
```

Rujia Liu's Present 4: A Contest Dedicated to Geometry and CG Lovers
Special Thanks: Dun Liang

```
/*
UVa 12307. Smallest Enclosing Rectangle
Both minimal rectangles (in area and perimeter) must have some side that
cointains at least 2 points, because otherwise it would be possible to shrink
and rotate the rectangle to get a (at least slightly) smaller rectangle.
Actually, these 2 points must belong the the convex hull of the set of points,
and each side of the convex hull polygon determines a potential minimal
rectangle. We can iterate over all the sides of the convex hull, keeping track
of the 3 points of the convex hull that determine a lower bound on the
dimensions of the rectangle and updating them in an amortized time of O(n)
using the fact that the 3 points for the next side are next to the corresponding
3 points for the current side. For each rectangle compute its area and
perimeter, updating the best area and perimeter seen so far.
*/
#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>

#define x first
#define ang first
#define y second
#define mod second

using namespace std;

typedef pair<double, double> PDD;

const double EPS = 1e-9, INF = 1e30;

double modulo(PDD v) {
  return sqrt(v.x*v.x + v.y*v.y);
}

// (AB x BC)_z
double cross_prod(PDD a, PDD b, PDD c) {
  return (b.x-a.x)*(c.y-b.y) - (c.x-b.x)*(b.y-a.y);
}

// (u x v)_z
double cross_prod(PDD u, PDD v) {
  return u.x*v.y - v.x*u.y;
}

inline PDD vect(PDD a, PDD b) {
  return PDD(b.x-a.x, b.y-a.y);
}

inline PDD rotate_ccw(PDD v) {
  return PDD(-v.y, v.x);
}

inline PDD rotate_180(PDD v) {
  return PDD(-v.x, -v.y);
}

inline PDD rotate_cw(PDD v) {
  return PDD(v.y, -v.x);
}

inline PDD cart2polar(PDD v) {
```

```
  return PDD(atan2(v.y, v.x), modulo(v));
}

inline PDD polar2cart(PDD v) {
  return PDD(cos(v.ang)*v.mod, sin(v.ang)*v.mod);
}

vector<PDD> convex_hull(vector<PDD>& p) {
  sort(p.begin(), p.end());
  vector<PDD> upper(p.size());
  int k = 0;
  for (int i = 0; i < int(p.size()); ++i) {
    for (; k >= 2 && cross_prod(upper[k-2], upper[k-1], p[i]) > 0.0;) --k;
    upper[k++] = p[i];
  }
  upper.resize(k);
  vector<PDD> lower(p.size());
  k = 0;
  for (int i = int(p.size())-1; i >= 0; --i) {
    for (; k >= 2 && cross_prod(lower[k-2], lower[k-1], p[i]) > 0.0;) --k;
    lower[k++] = p[i];
  }
  lower.resize(k);
  vector<PDD> hull(int(upper.size())+int(lower.size())-2);
  k = 0;
  for (int i = 0; i < int(upper.size())-1; ++i) {
    hull[k++] = upper[i];
  }
  for (int i = 0; i < int(lower.size())-1; ++i) {
    hull[k++] = lower[i];
  }
  return hull;
}

void rotate_rectangle(PDD v, PDD& a, PDD& b, PDD& c, PDD& d) {
  double alpha = atan2(v.y, v.x);
  PDD pa = cart2polar(vect(a, a));
  pa.ang -= alpha;
  PDD pb = cart2polar(vect(a, b));
  pb.ang -= alpha;
  PDD pc = cart2polar(vect(a, c));
  pc.ang -= alpha;
  PDD pd = cart2polar(vect(a, d));
  pd.ang -= alpha;
  a = polar2cart(pa);
  b = polar2cart(pb);
  c = polar2cart(pc);
  d = polar2cart(pd);
}

double area(PDD v, PDD a, PDD b, PDD c, PDD d) {
  rotate_rectangle(v, a, b, c, d);
  double mnx = min(min(a.x, b.x), min(c.x, d.x));
  double mxx = max(max(a.x, b.x), max(c.x, d.x));
  double mny = min(min(a.y, b.y), min(c.y, d.y));
  double mxy = max(max(a.y, b.y), max(c.y, d.y));
  return (mxx-mnx)*(mxy-mny);
}

double perimeter(PDD v, PDD a, PDD b, PDD c, PDD d) {
  rotate_rectangle(v, a, b, c, d);
```

```
  double mnx = min(min(a.x, b.x), min(c.x, d.x));
  double mxx = max(max(a.x, b.x), max(c.x, d.x));
  double mny = min(min(a.y, b.y), min(c.y, d.y));
  double mxy = max(max(a.y, b.y), max(c.y, d.y));
  return 2.0*(mxx-mnx) + 2.0*(mxy-mny);
}

int main() {
  ios_base::sync_with_stdio(false);
  cout.setf(ios::fixed);
  cout.precision(2);
  for (int n; cin >> n && n > 0;) {
    vector<PDD> p(n);
    for (int i = 0; i < n; ++i) {
      cin >> p[i].x >> p[i].y;
    }
    p = convex_hull(p);
    int n = int(p.size());
    double mnperim = INF, mnarea = INF;
    int i = 0, s1 = 0, s2 = 0, s3 = 0;
    PDD v0 = vect(p[i], p[(i+1)%n]);
    PDD v1 = rotate_cw(v0);
    PDD v2 = rotate_180(v0);
    PDD v3 = rotate_ccw(v0);
    for (; cross_prod(vect(p[s1], p[(s1+1)%n]), v1) < -EPS;) s1 = (s1+1)%n;
    for (; cross_prod(vect(p[s2], p[(s2+1)%n]), v1) < -EPS;) s2 = (s2+1)%n;
    for (; cross_prod(vect(p[s2], p[(s2+1)%n]), v2) < -EPS;) s2 = (s2+1)%n;
    for (; cross_prod(vect(p[s3], p[(s3+1)%n]), v1) < -EPS;) s3 = (s3+1)%n;
    for (; cross_prod(vect(p[s3], p[(s3+1)%n]), v2) < -EPS;) s3 = (s3+1)%n;
    for (; cross_prod(vect(p[s3], p[(s3+1)%n]), v3) < -EPS;) s3 = (s3+1)%n;
    for (i = 0; i < n; ++i) {
      v0 = vect(p[i], p[(i+1)%n]);
      v1 = rotate_cw(v0);
      v2 = rotate_180(v0);
      v3 = rotate_ccw(v0);
      for (; cross_prod(vect(p[s1], p[(s1+1)%n]), v1) < -EPS;) s1 = (s1+1)%n;
      for (; cross_prod(vect(p[s2], p[(s2+1)%n]), v2) < -EPS;) s2 = (s2+1)%n;
      for (; cross_prod(vect(p[s3], p[(s3+1)%n]), v3) < -EPS;) s3 = (s3+1)%n;
      mnarea = min(mnarea, area(v0, p[i], p[s1], p[s2], p[s3]));
      mnperim = min(mnperim, perimeter(v0, p[i], p[s1], p[s2], p[s3]));
    }
    cout << mnarea << " " << mnperim << endl;
  }
}
```

UVa 12350. Queen Game

E

Problem E
Queen Game

Queen game is played in a chessboard of size R rows and C columns. Rows are numbered from 1 to R and columns are numbered from 1 to C. The topmost square is in row 1 and column 1. The game is a 2 player game. Initially there are N queens placed in various squares of the chessboard. In his turn, the player picks a queen and moves it either towards the top vertically, or towards the left horizontally or towards the top-left diagonally and the queen should always stay on the board. When the queen reaches square (1, 1) it is removed from the board. The player who gives the last move wins. Each square is big enough to accommodate infinite number of queens. The players give their moves by turns. You are given the size of the chessboard and the initial positions of the N queens. Assuming that both of the players play perfectly your task is to determine who will win this game.

Input

First line of the input contains T the number of test cases. Each test case starts with a line containing 3 integers R($1 \le R \le 25$), C($1 \le C \le 10^{15}$) and N($1 \le N \le 1000$). Each of the next N lines contains the positions of N queens. The position is denoted by two integers. The first integer is the row number and the second integer is the column numbers.

Output

For each test case output "YES" if the first player has a winning strategy and "NO" otherwise.

Look at the output for sample input for details.

Sample Input	Sample output
3 5 5 1 2 3 5 5 2 4 4 4 4 5 5 3 1 2 2 1 2 2	NO NO YES

1 ACM-ICPC Asia Phuket Regional Programming Contest – 4 November 2011 – PSU, Phuket Campus

```
/*
UVa 12350. Queen Game
The Nimber of a state S is defined as the smallest natural number that is not
the Nimber of any state reachable from S (0 for losing states), and the Nimber
theorem says that the Nimber of a union of independent games can be obtained as
the XOR (bitwise exclusive or) of the Nimbers of each of the subgames.

The game described in the problem statement is a union of independent games in
which each player plays in one game of his choice per turn. We could compute
the Nimber of each state with a single queen and then compute the XOR of the
Nimber of the game corresponding to each queen. This approach, however, is too
slow because the number of columns of the board can be as high as 10^15.

Nevertheless, if we look at the matrix of Nimber values where the number at
(x, y) is the Nimber of the game with a single queen at cell (x, y), we can
notice that the difference between a number and the neighbour to its right
is cyclic in each row (with different period in different rows). With the help
of another program we can obtain the cycle length in each row, and once we have
this information it is possible to compute the Nimber of a state (x, y) without
computing the Nimber of all the states reachable from it, thus speeding things
up greatly.
*/
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;

ll cycle[] = {1, 3, 3, 6, 12, 24, 12, 24, 24, 24, 24, 48, 48, 96, 96, 96, 192, 192,
    384, 384, 384, 768, 768, 768, 768};

vector<vector<ll> > nimber(25, vector<ll>(3800));

ll get_nimber(ll x, ll y) {
  if (x < y) swap(x, y);
  if (x < ll(nimber[y].size())) return nimber[y][x];
  ll steps = (x-(ll(nimber[y].size())-1)+cycle[y]-1)/cycle[y];
  ll x2 = x - steps*cycle[y];
  return nimber[y][x2] + cycle[y]*steps;
}

int main() {
  nimber[0][0] = 0;
  for (int i = 1; i < int(nimber.size()); ++i) {
    nimber[i][0] = i;
  }
  for (int i = 1; i < int(nimber[0].size()); ++i) {
    nimber[0][i] = i;
  }
  ll mxsofar = max(int(nimber.size()), int(nimber[0].size()));
  for (int i = 1; i < int(nimber.size()); ++i) {
    for (int j = 1; j < int(nimber[i].size()); ++j) {
      vector<bool> seen(mxsofar+1, false);
      for (int k = 1; i-k >= 0; ++k) {
        seen[nimber[i-k][j]] = true;
      }
      for (int k = 1; j-k >= 0; ++k) {
        seen[nimber[i][j-k]] = true;
      }
      for (int k = 1; i-k >= 0 && j-k >= 0; ++k) {
```

```
        seen[nimber[i-k][j-k]] = true;
      }
      int mn = 0;
      for (; mn < int(seen.size()) && seen[mn];) ++mn;
      nimber[i][j] = mn;
      mxsofar = max(mxsofar, nimber[i][j]);
    }
  }
  int t;
  cin >> t;
  for (; t--;) {
    ll r, c, n;
    cin >> r >> c >> n;
    ll nim = 0;
    for (int i = 0; i < n; ++i) {
      ll y, x;
      cin >> y >> x;
      nim ^= get_nimber(x-1, y-1);
    }
    if (nim == 0) cout << "NO" << endl;
    else cout << "YES" << endl;
  }
}
```

BUET Inter-University Programming Contest

Problem A – Answering Queries on a Tree

Problem

You are given a tree with **N** nodes. The tree nodes are numbered from **1** to **N** and have colors $\mathbf{C_1, C_2 \dots C_N}$ initially. You have to handle **M** instructions on the tree of the following forms:

- **0 u c**: Change the color of node **u** to **c**.
- **1 u v**: Output the maximum number of times a color appeared on the unique path from node **u** to node **v**.

Input

The first line of input contains **T** $(\mathbf{1 \le T \le 10})$ which is the number of test cases. The first line of each test case contains two integers **N** and **M** $(\mathbf{1 \le N, M \le 10^5})$. Next line contains **N** space separated integers $\mathbf{C_1, C_2, \dots C_N}$ $(1 \le C_i \le 10)$ denoting the initial colors of the nodes. Each of the next **N-1** lines contain two integers **a** and **b** $(\mathbf{1 \le a, b \le N}$ **and** $\mathbf{a \ne b})$ meaning that there is an edge between node **a** and node **b**. Each of the next **M** lines contains an instruction of one of the two forms described above. For all the instructions: $\mathbf{1 \le u, v \le N\ and\ 1 \le c \le 10}$.

Output

For each of the second type instruction output the answer in one line.

Sample Input	Output for Sample Input
2 5 6 3 2 1 2 3 1 2 2 3 2 4 1 5 1 3 5 0 1 1 0 2 1 1 3 5 0 2 4 1 2 4 2 1 5 6 1 2 1 2 2	2 3 1 1

Problemsetter: Tasnim Imran Sunny

Special Thanks: Muntasir Mashuq and Tanaeem M Moosa

```
/*
UVa 12424. Answering Queries on a Tree
Given that there are few different colors (10 at most), in order to compute the
maximum number of nodes of the same color in a path, we can compute the number
of nodes of each color and keep the maximum. Let f(u,v,c) be the number of nodes
of color c that are in the unique path between u and v. If we know the value of
f(r,v,c) for every v, where r is the root of the tree (we fix the root
arbitrarily), then we can obtain f(u,v,c) with the following formula:
f(u,v,c) = f(r,u,c) + f(r,v,c) - f(r,lca(u,v),c) - f(r,parent(lca(u,v)))
If lca(u,v) = r, then the formula changes slightly (we have to ignore the last
term). Note: lca(u,v) = Lowest Common Ancestor of u and v. There exist
algorithms to compute the LCA in constant time after a linear-time computation.
In this solution, we have implemented a simplier version of the algorithm that
makes a precomputation in O(n*log(n)) and asks to each question in O(log(n)).
So, if we can compute f(r,v,c) (for aswering queries of type 1) and we can
change the color of an arbitrary node (queries of type 0), then we have the
solution. For each color c, we will have stored in a data structure the number
of nodes of color c in the path from the root r to v, for every v. This data
structure allows us to answer queries in O(log(n)). In each query of type 0 we
update this data structure. Observe that when we change the color of a node u,
this change affects only to the values f(r,v,c) with v belonging to u's subtree.
If we enumerate the nodes in preorder, then the nodes affected by a query of
type 0 have consecutive numbers and this allows us to update the interval
efficiently. In this implemenation we have used an Interval Tree, which allows
us to update intervals in O(log(n)) and make queries in O(log(n)) too.
*/
#include <iostream>
#include <vector>
#include <queue>
#include <stack>
#include <cstdio>
using namespace std;

#define PB push_back

typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef queue<int> Q;
typedef stack<int> S;

const int LMAX = 17;

int N;
Mi net;
Vi pare;
Vi prof;

Vi color;

Mi dp;
Vi et1, et2;
Mi trees;

void bfs() {
  prof = Vi(N, -1);
  pare = Vi(N, -1);

  prof[0] = 0;

  Q q;
  q.push(0);
```

```
  while (not q.empty()) {
    int n = q.front();
    q.pop();
    for (int i = 0; i < int(net[n].size()); ++i) {
      int m = net[n][i];
      if (prof[m] == -1) {
        prof[m] = prof[n] + 1;
        pare[m] = n;
        q.push(m);
      }
    }
  }
}

void compute_dp() {
  dp = Mi(N, Vi(LMAX + 1));
  for (int i = 0; i < N; ++i) dp[i][0] = pare[i];
  for (int j = 1; j <= LMAX; ++j)
    for (int i = 0; i < N; ++i) {
      if (dp[i][j - 1] == -1) dp[i][j] = -1;
      else dp[i][j] = dp[dp[i][j - 1]][j - 1];
    }
}

void compute_et() {
  et1 = et2 = Vi(N, -1);
  int et = 0;
  S st;
  st.push(0);
  while (not st.empty()) {
    int n = st.top();
    st.pop();

    if (et1[n] != -1) {
      et2[n] = et - 1;
      continue;
    }

    et1[n] = et++;

    st.push(n);
    for (int i = 0; i < int(net[n].size()); ++i) {
      int m = net[n][i];
      if (m != pare[n]) st.push(m);
    }
  }
}

int lca(int a, int b) {
  if (prof[a] < prof[b]) swap(a, b);

  int d = prof[a] - prof[b];
  for (int i = 0; (1<<i) <= d; ++i)
    if ((d>>i)&1) a = dp[a][i];
  if (a == b) return a;

  for (int k = LMAX; k >= 0; --k) {
    if (dp[a][k] != dp[b][k]) {
      a = dp[a][k];
      b = dp[b][k];
    }
```

```
  }
  return dp[a][0];
}

void update(int n, int e, int d, int x, int y, int v, Vi& tree) {
  if (d < x or y < e) return;
  if (x <= e and d <= y) {
    tree[n] += v;
    return;
  }
  int m = (e + d)/2;
  update(2*n, e, m, x, y, v, tree);
  update(2*n + 1, m + 1, d, x, y, v, tree);
}

int read(int n, int e, int d, int p, Vi& tree) {
  if (e == d) return tree[n];
  int m = (e + d)/2;
  int res = tree[n];
  if (p <= m) res += read(2*n, e, m, p, tree);
  else res += read(2*n + 1, m + 1, d, p, tree);
  return res;
}

void update(int c, int x, int y, int v) {
  update(1, 0, N - 1, x, y, v, trees[c]);
}

int read(int c, int p) {
  return read(1, 0, N - 1, p, trees[c]);
}

int main() {
  int tcas;
  scanf("%d", &tcas);
  for (int cas = 1; cas <= tcas; ++cas) {
    int q;
    scanf("%d%d", &N, &q);

    color = Vi(N);
    for (int i = 0; i < N; ++i) {
      scanf("%d", &color[i]);
      --color[i];
    }

    net = Mi(N);
    for (int i = 0; i + 1 < N; ++i) {
      int a, b;
      scanf("%d%d", &a, &b);
      --a; --b;
      net[a].PB(b);
      net[b].PB(a);
    }

    bfs();
    compute_dp();
    compute_et();

    trees = Mi(10);
    for (int i = 0; i < 10; ++i) trees[i] = Vi(4*N + 17, 0);
```

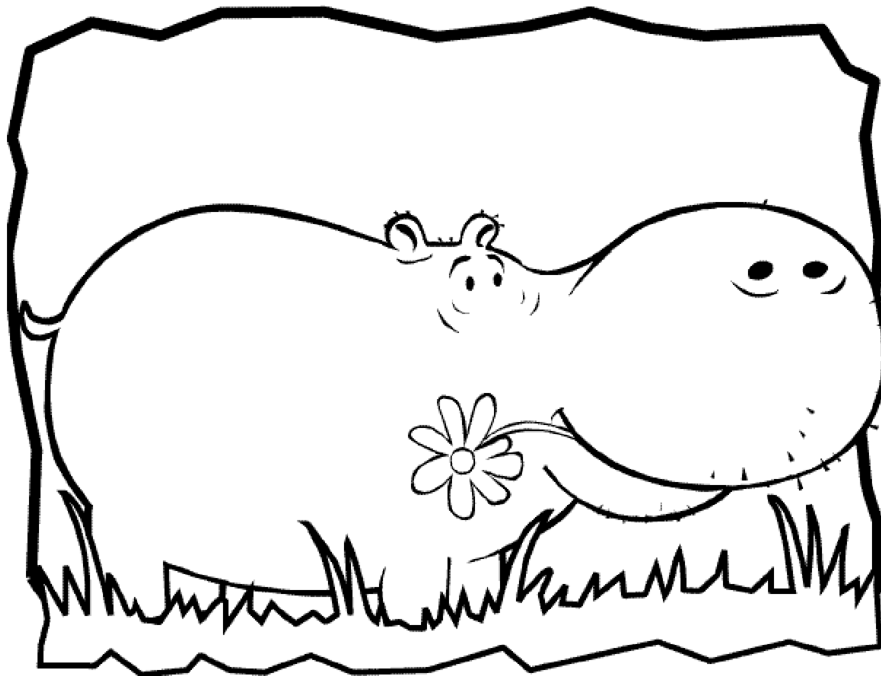
```
    for (int i = 0; i < N; ++i) {
      update(color[i], et1[i], et2[i], 1);
    }

    while (q--) {
      int a, b, c;
      scanf("%d%d%d", &a, &b, &c);
      --b; --c;

      if (a == 0) {
        update(color[b], et1[b], et2[b], -1);
        update(c, et1[b], et2[b], 1);
        color[b] = c;
      }
      else {
        int t = lca(b, c);
        int m = 0;
        for (int i = 0; i < 10; ++i) {
          int tmp = read(i, et1[b]) + read(i, et1[c]) - read(i, et1[t]);
          if (pare[t] != -1) tmp -= read(i, et1[pare[t]]);
          m = max(m, tmp);
        }
        printf("%d\n", m);
      }
    }
  }
}
```

BUET INTER-UNIVERSITY PROGRAMMING CONTEST

PROBLEM B – BEST FRIEND

Problem

I have a friend named Hippo. Hippo is my best friend. Whenever I am happy I call hippo. Whenever I am sad I call hippo. I call hippo even if I am not happy or sad. Hippo means a lot to me. So if hippo is in some sort of trouble and asks me for help I would do anything to solve that problem.



You might find it interesting that hippo is a very good programmer too. Hippo likes to solve difficult and challenging problems. We have been solving problems from the beginning of our friendship. So if anyone is facing some difficulties getting "Accepted" in some problem the other person always tries to help.

Yesterday, hippo gave me a code and asked me if the code is alright. Hippo told me that the code was getting the verdict "Time limit exceeded" again and again. So I asked about the problem. Let me tell you the problem exactly like hippo described:

An integer **N** is given. You need to find out how many numbers **I** $(\mathbf{1} \leq \boldsymbol{I} \leq \boldsymbol{N})$ are there such that $\boldsymbol{GCD}(\boldsymbol{I}, \boldsymbol{N}) \leq \boldsymbol{X}$. Here **GCD** means greatest common divisor.

Now being super busy with your contest, I have no time to help my friend. But I can't let hippo feel helpless with any problem. So can you please help hippo and save our friendship?

Input

The first line of input is an integer **T** ($\mathbf{1} \leq \boldsymbol{T} \leq \mathbf{10}$), which is the number of test cases. Each case starts with two integers **N** ($\mathbf{0} < N \leq \mathbf{10^{12}}$) and **Q** ($\mathbf{0} < Q \leq 50{,}000$). After that **Q** lines follow, each with a value of **X,** which will be a 64 bit signed integer.

Output

For each case print a line "Case C" (without the quotes), where **C** is the case number. After that for each **X** print a line with the value **R**, where there are **R** positive numbers less than or equal to **N** such that their **GCD** is less than or equal to **X**.

Sample Input	Output for Sample Input
2 30 3 1 2 10 11 2 1 2	Case 1 8 16 28 Case 2 10 10

Explanation of 1st Sample Input

For 1, 7, 11, 13, 17, 19, 23, 29 the GCD with 30 is 1.

For 2, 4, 8, 14, 16, 22, 26, 28 the GCD with 30 is 2.

GCD (30, 15) = 15. All the other values have GCD ≤ 10.

Problemsetter: Hasnain Heickal Jami

Special Thanks: Anindya Das

```
/*
UVa 12425. Best Friend
The implemented solution does the following: At each case, before asking any
query, a precalculation is made. For each possible value g that gcd(x,n)
(1<=x<=n) can give, which are as many as the number of divisors n has, the
number of different ways of choosing x such that gcd(x,n)=g is calculated. Once
this precalculation is made, the next structure is generated:
G[i] = i-th value gcd(x,n) can give
W[i] = number of different values x can be (1<=x<=n) such that gcd(x,n)=G[i]
S[i] = W[0] + W[1] + ... + W[i]
Once this is calculated, each query can be asked by doing a binary search over
the array G: Find the biggest y such that G[y] <= x, where x is the value read
from the input. If there is not such a y, then the answer is 0, otherwise the
answer is S[y].
*/
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <cstdio>
using namespace std;

#define X first
#define Y second
#define PB push_back

typedef long long ll;
typedef pair<ll, ll> P;
typedef vector<P> Vp;
typedef vector<ll> Vll;

ll N;
Vp desc;
Vll proh;
Vp vect;

void descomposa(ll n, Vp& v) {
  v.clear();
  for (ll i = 2; i*i <= n; ++i) {
    int q = 0;
    while (n%i == 0) {
      n /= i;
      ++q;
    }
    if (q > 0) v.PB(P(i, q));
  }
  if (n > 1) v.PB(P(n, 1));
}

ll fun(ll m) {
  ll res = 0;
  int n = proh.size();
  for (int i = 0; i < (1<<n); ++i) {
    ll t = 1;
    int s = 0;
    for (int j = 0; t <= m and j < n; ++j) {
      if ((i>>j)&1) {
        t *= proh[j];
        ++s;
      }
    }
```

```
    if (s%2 == 0) res += m/t;
    else res -= m/t;
  }
  return res;
}

void divi(int n, ll t) {
  if (n == int(desc.size())) {
    vect.PB(P(t, fun(N/t)));
    return;
  }

  proh.PB(desc[n].X);
  for (int i = 0; i <= desc[n].Y; ++i) {
    if (i == desc[n].Y) proh.pop_back();
    divi(n + 1, t);
    t *= desc[n].X;
  }
}

int main() {
  int tcas;
  scanf("%d", &tcas);
  for (int cas = 1; cas <= tcas; ++cas) {
    printf("Case %d\n", cas);

    ll n;
    int q;
    scanf("%lld%d", &n, &q);

    N = n;

    descomposa(n, desc);

    vect.clear();
    divi(0, 1);

    sort(vect.begin(), vect.end());

    int sz = vect.size();
    for (int i = 1; i < sz; ++i) vect[i].Y += vect[i - 1].Y;

    for (int i = 0; i < q; ++i) {
      ll x;
      scanf("%lld", &x);

      int e = 0, d = sz - 1, r = -1;
      while (e <= d) {
        int m = (e + d)/2;
        if (vect[m].X <= x) {
          r = m;
          e = m + 1;
        }
        else d = m - 1;
      }

      if (r == -1) printf("0\n");
      else printf("%lld\n", vect[r].Y);
    }
  }
}
```

UVa 12427. Donkey of the Sultan

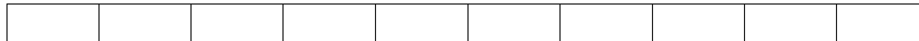
BUET INTER-UNIVERSITY PROGRAMMING CONTEST

PROBLEM D – DONKEY OF THE SULTAN

Problem

Finally our sultan got married. There were varieties of gifts in the wedding- gold, silver, diamond, platinum, elephant, horse, silk clothes etc. Among all gifts one gift was different from all the others. That is donkey. Though its name was donkey but it was very intelligent. Because of this donkey he won many battles without any bloodshed. How? The donkey was expert of game theory. So the sultan took the donkey with him to the country which he wished to conquer and proposed for a game.

The game is played with 1 gold coin, 1 diamond coin and 1 silver coin. The game is played in a long strip (It is really very very long).



First the opponent of sultan will get the opportunity to place the coins in the strip. Leftmost coin should be gold, then diamond and rightmost is silver. But the opponent has to follow some constraints given by the sultan. Sultan says:

1. There should be at least a1 and at most a2 cells between left end point of the strip and gold coin.
2. There should be at least b1 and at most b2 cells between gold and diamond coin.
3. There should be at least c1 and at most c2 cells between diamond and silver coin.

At each move a player may move one of the three coins to the adjacent cell to its left OR move both the gold and silver coins to the adjacent cell to its left. After movement no cell should contain more than one coin. Coins may not be moved beyond the left end of the strip. The game finishes when no player can move.

The sultan came to your country now and wants to play with you. He has given the numbers $\mathbf{a_1}$, $\mathbf{a_2}$, $\mathbf{b_1}$, $\mathbf{b_2}$, $\mathbf{c_1}$ and $\mathbf{c_2}$. You will place the coins in the strip. Then the sultan will give his move, then you will move, then again sultan and so on. You have to calculate the number of position you can start with following the Sultan's constraint so that you can win the game. Assume that both of you will play optimally. Two positions will be different if the placement of any of the coins is different.

Input

In the first line of there will be number of test case **T** ($\mathbf{1 \le T \le 10,000}$). Each of the following T lines contains $\mathbf{a_1}$, $\mathbf{a_2}$, $\mathbf{b_1}$, $\mathbf{b_2}$, $\mathbf{c_1}$ and $\mathbf{c_2}$ ($\mathbf{0 \le a_1, a_2, b_1, b_2, c_1, c_2 \le 100,000}$ ***and*** $\mathbf{a_1 \le a_2, b_1 \le b_2, c_1 \le c_2}$).

Output

For each test case output the case number followed by the number of winning positions. For specific format please follow the sample input output.

Sample Input	Output for Sample Input
2 1 1 0 0 1 1 2 3 1 1 3 4	Case 1: 0 Case 2: 1

Explanation of 1st Sample

The given sample results in only one possible starting position that is:

	G	D		S					

This is a losing position for you, because in the first move the sultan will move both Gold and Silver coins. Then you have no other way but moving Diamond coin. Then sultan will move Silver coin and you don't have any move. So, you lose.

Problemsetter: Md. Mahbubul Hasan

Special Thanks: Tasnim Imran Sunny

```cpp
/*
UVa 12427. Donkey of the Sultan
The state of the game can be described as (a, b, c), where a is the distance
from the left end of the strip to the 1st coin, b is the distance from the 1st
to the 2nd coin and c is the distance from the 2nd to the 3rd coin. The 4
possible moves are:
(-1, +1,  0) moving the 1st coin
( 0, -1, +1) moving the 2nd coin
( 0,  0, -1) moving the 3rd coin
(-1, +1, -1) moving the 1st and the 3rd coins
with the restriction that a, b and c have to be always nonnegative.

After some analysis of the moves we conclude that the states where a is odd
are losing states, and among the states where a is even those with an even
c are losing states and those with an odd c are winning states. Using this
we can easily compute the number of losing states in the cube a < a2, b < b2,
c < c2 for some fixed a2, b2 and c2. Applying the inclusion-exclusion principle
we can compute the number of losing states in the cube a1 < a < a2, b1 < b < b2,
c1 < c < c2 for some fixed a1, a2, b1, b2, c1 and c2. The number of winning
states is the volume of the cube minus the number of losing states.
*/
#include <iostream>

using namespace std;

typedef long long ll;

ll losing(ll a, ll b, ll c) {
  if (a < 0 || b < 0 || c < 0) return 0;
  return (a/2)*b*c + ((a+1)/2)*b*(c/2);
}

ll losing(ll a1, ll a2, ll b1, ll b2, ll c1, ll c2) {
  ll ret = losing(a2+1, b2+1, c2+1);
  ret -= losing(a2+1, b2+1, c1);
  ret -= losing(a2+1, b1, c2+1);
  ret -= losing(a1, b2+1, c2+1);
  ret += losing(a2+1, b1, c1);
  ret += losing(a1, b2+1, c1);
  ret += losing(a1, b1, c2+1);
  ret -= losing(a1, b1, c1);
  return ret;
}

int main() {
  int t;
  cin >> t;
  for (int ca = 1; t--; ++ca) {
    ll a1, a2, b1, b2, c1, c2;
    cin >> a1 >> a2 >> b1 >> b2 >> c1 >> c2;
    cout << "Case " << ca << ": ";
    cout << (a2-a1+1)*(b2-b1+1)*(c2-c1+1) - losing(a1, a2, b1, b2, c1, c2) << endl;
  }
}
```

UVa 12428. Enemy at the Gates

BUET INTER-UNIVERSITY PROGRAMMING CONTEST

PROBLEM E – ENEMY AT THE GATES

Problem

The kingdom of ByteLand is in trouble. The enemies are going to attack ByteLand. The enemies know that ByteLand has exactly **N** cities and exactly **M** bidirectional roads and it is possible to go from any city to every other city directly or via other cities. They also know that any pair of cities can be directly connected by at most one road. But they do not have any information about which road connects which two cities. They are planning to destroy all critical roads of ByteLand. A road is critical if after destroying that road only at least one pair of cities become disconnected. They are very optimistic so they expect to destroy maximum number of critical roads. What is the maximum number of critical roads that can be present in ByteLand according only to the information the enemies have about ByteLand?

Input

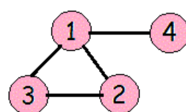
The first line of input contains **T** $(\mathbf{1 \leq T \leq 50})$ which is the number of tests cases. Each case contains two integers **N** which is the number of cities and **M** which is the number of roads $\left(\mathbf{2 \leq N \leq 10^5 \; and \; 1 \leq M \leq \frac{N(N-1)}{2}}\right)$.

Output

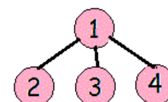
For each test case output one integer the maximum number of critical roads that could be present in ByteLand.

Sample Input	Output for Sample Input
2 4 4 4 3	1 3

Explanation of Sample Cases



For sample 1, one road network with maximum possible 1 critical road(1-4) is shown. There can be more valid networks, but none of them has more than 1 critical road.



For sample 2, one road network where every road is critical is shown.

Problemsetter: Tasnim Imran Sunny

Special Thanks: Shahriar Rouf Nafi

```
/*
UVa 12428. Enemy at the Gates
In order to maximize the number of bridge edges, an optimal construction of the
graph is to make all the bridges to form a single path, and to put the rest of
the edges between nodes such that: (1) do not belong to the path of bridges, or
(2) are one of the ends of the path of bridges. If we have a path of bridges of
length e (e edges, e+1 nodes), then the maximum number of edges we can add to
the graph (which has n nodes) is (n - e)*(n - e - 1)/2. The thing is to find the
maximum e such that: m <= e + (n - e)*(n - e - 1)/2, where m is the number of
edges the graph has to have. Given the constraints in the problem, the
implemented algorithm can iterate over all the possible values of e and get to
run within the allowed execution time.
*/
#include <iostream>
using namespace std;

typedef long long ll;

int main() {
  int tcas;
  cin >> tcas;

  for (int cas = 1; cas <= tcas; ++cas) {
    int n;
    ll m;
    cin >> n >> m;
    int r = 0;
    for (int i = n - 1; i > 0; --i) {
      ll t = n - i;
      if (i + t*(t - 1)/2 >= m) {
        r = i;
        break;
      }
    }
    cout << r << endl;
  }
}
```

UVa 12429. Finding Magic Triplets

BUET INTER-UNIVERSITY PROGRAMMING CONTEST

PROBLEM F – FINDING MAGIC TRIPLETS

Problem

Hermione Granger is very concerned about the magical disabilities of squibs (A Squib is someone who was born into a wizard family but hasn't got any magic powers). Since the fall of Voldemort, she has been working hard to invent a potion to cure these disabilities. After a lot of research work, she has invented that a certain amount of apple juice needs to be mixed with the burnt leaves of birch tree in a lot of cherry juice. Later, she invents that for a **k**-year old person, if **a** amount of apple juice, **b** amount of leaves of birch tree and **c** amount of cherry juice are mixed, it must satisfy the following equation:

$$(a + b^2) \bmod k = c^3 \bmod k, \text{ where } a<=b<=c \text{ and } 1<= a, b, c <= n.$$

She names such a triplet (a, b, c) as a magic triplet for a **k**-year old person. She wants to know how many different magic triplets exist for known values of **n** and **k**. A triplet is different from another if any of the three values is not same in both triplets.

Input

First line of the input contains a single positive integer **T** $(1 \le T \le 400)$ denoting the number of test cases. Then in each of the following **T** lines, there will be two integers **n** and **k** $(1 \le n, k \le 10^5)$.

Output

For each of the cases, output a single line containing "Case x: y", where **x** is the case number and **y** is the number of magic triplets.

Sample Input	Output for Sample Input
1 10 7	Case 1: 27

Problemsetter: Anindya Das

Special Thanks: Kazi Rakibul Hossain and Tasnim Imran Sunny

```
/*
UVa 12429. Finding Magic Triplets
Let M[x][y] be the number of ways to choose a, b, with a<=b and b<=x,
such that a+b^2 (mod k) = y. Then we can compute the answer in this way:
M[1][1^3 (mod k)] + M[2][2^3 (mod k)] + ... + M[n][n^3 (mod k)]
(Ways of choosing a, b, when c=1..n).
We can calculate T[x+1][y] (1<=y<k) using the values of T[x][y] (0<=y<k)
initially T[x+1][y] = T[x][y] for every y (0<=y<k) (1)
for (int a = 1; a <= x+1; ++a) ++T[x+1][a + (x+1)^2 (mod k)]; (2)
(when we "increment" x, the new "triplets" we can make are:
(1,x+1,x+1), (2,x+1,x+1), ..., (x+1,x+1,x+1))
If we simply update the necessary values of T[x] in order to turn it into
T[x+1], then we can avoid (1). Note that in (2) we update consecutive
positions in an array, and this can be done in O(log(k)) if we use an
Interval Tree instead. It's important to remark that a + (x+1)^2 (mod k)
is cyclic with a=1..x+1.
*/
#include <iostream>
#include <cstdio>
using namespace std;

typedef long long ll;

int N, K;
ll tree[400000];

void update(int n, int e, int d, int x, int y, ll v) {
  if (d < x or y < e) return;
  if (x <= e and d <= y) {
    tree[n] += v;
    return;
  }
  int m = (e + d)/2;
  update(2*n, e, m, x, y, v);
  update(2*n + 1, m + 1, d, x, y, v);
}

ll read(int n, int e, int d, int p) {
  if (e == d) return tree[n];
  int m = (e + d)/2;
  if (p <= m) return tree[n] + read(2*n, e, m, p);
  return tree[n] + read(2*n + 1, m + 1, d, p);
}

void update(int x, int y, ll v) {
  return update(1, 0, K - 1, x, y, v);
}

ll read(int p) {
  return read(1, 0, K - 1, p);
}

void fun(ll a, ll b) {
  ll t = (b - a + 1)/K;
  if (t > 0) update(0, K - 1, t);

  ll q = (b - a + 1)%K;
  if (q > 0) {
    a %= K;
    b = (a + q - 1)%K;
    if (a <= b) update(a, b, 1);
```

```
    else {
      update(a, K - 1, 1);
      update(0, b, 1);
    }
  }
}

int main() {
  int tcas;
  scanf("%d", &tcas);
  for (int cas = 1; cas <= tcas; ++cas) {
    scanf("%d%d", &N, &K);
    for (int i = 0; i < 4*K; ++i) tree[i] = 0;

    ll res = 0;
    for (int i = 1; i <= N; ++i) {
      fun(1 + ll(i)*i, i + ll(i)*i);
      res += read((ll(i)*ll(i)*ll(i))%K);
    }
    printf("Case %d: %lld\n", cas, res);
  }
}
```

BUET INTER-UNIVERSITY PROGRAMMING CONTEST

PROBLEM G – GRAND WEDDING

Problem

Finally it's the big day. The sultan is getting married. Kings, Queens and Diplomats around the world are coming to attend the Sultan's weddings. But alas!!! Just before the grand day the royal treasury was raided. And the Sultan is now very upset about the security of his guests. As the Sultan's minister of homeland security your job is to ensure that the whole kingdom is secured and the grand day goes on without any drift. The sultan wants that:

1. Every road in the kingdom is secured by a royal guard.

2. The royal guards are highly trained. When they are standing on a road intersection they can keep an eye on every road meeting at that intersection. And you have to place the guards on the intersection.

3. But guards are human too!! They get bored while standing on the intersections. When they see another fellow guard on the other end of a road leaving his intersection he starts to chat with him and sometimes forget what he was supposed to do!!! The Sultan wants you to place them in such way that this incident never occurs.

4. You have to ensure that conditions (1, 2, and 3) are met, and to do this you may even ask the civil works department to demolish a proper subset of the kingdom roads. The people in the civil works department are very lazy. They are so lazy that they do not want to have a list of roads and go demolish them. They just ask for a road length **K** and demolish all the roads with length greater or equal to K.

Now you have to secure the kingdom by conditions [1, 2, 3 and 4] and also maximize **K** if any road is needed to be demolished.

Input

The first line of input will contain an integer **T** $(\mathbf{1 \le T \le 40})$ which is the number of test cases. Each of the **T** test cases will begin with an integer pair **N** and **M** $(\mathbf{1 \le N \le 50,000} \; \boldsymbol{and} \; \mathbf{1 \le M \le 80,000})$ which are the number of intersections and number of kingdom roads respectively. The intersections are numbered from **1** to **N**. After that, **M** lines will follow. Each line contains 3 integers **a, b, w** $(\mathbf{1 \le a, b \le N} \; \boldsymbol{and} \; \mathbf{1 \le w \le 2^{31} - 1})$ which means there is a road between intersections **a** and **b** of **w** length. A road between intersections **(a, b)** implies a road between intersections **(b, a)** with same length as well.

Output

For each test case you have to print a single integer in a separate line, the maximum **K** if any road is required to be demolished to secure the kingdom by conditions [1, 2, 3 and 4]. Print **0** if no road is required to be demolished. Print **-1** if it is impossible to secure the kingdom by conditions [1, 2, 3 and 4].

Sample Input	Output for Sample Input
1 4 5 1 2 1 3 1 7 2 3 3 2 4 11 3 4 5	7

Problemsetter: Shihabur Rahman Chowdhury

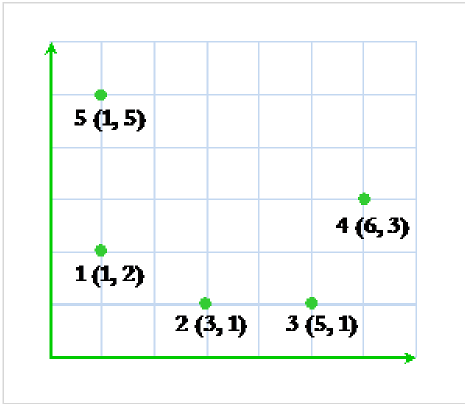
Special Thanks: Md. Mahbubul Hasan

```
/*
UVa 12430. Grand Wedding
The problem asks for the maximum value K such that the resulting graph after
removing all the roads with length greater or equal to K is bicolorable.
We can perform a binary search on the value of K because if for some value of K
the resulting graph is bicolorable, for every value K' < K the resulting graph
will be bicolorable as well; and if for some value of K the resulting graph is
not bicolorable, for every value K' > K the resulting graph will not be
bicolorable either.
*/
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;
typedef pair<int, ll> PII;

const ll INF = 1000000000000000000LL;

bool bicolor(vector<vector<PII> >& g, vector<int>& col, ll k, int nd) {
  for (int i = 0; i < int(g[nd].size()); ++i) if (g[nd][i].second < k) {
    if (col[g[nd][i].first] == col[nd]) return false;
    if (col[g[nd][i].first] == -1) {
      col[g[nd][i].first] = 1-col[nd];
      if (!bicolor(g, col, k, g[nd][i].first)) return false;
    }
  }
  return true;
}

bool bicolorable(vector<vector<PII> >& g, ll k) {
  vector<int> col(g.size(), -1);
  for (int i = 0; i < int(g.size()); ++i) if (col[i] == -1) {
    col[i] = 0;
    if (!bicolor(g, col, k, i)) return false;
  }
  return true;
}

int main() {
  int t;
  cin >> t;
  for (; t--;) {
    int n, m;
    cin >> n >> m;
    vector<vector<PII> > g(n);
    ll mnw = INF, mxw = -INF;
    for (int i = 0; i < m; ++i) {
      ll a, b, w;
      cin >> a >> b >> w;
      --a;
      --b;
      g[a].push_back(PII(b, w));
      g[b].push_back(PII(a, w));
      if (w > mxw) mxw = w;
      if (w < mnw) mnw = w;
    }
    ll lo = mnw, hi = mxw+2;
    for (; hi-lo > 1;) {
      ll md = lo+(hi-lo)/2;
```

```
      if (bicolorable(g, md)) {
        lo = md;
      }
      else {
        hi = md;
      }
    }
    ll ans = lo;
    if (ans > mxw) ans = 0;
    else if (ans <= mnw) ans = -1;
    cout << ans << endl;
  }
}
```

C Consistent Verdicts

In a 2D plane **N** persons are standing and each of them has a gun in his hand. The plane is so big that the persons can be considered as points and their locations are given as Cartesian coordinates. Each of the **N** persons fire the gun in his hand exactly once and no two of them fire at the same or similar time (the sound of two gun shots are never heard at the same time by anyone so no sound is missed due to concurrency). The hearing ability of all these persons is exactly same. That means if one person can hear a sound at distance $\mathbf{R_1}$, so can every other person and if one person cannot hear a sound at distance $\mathbf{R_2}$ the other **N-1** persons cannot hear a sound at distance $\mathbf{R_2}$ as well.



The **N** persons are numbered from **1** to **N**. After all the guns are fired, all of them are asked how many gun shots they have heard (not including their own shot) and they give their verdict. It is not possible for you to determine whether their verdicts are true but it is possible for you to judge if their verdicts are consistent. For example, look at the figure above. There are five persons and their coordinates are (1, 2), (3, 1), (5, 1), (6, 3) and (1, 5) and they are numbered as 1, 2, 3, 4 and 5 respectively. After all five of them have shot their guns, you ask them how many shots each of them have heard. Now if there response is 1, 1, 1, 2 and 1 respectively then you can represent it as (1, 1, 1, 2, 1). But this is an inconsistent verdict because if person 4 hears 2 shots then he must have heard the shot fired by person 2, then obviously person 2 must have heard the shot fired by person 1, 3 and 4 (person 1 and 3 are nearer to person 2 than person 4). But their opinions show that Person 2 says that he has heard only 1 shot. On the other hand (1, 2, 2, 1, 0) is a consistent verdict for this scenario so is (2, 2, 2, 1, 1). In this scenario (5, 5, 5, 4, 4) is not a consistent verdict because a person can hear at most 4 shots.

Given the locations of **N** persons, your job is to find the total number of different consistent verdicts for that scenario. Two verdicts are different if opinion of at least one person is different.

Input

The first line of input will contain **T (≤ 550)** denoting the number of cases.

Each case starts with a line containing a positive integer **N**. Each of the next **N** lines contains two integers $\mathbf{x_i}$ $\mathbf{y_i}$ **(0 ≤** $\mathbf{x_i}$**,** $\mathbf{y_i}$ **≤ 30000)** denoting a co-ordinate of a person. Assume that all the co-ordinates are distinct.

1) For 10 cases, **N = 1000**.
2) For 15 cases, **100 ≤ N < 1000**.
3) For others, **N < 100**.

Output

For each case, print the case number and the total number of different consistent verdicts for the given scenario.

Sample Input	Output for Sample Input
2 3 1 1 2 2 4 4 2 1 1 5 5	Case 1: 4 Case 2: 2

Problem Setter: Shahriar Manzoor, Special Thanks: Jane Alam Jan

```
/*
UVa 12435. Consistent Verdicts
The answer is exactly: #{d(p_i,p_j)|1<=i,j<=N}
where d(p_i,p_j) denotes the distance between i-th point and j-th point.
The algorithm calculates the distance d((x1,y1),(x2,y2))=(x2-x1)^2+(y2-y1)^2
for each possible pair of points and stores this value in a hash table in order
to count the number of different distances. The use of a hash table guarantees
a time complexity of O(N^2), where N is the number of points in the input.
*/
#include <iostream>
#include <cstdio>
using namespace std;

#define X first
#define Y second

typedef pair<int, int> P;

const int MASK = (1<<20) - 1;

int T, nodes;
P first[MASK + 1]; // first_id, update_T
P node[1000000];   // value, next_id

P point[1000];

int main() {
  int tcas;
  scanf("%d", &tcas);
  for (int cas = 1; cas <= tcas; ++cas) {
    int n;
    scanf("%d", &n);
    for (int i = 0; i < n; ++i) scanf("%d%d", &point[i].X, &point[i].Y);

    ++T;
    nodes = 0;

    int res = 1;
    for (int i = 0; i < n; ++i)
      for (int j = i + 1; j < n; ++j) {
        int x = point[i].X - point[j].X;
        int y = point[i].Y - point[j].Y;
        int d = x*x + y*y;

        int h = d&MASK;
        if (first[h].Y == T) {
          bool ok = false;
          for (int k = first[h].X; k != -1; k = node[k].Y)
            if (node[k].X == d) {
              ok = true;
              break;
            }
          if (not ok) {
            node[nodes].X = d;
            node[nodes].Y = first[h].X;
            first[h].X = nodes;
            ++nodes;
            ++res;
          }
        }
        else {
```

```
          node[nodes].X = d;
          node[nodes].Y = -1;
          first[h].X = nodes;
          first[h].Y = T;
          ++nodes;
          ++res;
        }
      }

    printf("Case %d: %d\n", cas, res);
  }
}
```

D Rip Van Winkle's Code

Rip Van Winkle was fed up with everything except programming. One day he found a problem which required to perform three types of update operations **(A, B, C)**, and one query operation **S** over an array **data[]**. Initially all elements of data are equal to 0. Though Rip Van Winkle is going to sleep for 20 years, and his code is also super slow, you need to perform the same update operations and output the result for the query operation **S** in an efficient way.

```
long long data[250001];
void A( int st, int nd ) {
    for( int i = st; i <= nd; i++ ) data[i] = data[i] + (i - st + 1);
}
void B( int st, int nd ) {
    for( int i = st; i <= nd; i++ ) data[i] = data[i] + (nd - i + 1);
}
void C( int st, int nd, int x ) {
    for( int i = st; i <= nd; i++ ) data[i] = x;
}
long long S( int st, int nd ) {
    long long res = 0;
    for( int i = st; i <= nd; i++ ) res += data[i];
    return res;
}
```

Input

The first line of input will contain **T** ($\leq 4*10^5$) denoting the number of operations. Each of the next **T** lines starts with a character **('A', 'B', 'C' or 'S')**, which indicates the type of operation. Character **'A', 'B'** or **'S'** will be followed by two integers, **st** and **nd** in the same line. Character **'C'** is followed by three integers, **st, nd** and **x**. It's assumed that, $1 \leq st \leq nd \leq 250000$ and $-10^5 \leq x \leq 10^5$. The meanings of these integers are explained by the code of Rip Van Winkle.

Output

For each line starting with the character **'S'**, print **S(st, nd)** as defined in the code. Dataset is huge, so use faster I/O methods.

Sample Input	Output for Sample Input
7 A 1 4 B 2 3 S 1 3 C 3 4 -2 S 2 4 B 1 3 S 2 4	9 0 3

Problem Setter: Anindya Das, Special Thanks: Tanaeem M Moosa, Jane Alam Jan

```
/*
UVa 12436. Rip Van Winkle's Code
This problem can be solved with a data structure called Interval Tree.
The leaves of the tree represent the values of the original vector, while
intermediate nodes store changes that affect the value of the leaves of their
respective subtrees.
The operations A, B, C always change the value of an interval. It's enough
to store that change in the minimum number of intermediate nodes so that
all the leaves of the interval are contained in their subtrees. This minimum
number of intermediate nodes is always O(log(N)).
When applying operations A, B, C, store and update the partial results for
operation S in the intermediate nodes or information to get the result in
constant time, so later S can be performed with time complexity of O(log(N)) too.
*/
#include <iostream>
#include <cstdio>
using namespace std;

typedef long long ll;

typedef struct {
  ll h, e1, e2;
  ll s, nz;     // info
} ST;

const int N = 250000;

ST tree[1100000];

inline bool fora(int e, int d, int x, int y) {
  return d < x or y < e;
}

inline bool dins(int e, int d, int x, int y) {
  return x <= e and d <= y;
}

inline ll sum(ll n) {
  return n*(n + 1)/2;
}

inline ll sum(ll a, ll b) {
  return sum(b) - sum(a - 1);
}

void update_info(int n, int e, int d) {
  ll t = d - e + 1;

  tree[n].s = tree[n].h*t + (tree[n].e1 + tree[n].e2)*t*(t - 1)/2;
  if (t > 1) tree[n].s += tree[2*n].s + tree[2*n + 1].s;

  tree[n].nz = 0;
  if (tree[n].h != 0 or tree[n].e1 != 0 or tree[n].e2 != 0) tree[n].nz = 1;
  else if (t > 1) tree[n].nz = min(1LL, tree[2*n].nz + tree[2*n + 1].nz);
}

void update(int n, int e, int d, int x, int y, int h, int e1, int e2) {
  if (fora(e, d, x, y)) return;
  if (dins(e, d, x, y)) {
    tree[n].h += h + ll(e - x)*e1 + ll(y - d)*e2;
    tree[n].e1 += e1;
```

```
    tree[n].e2 += e2;
    update_info(n, e, d);
    return;
  }
  int m = (e + d)/2;
  update(2*n, e, m, x, y, h, e1, e2);
  update(2*n + 1, m + 1, d, x, y, h, e1, e2);
  update_info(n, e, d);
}

ll query(int n, int e, int d, int x, int y) {
  if (fora(e, d, x, y)) return 0;
  if (dins(e, d, x, y)) return tree[n].s;
  int a = max(e, x), b = min(d, y);
  ll res = tree[n].h*(b - a + 1);
  res += tree[n].e1*sum(a - e, b - e);
  res += tree[n].e2*sum(d - b, d - a);
  int m = (e + d)/2;
  res += query(2*n, e, m, x, y);
  res += query(2*n + 1, m + 1, d, x, y);
  return res;
}

void baixa_marron(int n, int e, int d) {
  int m = (e + d)/2;
  update(2*n, e, m, e, d, tree[n].h, tree[n].e1, tree[n].e2);
  update(2*n + 1, m + 1, d, e, d, tree[n].h, tree[n].e1, tree[n].e2);
  tree[n].h = tree[n].e1 = tree[n].e2 = 0;
  update_info(n, e, d);
}

void setzero(int n, int e, int d, int x, int y) {
  if (fora(e, d, x, y)) return;
  if (tree[n].nz == 0) return;
  if (dins(e, d, x, y)) tree[n].h = tree[n].e1 = tree[n].e2 = 0;
  else baixa_marron(n, e, d);
  int m = (e + d)/2;
  setzero(2*n, e, m, x, y);
  setzero(2*n + 1, m + 1, d, x, y);
  update_info(n, e, d);
}

void opA(int a, int b) {
  update(1, 0, N - 1, a, b, 1, 1, 0);
}

void opB(int a, int b) {
  update(1, 0, N - 1, a, b, 1, 0, 1);
}

void opC(int a, int b, ll x) {
  setzero(1, 0, N - 1, a, b);
  update(1, 0, N - 1, a, b, x, 0, 0);
}

ll opS(int a, int b) {
  return query(1, 0, N - 1, a, b);
}

int main() {
  int n;
```

```
  scanf("%d", &n);
  for (int i = 0; i < n; ++i) {
    char op;
    int a, b;
    scanf(" %c %d%d", &op, &a, &b);
    --a; --b;
    if (op == 'A') opA(a, b);
    else if (op == 'B') opB(a, b);
    else if (op == 'C') {
      int x;
      scanf("%d", &x);
      opC(a, b, x);
    }
    else {
      printf("%lld\n", opS(a, b));
    }
  }
}
```

E Kisu Pari Na 2

So, it's learning time!! Suppose a word consisting of only 0 and 1 is called good if there is no adjacent one in the word. We are asked to find good words of length **n**. Now how to solve this problem? Let's get our hand dirty and try to find the number of good words for several values of **n**.

Value of n	Good Words	Number of Good words
1	0, 1	2
2	00, 01, 10	3
3	000, 001, 010, 100, 101	5
4	0000, 0001, 0010, 0100, 0101, 1000, 1001, 1010	8

So, do you find any pattern? The numbers looks like Fibonacci numbers and yes it is Fibonacci number sequence. But the question is why it turned out to be Fibonacci sequence? Suppose we put 0 at the first position. Our constraint is not to put two adjacent ones. So we may fill up the remaining **n-1** places as we like except the condition that there cannot be any adjacent ones. Now if we put 1 at the first position then? Well as we cannot put two adjacent ones, so the second position must be filled by a 0. And following the previous argument rest of the positions can be filled up by as we like except the condition of adjacent ones. So if we denote the number of good words of **n** length is W_n then $W_n = W_{(n-1)} + W_{(n-2)}$ which is of course the recurrence formula of Fibonacci number. And the base cases of this sequence are almost same as Fibonacci number as well.

When I gave this analysis to Mr. Oh he said, "Uh! Come on, this is a very naïve problem, if you are so much intelligent then solve this one: There are **N** islands and **M** bridges. All the bridges are setup between two islands and to pass a bridge you have to give toll of **$1**. The bridges are built in such a way that there is not more than one path among two islands. Now, you have to visit at least **K** different islands. You may choose starting island of your choice, but you have to visit at least **K** different islands in minimum cost. (Starting island is considered to be already visited)"

Input

The first line of input will contain **T (≤ 10)** denoting the number of cases.

Each case starts with two integers **N, M (1 ≤ N ≤ 10000, 0 ≤ M < N)**. Each of the next **M** lines contains two integers **u v (1 ≤ u, v ≤ N, u ≠ v)** meaning that there is a bridge between island **u** and **v**. No bridge will be reported more than once.

The next line contains an integer **q (1 ≤ q ≤ 10000)** denoting the number of queries. Each of the next **q** lines contains one integer **K (1 ≤ K ≤ 10000)**.

Output

For each case, print the case number first. Then for each query print the minimum

amount of toll you need to pay to visit at least **K** different islands. If it is not possible, print **"impossible"**.

Sample Input	Output for Sample Input
2 2 1 1 2 3 1 2 3 5 4 1 2 2 3 2 4 2 5 2 3 2	Case 1: 0 1 impossible Case 2: 2 1

Note

For case 1:

1. For **K = 1**, which ever island we start with, we visit this. So without giving any toll we can visit one island.
2. For **K = 2**, we choose island 1 to start. So we visit island 2 using the only bridge. So it costs $1.
3. For **K = 3**, as there are only 2 islands in total so we cannot visit 3 islands.

Problem Setter: Md. Mahbubul Hasan, Special Thanks: Jane Alam Jan

```
/*
UVa 12437. Kisu Pari Na 2
Given a tree, where 'd' is its diameter, there exists an optimal
path of length 2*(k-1) - min(k-1,d) which visits 'k' different nodes.
In this problem, we are given a forest of trees. For each query,
we will choose the tree of at least the given amount of nodes, with
maximal diameter. If there is no tree with such an amount of nodes,
then the answer is impossible.
*/
#include <cstdio>
#include <vector>
using namespace std;

#define X first
#define Y second
#define PB push_back

typedef long long ll;
typedef pair<int, int> P;
typedef vector<int> Vi;
typedef vector<Vi> Mi;

Mi net;
Vi tree;

P dfs(int n, int m) {
  P res(0, n);
  for (int i = 0; i < int(net[n].size()); ++i) {
    if (net[n][i] == m) continue;
    P tmp = dfs(net[n][i], n);
    ++tmp.X;
    res = max(res, tmp);
  }
  return res;
}

int diameter(int n) {
  return dfs(dfs(n, -1).Y, -1).X;
}

int root(int n) {
  if (tree[n] == -1) return n;
  return tree[n] = root(tree[n]);
}

int main() {
  int tcas;
  scanf("%d", &tcas);
  for (int cas = 1; cas <= tcas; ++cas) {
    int n, m;
    scanf("%d%d", &n, &m);
    net = Mi(n);
    tree = Vi(n, -1);
    Vi comp(n, 1);
    for (int i = 0; i < m; ++i) {
      int a, b;
      scanf("%d%d", &a, &b);
      --a; --b;
      net[a].PB(b);
      net[b].PB(a);
      int ra = root(a), rb = root(b);
```

```
      tree[rb] = ra;
      comp[ra] += comp[rb];
    }

    Vi nodes, diametre;
    for (int i = 0; i < n; ++i)
      if (root(i) == i) {
        nodes.PB(comp[i]);
        diametre.PB(diameter(i));
      }
    int c = nodes.size();

    Vi estalvi(n + 1, -1);
    for (int i = 0; i < c; ++i) estalvi[nodes[i]] = diametre[i];
    for (int i = n - 1; i > 0; --i) estalvi[i] = max(estalvi[i], estalvi[i + 1]);

    int q;
    scanf("%d", &q);

    printf("Case %d:\n", cas);
    while (q--) {
      int k;
      scanf("%d", &k);
      if (k > n or estalvi[k] == -1) printf("impossible\n");
      else printf("%d\n", 2*(k - 1) - min(k - 1, estalvi[k]));
    }
  }
}
```

G February 29

It is 2012, and it's a leap year. So there is a "February 29" in this year, which is called leap day. Interesting thing is the infant who will born in this February 29, will get his/her birthday again in 2016, which is another leap year. So February 29 only exists in leap years. Does leap year comes in every 4 years? Years that are divisible by 4 are leap years, but years that are divisible by 100 are not leap years, unless they are divisible by 400 in which case they are leap years.

In this problem, you will be given two different date. You have to find the number of leap days in between them.

Input

The first line of input will contain **T (≤ 500)** denoting the number of cases.

Each of the test cases will have two lines. First line represents the first date and second line represents the second date. Note that, the second date will not represent a date which arrives earlier than the first date. The dates will be in this format - **"month day, year"**, See sample input for exact format. You are guaranteed that dates will be valid and the year will be in between $\mathbf{2 * 10^3}$ to $\mathbf{2 * 10^9}$. For your convenience, the month list and the number of days per months are given below. You can assume that all the given dates will be a valid date.

Output

For each case, print the case number and the number of leap days in between two given dates (inclusive).

Sample Input	Output for Sample Input
4 January 12, 2012 March 19, 2012 August 12, 2899 August 12, 2901 August 12, 2000 August 12, 2005 February 29, 2004 February 29, 2012	Case 1: 1 Case 2: 0 Case 3: 1 Case 4: 3

Note

The names of the months are {"January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November" and "December"}. And the numbers of days for the months are {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30 and 31} respectively in a non-leap year. In a leap year, number of days for February is 29 days; others are same as shown in previous line.

Problem Setter: Md. Arifuzzaman Arif, Special Thanks: Jane Alam Jan

```
/*
UVa 12439. February 29
This is an easy problem which doesn't need any explanation.
Just be careful with the input format and cover all possible
corner cases.
*/
#include <iostream>
#include <utility>
using namespace std;

#define X first
#define Y second

typedef long long ll;
typedef pair<int, int> P;
typedef pair<int, P> PP;

const string months[12] = { "January", "February", "March", "April", "May", "June",
    "July", "August", "September", "October", "November", "December" };
const P feb29 = P(2, 29);

PP read() {
  string month;
  char spam;
  int day, year;
  cin >> month >> day >> spam >> year;
  PP res(year, P(0, day));
  for (int i = 0; i < 12; ++i)
    if (month == months[i]) {
      res.Y.X = i + 1;
      break;
    }
  return res;
}

// retorna quants multiples de 'm' hi ha entre 'a' i 'b' inclosos
ll multip(ll a, ll b, ll m) {
  return b/m - (a - 1)/m;
}

ll leapyears(int a, int b) {
  if (a > b) return 0;
  ll res = multip(a, b, 4);
  res -= multip(a, b, 100);
  res += multip(a, b, 400);
  return res;
}

bool leap(int y) {
  if (y%400 == 0) return true;
  if (y%100 == 0) return false;
  return y%4 == 0;
}

int main() {
  int tcas;
  cin >> tcas;
  for (int cas = 1; cas <= tcas; ++cas) {
    PP a = read();
    PP b = read();
    cout << "Case " << cas << ": ";
```

```
    if (a.X == b.X) {
      if (leap(a.X) and a.Y <= feb29 and feb29 <= b.Y) cout << 1 << endl;
      else cout << 0 << endl;
    }
    else {
      ll res = leapyears(a.X + 1, b.X - 1);
      if (leap(a.X) and a.Y <= feb29) ++res;
      if (leap(b.X) and feb29 <= b.Y) ++res;
      cout << res << endl;
    }
  }
}
```

H Save the Trees

N trees were planted on a side of a long straight road. In the other side there are farms, industries etc. The market price of these trees is huge now. Some greedy people want to cut these trees down and want to be millionaires. They got the permission from the Govt. decoying that they would develop the area. You published this fact in the web and millions raised their voices against this conspiracy.

You gathered the information of the trees and found the **type** and **height** of all trees. For simplicity, you represented them as integers. You want to find the overall price of the trees. To find the price, the following method is used.

1) The trees are first partitioned into groups with the condition that, types of two trees will not be similar in a group. A group can only be formed using contiguous trees.
2) The price of a group is equal to the height of the tallest tree.
3) The overall price is the summation of prices of all groups.

Now you want to find the minimum possible price of the trees in this scheme and show the Govt. that even though you calculated the minimum possible price, it's actually huge.

Input

The first line of input will contain **T** (≤ 12) denoting the number of cases.

Each case starts with and integer **N** $(1 \leq N \leq 10^5)$. Each of the next **N** lines contains two integers $type_i$ $height_i$ $(1 \leq type_i \leq 10^5, 1 \leq height_i \leq 20000)$ of the ith tree from left to right.

Output

For each case, print the case number and the minimum possible price of the trees according to the scheme described above.

Sample Input	Output for Sample Input
1 5 3 11 2 13 1 12 2 9 3 13	Case 1: 26

Note

Dataset is huge, use faster I/O methods.

Problem Setter: Jane Alam Jan, Special Thanks: Md. Mahbubul Hasan, Md. Arifuzzaman Arif

```
/*
UVa 12440. Save the Trees
One of the first solutions that might come to one's mind is a Dynamic
Programming algorithm with a time complexity of O(N^2) as the follows:
dp[i] is the price of the cheapest way to group the trees in the
interval [i,N]. Each value dp[i] (1<=i<=N) can be computed in O(N)
reusing the previously computed values dp[j] (i<j). Although this
solution is too slow, it may be slightly modified to run in O(N*log(N)).
Let's call last[i] to the greatest value such that the interval
[i, last[i]] doesn't contain two different trees of the same type.
The values last[i] (1<=i<=N) can be easily computed in O(N) (see
function compute_last() below).
When computing dp[i] (1<=n<=N) two cases are possible:
(1) The optimal arrangement consists in having the interval [i,j] (where
  j is the greatest integer such that j<=last[i] and
  height[i+1],height[i+2],...,height[j]<=height[i]) as one group, and an
  optimal arrangement for [j+1,N]. In this case, we pay height[i] for the
  first group, and dp[i+1] for the rest.
(2) It's better to pay more than height[i] for the first group. If we
  decide to pay h for the first group, then the first group will be the
  interval [i,f(h)], where f(h) is the largest integer such that j<=last[i]
  and height[i+1],height[i+2],...,height[j]<=h, and it makes no sense to
  choose a smaller f(h) than this one. So, the optimal value for dp[i] can
  be calculated as dp[i]=min{h+dp[f(h)+1]|height[i]<h<=MAX_HEIGHT}.
  If using and updating at every step an appropriate data structure, the
  optimal value of h can be obtained in amortized time complexity of
  O(log(N)). The idea is to have a priority queue storing the f(h) values,
  sorted by the value h+dp[f(h)+1]. So, in order to find dp[i], we can
  look at the priority queue and take the first one satisfying f(h)<=last[i]
  (dropping all the values not satisfying the restriction (they are not
  useful anymore)). This priority queue must be updated for each i, this can
  be done with an amortized time complexity of O(log(N)).
*/
#include <cstdio>
#include <iostream>
#include <queue>
#include <stack>
#include <utility>
using namespace std;

#define X first
#define Y second

typedef pair<int, int> P;
typedef priority_queue<P> PQ;

const int NMAX = 100000;
const int TMAX = 100000;
const int HMAX = 20000;

int N, type[NMAX], height[NMAX];
int last[NMAX + 1], dp[NMAX + 1];
int dins[NMAX];

int tmp[TMAX + 1];
void compute_last() {
  for (int i = 0; i <= TMAX; ++i) tmp[i] = N;
  last[N] = N - 1;
  for (int i = N - 1; i >= 0; --i) {
    last[i] = min(last[i + 1], tmp[type[i]] - 1);
    tmp[type[i]] = i;
```

```
  }
}

int main() {
  int tcas;
  scanf("%d", &tcas);
  for (int cas = 1; cas <= tcas; ++cas) {
    scanf("%d", &N);
    for (int i = 0; i < N; ++i) scanf("%d%d", &type[i], &height[i]);

    compute_last();

    stack<int> cand;
    PQ cua, alt;

    dp[N] = 0;
    for (int i = N - 1; i >= 0; --i) {
      int h = height[i], j = last[i];
      while (not cand.empty()) {
        int p = cand.top();
        cand.pop();
        if (p > j) break;
        if (dins[p] <= h and height[p] <= h) dins[p] = -1;
        else {
          dins[p] = h;
          cua.push(P(-h - dp[p], p));
          cand.push(p);
          break;
        }
      }

      alt.push(P(h, i));
      while (alt.size() and j < alt.top().Y) alt.pop();
      int hmax = alt.top().X;
      dp[i] = hmax + dp[j + 1];

      while (cua.size()) {
        int x = -cua.top().X;
        int p = cua.top().Y;
        if (j < p or x != dins[p] + dp[p]) cua.pop();
        else break;
      }
      if (cua.size()) dp[i] = min(dp[i], -cua.top().X);

      dins[i] = 0;
      cua.push(P(-dp[i], i));
      cand.push(i);
    }

    printf("Case %d: %d\n", cas, dp[0]);
  }
}
```

I Superb Sequence

There were three friends (Alice, Bob and Carol) who regularly went to expeditions and discovered new mountain peaks. They often proposed different names and it was a problem to decide which name they would choose for the newly discovered peaks. Alice and Bob both said that the name of the peak must be a super sequence of their proposed names **A** and **B**, i.e. **A** and **B** should be **subsequences** of the name of the peak. Carol said that the name of the peak must be a **subsequence** of her proposed name **C**. As they don't like long names, they want to know the number of distinct shortest names which satisfy their needs.

So, given three strings **A, B** and **C**, you have to find the number of distinct shortest common super sequences of **A** and **B** who are also a subsequence of **C**. Moreover, you need to find the lexicographically earliest such sequence. Two sequences are distinct if they differ in at least one position. A **subsequence** is a sequence obtained by deleting zero or more characters from a string. A **super-sequence** is a sequence obtained by inserting zero or more characters in one or more positions of the string.

For example, say, **A = "cdfa"**, **B = "dga"** and **C = "bcdfgaga"**. Then there are two shortest common super sequences of **A** and **B**: **"cdfga"** and **"cdgfa"**, but **"cdgfa"** is not a subsequence of **C**. So the only possible name for the peak is **"cdfga"**.

Input

The first line of input will contain **T (≤ 250)** denoting the number of cases.

Each case contains three lines. First line contains a string denoting **A**, second line contains **B** and third line contains **C**. Assume that the strings are non-empty and length of **A** and **B** will not be more than **100** and length of **C** will not be more than **300**.

Output

For each case, print the case number and the number of distinct possible shortest names for the peak modulo **1000 000 007**. And second line should contain the lexicographically earliest name. If no solution is found then print **"NOT FOUND"** in second line.

Sample Input	Output for Sample Input
2 cdfa dga bcdfgaga abc defm abcdfghm	Case 1: 1 cdfga Case 2: 0 NOT FOUND

Problem Setter: Anindya Das, Special Thanks: Jane Alam Jan

```
/*
UVa 12441. Superb Sequence
We can use Dynamic Programming, defining a state (i, j, k) as the shortest
possible subsequence of c[0..k] that is a supersequence of both a[0..i] and
b[0..j]. This definition of state can be used to:
a) Compute the length of the shortest valid string.
b) Compute the number of shortest valid strings.
c) Compute the lexicographically smallest shortest valid string.

From every state (i, j, k) that has been computed, two other states can be
updated: (i+1, j, k+x) and (i, j+1, k+y). These states are reached by appending
to the string either a[i+1] or b[j+1], as long as the character appears in
c[k+1..(size(c)-1)]. k+x or k+y is the first position in c where the character
(a[i+1] or b[j+1] respectively) appears.

Once the length of the shortest valid string has been computed, a very similar
recursion can be used to compute the number of valid strings and the
lexicographically smallest shortest valid string.
*/
#include <iostream>
#include <string>
#include <vector>

using namespace std;

typedef long long ll;

const int INF = 1001000000;
const ll MOD = 1000000007;

int ca;

vector<vector<vector<int> > > len(101, vector<vector<int> >(101, vector<int>(300,
    -1)));
vector<vector<vector<int> > > ca_len(101, vector<vector<int> >(101, vector<int>(300,
     -1)));
vector<vector<vector<ll> > > cnt(101, vector<vector<ll> >(101, vector<ll>(300, -1)))
    ;
vector<vector<vector<int> > > ca_cnt(101, vector<vector<int> >(101, vector<int>(300,
     -1)));

int get_len(string& a, string& b, string& c, vector<vector<int> >& cnxt, int ia, int
     ib, int ic) {
  if (ic > int(c.size())) {
    return INF;
  }
  if (ia == int(a.size()) && ib == int(b.size())) {
    return 0;
  }
  if (ic == int(c.size())) {
    return INF;
  }
  int& ret = len[ia][ib][ic];
  if (ca_len[ia][ib][ic] == ca) {
    return ret;
  }
  else {
    ca_len[ia][ib][ic] = ca;
  }
  if (ia == int(a.size())) {
    return ret = get_len(a, b, c, cnxt, ia, ib+1, cnxt[ic][int(b[ib]-'a')]+1)+1;
```

```
  }
  if (ib == int(b.size())) {
    return ret = get_len(a, b, c, cnxt, ia+1, ib, cnxt[ic][int(a[ia]-'a')]+1)+1;
  }
  if (a[ia] == b[ib]) {
    return ret = get_len(a, b, c, cnxt, ia+1, ib+1, cnxt[ic][int(a[ia]-'a')]+1)+1;
  }
  return ret = min(get_len(a, b, c, cnxt, ia+1, ib, cnxt[ic][int(a[ia]-'a')]+1),
      get_len(a, b, c, cnxt, ia, ib+1, cnxt[ic][int(b[ib]-'a')]+1))+1;
}

ll get_cnt(string& a, string& b, string& c, vector<vector<int> >& cnxt, int ia, int
    ib, int ic) {
  if (ic > int(c.size())) {
    return 0;
  }
  if (ia == int(a.size()) && ib == int(b.size())) {
    return 1;
  }
  if (ic == int(c.size())) {
    return 0;
  }
  ll& ret = cnt[ia][ib][ic];
  if (ca_cnt[ia][ib][ic] == ca) {
    return ret;
  }
  else {
    ca_cnt[ia][ib][ic] = ca;
  }
  if (ia == int(a.size())) {
    return ret = get_cnt(a, b, c, cnxt, ia, ib+1, cnxt[ic][int(b[ib]-'a')]+1);
  }
  if (ib == int(b.size())) {
    return ret = get_cnt(a, b, c, cnxt, ia+1, ib, cnxt[ic][int(a[ia]-'a')]+1);
  }
  if (a[ia] == b[ib]) {
    return ret = get_cnt(a, b, c, cnxt, ia+1, ib+1, cnxt[ic][int(a[ia]-'a')]+1);
  }
  ret = 0;
  if (get_len(a, b, c, cnxt, ia+1, ib, cnxt[ic][int(a[ia]-'a')]+1) <= get_len(a, b,
      c, cnxt, ia, ib+1, cnxt[ic][int(b[ib]-'a')]+1)) {
    ret += get_cnt(a, b, c, cnxt, ia+1, ib, cnxt[ic][int(a[ia]-'a')]+1);
    ret %= MOD;
  }
  if (get_len(a, b, c, cnxt, ia, ib+1, cnxt[ic][int(b[ib]-'a')]+1) <= get_len(a, b,
      c, cnxt, ia+1, ib, cnxt[ic][int(a[ia]-'a')]+1)) {
    ret += get_cnt(a, b, c, cnxt, ia, ib+1, cnxt[ic][int(b[ib]-'a')]+1);
    ret %= MOD;
  }
  return ret;
}

void get_string(string& a, string& b, string& c, vector<vector<int> >& cnxt, int ia,
     int ib, int ic, string& res) {
  if (ic > int(c.size())) {
    return;
  }
  if (ia == int(a.size()) && ib == int(b.size())) {
    return;
  }
  if (ic == int(c.size())) {
```

```
    return;
  }
  if (ia == int(a.size())) {
    res += b[ib];
    get_string(a, b, c, cnxt, ia, ib+1, cnxt[ic][int(b[ib]-'a')]+1, res);
    return;
  }
  if (ib == int(b.size())) {
    res += a[ia];
    get_string(a, b, c, cnxt, ia+1, ib, cnxt[ic][int(a[ia]-'a')]+1, res);
    return;
  }
  if (a[ia] == b[ib]) {
    res += a[ia];
    get_string(a, b, c, cnxt, ia+1, ib+1, cnxt[ic][int(a[ia]-'a')]+1, res);
    return;
  }
  if (get_len(a, b, c, cnxt, ia+1, ib, cnxt[ic][int(a[ia]-'a')]+1) < get_len(a, b, c
      , cnxt, ia, ib+1, cnxt[ic][int(b[ib]-'a')]+1)) {
    res += a[ia];
    get_string(a, b, c, cnxt, ia+1, ib, cnxt[ic][int(a[ia]-'a')]+1, res);
  }
  else if (get_len(a, b, c, cnxt, ia, ib+1, cnxt[ic][int(b[ib]-'a')]+1) < get_len(a,
       b, c, cnxt, ia+1, ib, cnxt[ic][int(a[ia]-'a')]+1)) {
    res += b[ib];
    get_string(a, b, c, cnxt, ia, ib+1, cnxt[ic][int(b[ib]-'a')]+1, res);
  }
  else {
    if (a[ia] < b[ib]) {
      res += a[ia];
      get_string(a, b, c, cnxt, ia+1, ib, cnxt[ic][int(a[ia]-'a')]+1, res);
    }
    else {
      res += b[ib];
      get_string(a, b, c, cnxt, ia, ib+1, cnxt[ic][int(b[ib]-'a')]+1, res);
    }
  }
}

int main() {
  int t;
  cin >> t;
  for (ca = 1; t--; ++ca) {
    string a, b, c;
    cin >> a >> b >> c;
    vector<vector<int> > cnxt(c.size(), vector<int>(int('z'-'a')+1, int(c.size())));
    cnxt[int(c.size())-1][int(c[int(c.size())-1]-'a')] = int(c.size())-1;
    for (int i = int(c.size())-2; i >= 0; --i) {
      for (int j = 0; j < int(cnxt[i].size()); ++j) {
        cnxt[i][j] = cnxt[i+1][j];
      }
      cnxt[i][int(c[i]-'a')] = i;
    }
    cout << "Case " << ca << ": ";
    if (get_len(a, b, c, cnxt, 0, 0, 0) >= INF) cout << 0 << endl << "NOT FOUND" <<
       endl;
    else {
      string res = "";
      get_string(a, b, c, cnxt, 0, 0, 0, res);
      cout << get_cnt(a, b, c, cnxt, 0, 0, 0) << endl << res << endl;
    }
```

```
    }
}
```

J Forwarding Emails

"... so forward this to ten other people, to prove that you believe the emperor has new clothes."

Aren't those sorts of emails annoying?

Martians get those sorts of emails too, but they have an innovative way of dealing with them. Instead of just forwarding them willy-nilly, or not at all, they each pick one other person they know to email those things to every time - exactly one, no less, no more (and never themselves). Now, the Martian clan chieftain wants to get an email to start going around, but he stubbornly only wants to send one email. Being the chieftain, he managed to find out who forwards emails to whom, and he wants to know: which Martian should he send it to maximize the number of Martians that see it?

Input

The first line of input will contain **T (≤ 20)** denoting the number of cases.

Each case starts with a line containing an integer **N (2 ≤ N ≤ 50000)** denoting the number of Martians in the community. Each of the next **N** lines contains two integers: **u v (1 ≤ u, v ≤ N, u ≠ v)** meaning that Martian **u** forwards email to Martian **v**.

Output

For each case, print the case number and an integer **m**, where **m** is the Martian that the chieftain should send the initial email to. If there is more than one correct answer, output the smallest number.

Sample Input	Output for Sample Input
3 3 1 2 2 3 3 1 4 1 2 2 1 4 3 3 2 5 1 2 2 1 5 3 3 4 4 5	Case 1: 1 Case 2: 4 Case 3: 3

Problem Setter: Wheeler, Zachary J, Special Thanks: Jane Alam Jan

```cpp
/*
UVa 12442. Forwarding Emails
The e-mail will go through a sequence of nodes and eventually repeat some node
and start moving in a loop. We can compute recursively the number of different
nodes that will be reached from each node as:
r(v) = 1+r(node to which v forwards the e-mail)
The base case is if the node currently being computed v is in a cycle, and then:
r(v) = length of the cycle
*/
#include <iostream>
#include <vector>

using namespace std;

int path_length(vector<int>& v, vector<int>& len, int nd) {
  if (len[nd] >= 0) return len[nd];
  if (len[nd] == -2) {
    return len[nd] = 0;
  }
  len[nd] = -2;
  int rec = path_length(v, len, v[nd]);
  if (len[nd] == 0) {
    for (int i = v[nd]; i != nd; i = v[i]) {
      len[i] = rec+1;
    }
  }
  return len[nd] = rec+1;
}

int main() {
  int t;
  cin >> t;
  for (int ca = 1; t--; ++ca) {
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; ++i) {
      int u;
      cin >> u;
      --u;
      cin >> v[u];
      --v[u];
    }
    vector<int> len(n, -1);
    int best = 0;
    for (int i = 0; i < n; ++i) {
      if (path_length(v, len, i) > path_length(v, len, best)) {
        best = i;
      }
    }
    cout << "Case " << ca << ": " << best+1 << endl;
  }
}
```

3 ACM-ICPC Live Archive

L'ACM-ICPC Live Archive és un jutge online que conté els problemes de les competicions de l'ICPC celebrades des de 1988, tant els de la fase final com de les regionals. El funcionament és pràcticament el mateix que el de l'UVa Online Judge, perquè fa servir la mateixa plataforma i també és gestionat per la Universidad de Valladolid. La diferència principal és que en el Live Archive no s'hi realitza cap competició per practicar.

A continuació presentem els enunciats i solucions dels 24 problemes que vam resoldre de l'ACM-ICPC Live Archive.

ACM-ICPC Live Archive 5780. The Agency

Problem A: The Agency

Following in the footsteps of a number of flight searching startups you want to create the first interplanetary travel website. Your first problem is to quickly find the cheapest way to travel between two planets. You have an advantage over your competitors because you have realized that all the planets and the flights between them have a special structure. Each planet is represented by a string of N bits and there is a flight between two planets if their N-bit strings differ in exactly one position.

The cost of a flight is the cost of landing on the destination planet. If the ith character in a planet's string is a 1 then the ith tax must be paid to land. The cost of landing on a planet is the sum of the applicable taxes.

Given the starting planet, ending planet, and cost of the ith tax compute the cheapest set of flights to get from the starting planet to the ending planet.

Input

Input for each test case will consist of two lines. The first line will have N ($1 \leq N \leq 1{,}000$), the number of bits representing a planet; S, a string of N zeroes and ones representing the starting planet; and E, a string representing the ending planet in the same format. The second line will contain N integers the ith of which is the cost of the ith tax. All costs will be between 1 and 1,000,000. The input will be terminated by a line with a single 0.

Output

For each test case output one number, the minimum cost to get from the starting planet to the ending planet, using the format given below.

Sample Input

```
3 110 011
3 1 2
5 00000 11111
1 2 3 4 5
4 1111 1000
100 1 1 1
30 000000000000000000000000000000 111111111111111111111111111111
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
0
```

Sample Output

```
Case 1: 4
Case 2: 35
Case 3: 106
Case 4: 4960
```

```
/*
ACM-ICPC Live Archive 5780. The Agency
An optimal way to turn the first string into the second is as follows:
Starting from the most expensive bit and decreasingly in price, change
the bits that are intially set to 1 but need to be turned into 0 and also
those bits set to 1 that need to remain at 1 whose price is over some
limit X. After that, starting from the cheapest bit and increasingly in
price, change the bits that are initially 0 but need to be turned into 1
and also those bits that were changed to 0 in the first phase but need
to be 1 at the end.
Ok, but what is the value of X? Just run the algorithm for every possible
value of X (well, you'll notice that it's possible to just assign to X as
many values as different prices of tax).
*/
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>
using namespace std;

#define X first
#define Y second

typedef long long ll;
typedef pair<int, int> P;
typedef vector<P> Vp;
typedef vector<int> Vi;

const ll INF = 4000000000000000000LL;

int main() {
  int cas = 1;

  int n;
  while (cin >> n and n > 0) {
    string a, b;
    cin >> a >> b;

    Vi cost(n);
    for (int i = 0; i < n; ++i) cin >> cost[i];

    Vp v(n);
    for (int i = 0; i < n; ++i) v[i] = P(cost[i], i);
    sort(v.begin(), v.end());

    ll res = INF;
    for (int k = 0; k <= n; ++k) {
      int x = (k == n ? 0 : cost[k]);

      ll s = 0;
      for (int i = 0; i < n; ++i)
        if (a[i] == '1') s += cost[i];

      ll tmp = 0;
      for (int i = n - 1; i >= 0; --i) {
        if ((a[v[i].Y] == '1' and b[v[i].Y] == '0') or (a[v[i].Y] == '1' and b[v[i].
            Y] == '1' and v[i].X > x)) {
          s -= v[i].X;
          tmp += s;
        }
      }
```

```
      for (int i = 0; i < n; ++i) {
        if ((a[v[i].Y] == '0' and b[v[i].Y] == '1') or (a[v[i].Y] == '1' and b[v[i].
            Y] == '1' and v[i].X > x)) {
          s += v[i].X;
          tmp += s;
        }
      }
      res = min(res, tmp);
    }
    cout << "Case " << cas++ << ": " << res << endl;
  }
}
```

ACM-ICPC Live Archive 5782. Condorcet Winners

Problem C: Condorcet Winners

A *Condorcet winner* of an election is a candidate who would beat any of the other candidates in a one-on-one contest. Determining a Condorcet winner can only be done when voters specify a ballot listing all of the candidates in their order of preference (we will call such a ballot a *preference list*). For example, assume we have 3 candidates — A, B and C — and three voters whose preference lists are ABC, BAC, CBA. Here, B is the Condorcet winner, beating A in 2 of the three ballots (ballots 2 and 3) and beating C in 2 of the three ballots (1 and 2).

The *Condorcet voting system* looks for the Condorcet winner and declares that person the winner of the election. Note that if we were only considering first place votes in the example above (as we do in most elections in the US and Canada), then there would be a tie for first place. There can be at most only one Condorcet winner, but there is one small drawback in the Condorcet system — it is possible that there may be no Condorcet winner.

Input

Input for each test case will start with a single line containing two positive integers b c, where b indicates the number of ballots and c indicates the number of candidates. Candidates are considered numbered $0, \ldots, c-1$. Following this first line will be b lines containing c values each. Each of these lines represents one ballot and will contain the values $0, \ldots, c-1$ in some permuted order. Values of b and c will lie in the ranges $1 \ldots 500$ and $1 \ldots 2500$, respectively, and the line 0 0 will follow the last test case.

Output

For each test case output a single line containing either the candidate number of the Condorcet winner, or the phrase `No Condorcet winner` using the format given.

Sample Input

```
3 3
0 1 2
1 0 2
2 1 0
3 3
0 1 2
1 2 0
2 0 1
0 0
```

Sample Output

```
Case 1: 1
Case 2: No Condorcet winner
```

```
/*
ACM-ICPC Live Archive 5782. Condorcet Winners
Note that a Condorcet winner can not lose to any other candidate in a
one-on-one, therefore if we compare two candidates in a one-on-one, the one that
loses can be discarded. Using this we can get a single potential Condorcet
winner by computing the result of n-1 one-on-ones. After that, check if the
potential Condorcet winner is indeed a Condorcet winner by verifying that he
beats each of the other n-1 candidates in a one-on-one.

Computing the winner of a one-on-one can be done efficiently by storing the
position of each candidate in each ballot, and then looping through all ballots,
comparing the respective positions of the dueling candidates in the ballot.
*/
#include <iostream>
#include <vector>

using namespace std;

bool beats(const vector<vector<int> >& pos, int a, int b) {
  int adv = 0;
  for (int i = 0; i < int(pos.size()); ++i) {
    if (pos[i][a] < pos[i][b]) ++adv;
    else --adv;
  }
  return adv > 0;
}

int main() {
  for (int b, c, ca = 1; cin >> b >> c && (b != 0 || c != 0); ++ca) {
    vector<vector<int> > pos(b, vector<int>(c, -1));
    for (int i = 0; i < b; ++i) {
      for (int j = 0; j < c; ++j) {
        int cand;
        cin >> cand;
        pos[i][cand] = j;
      }
    }
    int candidate = 0;
    for (int i = 1; i < c; ++i) {
      if (beats(pos, i, candidate)) {
        candidate = i;
      }
    }
    for (int i = 0; i < c && candidate != -1; ++i) if (i != candidate) {
      if (!beats(pos, candidate, i)) {
        candidate = -1;
      }
    }
    cout << "Case " << ca << ": ";
    if (candidate == -1) cout << "No Condorcet winner" << endl;
    else cout << candidate << endl;
  }
}
```

Problem E: The Banzhaf Buzz-Off

A young researcher named George Lurdan has just been promoted to the board of trustees for Amalgamated Artichokes. Each member of the board has a specified number of votes (or *w*eights) assigned to him or her which can be used when voting on various issues that come before the board. Needless to say, the higher your weight, the more power you have on the board, but exactly how much power you have is a difficult question. It depends not only on the distribution of the weights, but also on what percentage of the votes are needed to pass any resolution (known as the *q*uota). For example, the current board has 5 members whose weights are 20, 11, 10, 8 and 1, where George has the 1. Whenever a simple majority of votes are needed (i.e., when the quota is 26) George has very little power. But when the vote has to be unanimous (quota = 50), George has just as much power as anyone else. George would like to know how much power he has depending on what the quota is; he figures that if his power is zero, then there's no point in going to the meetings—he can buzz off and spend more time on artichoke research.

After doing some reading, George discovered the Banzhaf Power Index (BPI). The BPI measures how often each board member is a critical voter in a winning coalition. A winning coalition is a group of board members whose total weights are greater than or equal to the quota (i.e., they can pass a resolution). A *c*ritical voter in a winning coalition is any member whose departure results in a non-winning coalition. For example, if the quota is 26, then one possible winning coalition would consist of 20, 10 and 1 (George). Here, each of the first two members of the coalition are critical (since if they leave the resulting coalition has only 11 or 21 votes), while George is not critical. In the winning coalition 20, 11, 10, 8, 1, no one is critical when the quota is 26, but everyone is when the quota is 50. The BPI for any member is just the total number of winning coalitions in which that member is critical (NOTE: in reality, the BPI is actually double this figure, but that's not important for us here). For example, when the quota is 26, the BPIs for the members turn out to be 12, 4, 4, 4 and 0, i.e., the board member with weight 20 is a critical voter in 12 different winning coalitions, the board member with weight 11 is a critical voter in 4 different winning coalitions, etc. As expected, George has no power in this case. If the quota is raised to 42, then the BPIs are 3, 3, 3, 1 and 1. In this case George has just as much power as the member with 8 votes!

Since the number of members on the board can vary from 1 to 60 , and board members' weights can change over time, George would like a general program to determine his BPI power.

Input

Input for each test case will consist of two lines: the first line will contain two integers n, q, where n indicates the number of distinct weight values ($1 \leq n \leq 60$) and q is the quota. The second line will contain n pairs of positive integers $w_1\ m_1\ w_2\ m_2\ \cdots\ w_n\ m_n$ where w_i is a weight value and m_i is the number of board members with that weight. The total number of votes $V = w_1m_1 + w_2m_2 + \cdots + w_nm_n$ will be in the range $1 \leq V \leq 60$. The quota will satisfy the condition $V/2 < q \leq V$, and $w_i \neq w_j$ when $i \neq j$. A line containing "`0 0`" will signal the end of input.

Output

For each test case, output a single line of the form:

`Case` n: $b_1\ b_2\ \ldots\ b_n$

where b_i is the Banzhaf Power Index for any member with weight w_i. Separate the BPIs with a single blank.

Sample Input

```
5 26
20 1 11 1 10 1 8 1 1 1
5 42
20 1 11 1 10 1 8 1 1 1
5 50
20 1 11 1 10 1 8 1 1 1
1 31
1 60
0 0
```

Sample Output

```
Case 1: 12 4 4 4 0
Case 2: 3 3 3 1 1
Case 3: 1 1 1 1 1
Case 4: 59132290782430712
```

```
/*
ACM-ICPC Live Archive 5784. The Banzhaf Buzz-Off
For each weight W, count how many different coalitions can be made
when a member of that weight is missing, such that the power of the
coalition is between Q - W and Q - 1, where Q is the quota needed.
In other words, count how many different coalitions can be made
when a member of that weiht is missing such that can not reach the
quota, but they would if the missing member joined the coalition.
This can be done efficiently with DP.
*/
#include <iostream>
using namespace std;

typedef long long ll;

int N, Q;
int W[100], M[100];

ll cn[62][62];

ll dp[61][61][61];
int tdp[61][61][61], T;
ll fun(int n, int a, int b) {
  if (n == N) {
    if (a == 0) return 1;
    return 0;
  }

  if (tdp[n][a][b] == T) return dp[n][a][b];
  tdp[n][a][b] = T;

  ll res = 0;
  for (int i = 0; i <= M[n]; ++i) {
    int t = i*W[n];
    if (t <= b) {
      res += cn[M[n]][i]*fun(n + 1, max(0, a - t), b - t);
    }
  }

  dp[n][a][b] = res;
  return res;
}

int main() {
  for (int n = 0; n < 62; ++n) {
    cn[n][0] = cn[n][n] = 1;
    for (int k = 1; k < n; ++k)
      cn[n][k] = cn[n - 1][k] + cn[n - 1][k - 1];
  }

  int cas = 1;

  while (cin >> N >> Q and N > 0) {
    for (int i = 0; i < N; ++i) cin >> W[i] >> M[i];

    cout << "Case " << cas++ << ":";
    for (int i = 0; i < N; ++i) {
      --M[i];
      ++T;
      cout << " " << fun(0, Q - W[i], Q - 1);
      ++M[i];
```

```
    }
    cout << endl;
  }
}
```

ACM-ICPC Live Archive 5785. GPS I Love You

Problem F: GPS I Love You

Thomas T. Garmin got a GPS for his birthday last year, and he loved it! Unfortunately, sometimes Tom wanted to take a scenic route rather than the shortest one as suggested by the GPS. He did a little reading in the manual and found that he could override the default path-finding algorithm by specifying roads which the GPS system would be forced to use when determining a route. After some experimentation, Tom discovered that often, forcing a single road sufficed to get the desired route. However, for some windier routes, Tom needed to force more roads. Eventually Tom began to worry that he wasted too much time picking roads to force before each trip. Now, instead of enjoying the wonders of his GPS, he spends his drives agonizing over the following question: could he have gotten his GPS to pick the scenic route using fewer forced roads?

Can you save this love affair, or are Tom and his GPS doomed to walk separate paths?

Input

Input for each test case will consist of a number of lines. The first line will contain a single integer $n < 100$ indicating the number of endpoints for the roads, numbered 0 to $n-1$. There will then follow n lines each containing n non-negative integers. If the jth value in row i is positive, it indicates the length of a road from endpoint i to endpoint j; if the value is 0, it indicates that there is no road between those two endpoints. Following these lines will be a line of the form $m\ p_1\ p_2\ p_3\ \ldots\ p_m$ specifying the scenic route that Tom wants – the route contains $m-1$ roads and goes between endpoints p_1 and p_m, visiting endpoints p_2, p_3, etc., in that order. The last test case will be followed by a line containing 0.

Note that when Tom specifies his forced roads to his GPS, he specifies both their direction and order. All the routes are simple paths and all roads lengths are ≤ 100.

Output

For each test case one line of output as follows:

`Case` n: k

where k is the smallest number of roads Tom must force such that the GPS will choose the specified route. You should assume that if there are multiple shortest paths, the GPS always selects the most scenic of these. Therefore, if Tom's route is among the shortest to use a given set of forced roads, it will be picked by the GPS.

Sample Input

```
4
0 4 0 2
4 0 2 0
0 2 0 2
2 0 2 0
4 0 3 2 1
4
0 4 0 1
4 0 1 0
0 1 0 1
1 0 1 0
4 0 3 2 1
0
```

Sample Output

```
Case 1: 1
```

```
/*
ACM-ICPC Live Archive 5785. GPS I Love You
Between two points of the specified route, the GPS will
follow that path iff the distance obtained by following the
route is minimal. First, run the Foyd-Warshall algorithm to
know the minimal distance between all pairs of nodes. Then,
determine with a DP the minimal amount of roads in the route
that need to be fixed.
*/
#include <iostream>
using namespace std;

typedef long long ll;

const ll INF = 4000000000000000000LL;

int N;
ll dist[110][110];

int M;
int path[110];
ll suma[110];

ll dp[110];

int main() {
  int cas = 1;
  while (cin >> N and N > 0) {
    for (int i = 0; i < N; ++i)
      for (int j = 0; j < N; ++j) {
        cin >> dist[i][j];
        if (dist[i][j] == 0) dist[i][j] = INF;
      }

    cin >> M;
    for (int i = 0; i < M; ++i) cin >> path[i];

    suma[0] = 0;
    for (int i = 1; i < M; ++i) suma[i] = suma[i - 1] + dist[path[i - 1]][path[i]];

    for (int i = 0; i < N; ++i) dist[i][i] = 0;
    for (int k = 0; k < N; ++k)
      for (int i = 0; i < N; ++i)
        for (int j = 0; j < N; ++j)
          dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);

    dp[M - 1] = 0;
    for (int i = M - 2; i >= 0; --i) {
      dp[i] = 1 + dp[i + 1];
      for (int j = i + 1; j < M; ++j)
        if (dist[path[i]][path[j]] == suma[j] - suma[i])
          dp[i] = min(dp[i], (j == M - 1 ? 0 : 1 + dp[j + 1]));
    }
    cout << "Case " << cas++ << ": " << dp[0] << endl;
  }
}
```

Problem I: Wally World

Two star-crossed lovers want to meet. The two lovers are standing at distinct points in the plane (but then again, aren't we all?). They can travel freely except that there is a single wall which cannot be crossed. The wall is a line segment which is parallel to either the x or y axis. Each lover can move 1 unit in 1 second. How long will it take them to be together if they both choose the best path?

Input

Input for each test case will consist of two lines each containing four integers. The first two integers will specify the x and y coordinates of the first lover; the next two integers will specify the x and y coordinates of the second lover. The next four integers will specify the start and end points of the wall. Furthermore, in all cases both lovers will not be on the (infinite) line containing the wall — that is, the wall extended in both directions. All coordinates will be positive and less than or equal to 10000 and neither lover will start on the wall. The input will be terminated by a line containing four zeroes.

Output

For each test case, output the minimum time in seconds for the two lovers to meet. Print the answer to exactly 3 decimal places, using the output format shown in the example.

Sample Input

```
5 2 7 2
1 1 1 100
1 2 3 2
2 1 2 100
0 0 0 0
```

Sample Output

```
Case 1: 1.000
Case 2: 1.414
```

```
/*
ACM-ICPC Live Archive 5788. Wally World
The shortest path between the two points A and B is the euclidean distance
between them if the wall does intersect the straight segment AB. Otherwise, the
shortest distance of the shortest path between A and B is
min(dist(A, W1)+dist(W1, B), dist(A, W2)+dist(W2, B))
where dist(X, Y) is the euclidean distance between X and Y, and W1, W2 are the
endpoints of the wall.
*/
#include <cmath>
#include <iostream>

#define X first
#define Y second

using namespace std;

typedef pair<int, int> PDD;

double dist(PDD a, PDD b) {
  return sqrt((a.X-b.X)*(a.X-b.X) + (a.Y-b.Y)*(a.Y-b.Y));
}

double cross_prod_z(PDD u, PDD v) {
  return u.X*v.Y - v.X*u.Y;
}

int sign(double d) {
  if (d > 0.0) return 1;
  if (d < 0.0) return -1;
  return 0;
}

bool opposite_sides(PDD a, PDD b, PDD s1, PDD s2) {
  PDD as1(s1.X-a.X, s1.Y-a.Y);
  PDD as2(s2.X-a.X, s2.Y-a.Y);
  PDD bs1(s1.X-b.X, s1.Y-b.Y);
  PDD bs2(s2.X-b.X, s2.Y-b.Y);
  return sign(cross_prod_z(as1, as2))*sign(cross_prod_z(bs1, bs2)) == -1;
}

bool segments_intersect(PDD a1, PDD a2, PDD b1, PDD b2) {
  return opposite_sides(a1, a2, b1, b2) && opposite_sides(b1, b2, a1, a2);
}

int main() {
  cout.setf(ios::fixed);
  cout.precision(3);
  PDD a, b, w1, w2;
  for (int ca = 1; cin >> a.X >> a.Y >> b.X >> b.Y && (a.X != 0.0 || a.Y != 0.0 || b
      .X != 0.0 || b.Y != 0.0); ++ca) {
    cin >> w1.X >> w1.Y >> w2.X >> w2.Y;
    cout << "Case " << ca << ": ";
    if (!segments_intersect(a, b, w1, w2)) {
      cout << dist(a, b)/2.0 << endl;
    }
    else {
      cout << min(dist(a, w1)+dist(w1, b), dist(a, w2)+dist(w2, b))/2.0 << endl;
    }
  }
}
```

ACM International Collegiate Programming Contest
Asia Regional Contest, Fukuoka, 2011–11–13

Problem A
Gift from the Goddess of Programming

Input: Standard Input
Time Limit: 30 seconds

The goddess of programming is reviewing a thick logbook, which is a yearly record of visitors to her holy altar of programming. The logbook also records her visits at the altar.

The altar attracts programmers from all over the world because one visitor is chosen every year and endowed with a gift of miracle programming power by the goddess. The endowed programmer is chosen from those programmers who spent the longest time at the altar *during the goddess's presence*. There have been enthusiastic visitors who spent very long time at the altar but failed to receive the gift because the goddess was absent during their visits.

Now, your mission is to write a program that finds how long the programmer to be endowed stayed at the altar during the goddess's presence.

Input

The input is a sequence of datasets. The number of datasets is less than 100. Each dataset is formatted as follows.

n
M_1/D_1 h_1:m_1 e_1 p_1
M_2/D_2 h_2:m_2 e_2 p_2
$\vdots$
M_n/D_n h_n:m_n e_n p_n

The first line of a dataset contains a positive even integer, $n \leq 1000$, which denotes the number of lines of the logbook. This line is followed by n lines of space-separated data, where M_i/D_i identifies the month and the day of the visit, $h_i : m_i$ represents the time of either the entrance to or exit from the altar, e_i is either `I` for entrance, or `O` for exit, and p_i identifies the visitor.

All the lines in the logbook are formatted in a fixed-column format. Both the month and the day in the month are represented by two digits. Therefore April 1 is represented by `04/01` and not by `4/1`. The time is described in the 24-hour system, taking two digits for the hour, followed by a colon and two digits for minutes, `09:13` for instance and not like `9:13`. A programmer is identified by an ID, a unique number using three digits. The same format is used to indicate entrance and exit of the goddess, whose ID is `000`.

All the lines in the logbook are sorted in ascending order with respect to date and time. Because the altar is closed at midnight, the altar is emptied at `00:00`. You may assume that each time in the input is between `00:01` and `23:59`, inclusive.

A programmer may leave the altar just after entering it. In this case, the entrance and exit time are the same and the length of such a visit is considered 0 minute. You may assume for such entrance and exit records, the line that corresponds to the entrance appears earlier in the input than the line that corresponds to the exit. You may assume that at least one programmer appears in the logbook.

The end of the input is indicated by a line containing a single zero.

Output

For each dataset, output the total sum of the *blessed time* of the endowed programmer. The blessed time of a programmer is the length of his/her stay at the altar during the presence of the goddess. The endowed programmer is the one whose total blessed time is the longest among all the programmers. The output should be represented in minutes. Note that the goddess of programming is not a programmer.

Sample Input

```
14
04/21 09:00 I 000
04/21 09:00 I 001
04/21 09:15 I 002
04/21 09:30 O 001
04/21 09:45 O 000
04/21 10:00 O 002
04/28 09:00 I 003
04/28 09:15 I 000
04/28 09:30 I 004
04/28 09:45 O 004
04/28 10:00 O 000
04/28 10:15 O 003
04/29 20:00 I 002
04/29 21:30 O 002
20
06/01 09:00 I 001
06/01 09:15 I 002
06/01 09:15 I 003
06/01 09:30 O 002
06/01 10:00 I 000
06/01 10:15 O 001
06/01 10:30 I 002
06/01 10:45 O 002
```

```
06/01 11:00 I 001
06/01 11:15 O 000
06/01 11:30 I 002
06/01 11:45 O 001
06/01 12:00 O 002
06/01 12:15 I 000
06/01 12:30 I 002
06/01 12:45 O 000
06/01 13:00 I 000
06/01 13:15 O 000
06/01 13:30 O 002
06/01 13:45 O 003
0
```

Output for the Sample Input

```
45
120
```

```cpp
/*
ACM-ICPC Live Archive 5851. Gift from the Goddess of Programming
For the sake of simplicity, convert all the dates to a single integer
(see function convert() below). Each visitant (even the goddess)
defines a set of disjoint intervals. For each visitant different
from the goddes, find the intersection of its set of intervals
with the goddess' set. Output the size of the largest intersection.
*/
#include <iostream>
#include <vector>
using namespace std;

#define PB push_back

typedef vector<int> Vi;
typedef vector<Vi> Mi;

int convert(int month, int day, int hour, int minute) {
  int res = minute;
  res += 60*hour;
  res += 24*60*day;
  res += 31*24*60*month;
  return res;
}

int intersect(const Vi& va, const Vi& vb) {
  int res = 0;
  int na = va.size(), nb = vb.size();
  int j = 0;
  for (int i = 0; i < na; i += 2) {
    while (j < nb) {
      res += max(0, min(va[i + 1], vb[j + 1]) - max(va[i], vb[j]));
      if (va[i + 1] < vb[j + 1]) break;
      j += 2;
    }
  }
  return res;
}

int main() {
  int n;
  while (cin >> n and n > 0) {
    Mi mat(1000);
    for (int i = 0; i < n; ++i) {
      char spam;
      int month, day, hour, minute, id;
      cin >> month >> spam >> day >> hour >> spam >> minute >> spam >> id;
      mat[id].PB(convert(month, day, hour, minute));
    }

    int res = 0;
    for (int i = 1; i < 1000; ++i)
      res = max(res, intersect(mat[0], mat[i]));
    cout << res << endl;
  }
}
```

ACM International Collegiate Programming Contest
Asia Regional Contest, Fukuoka, 2011–11–13

Problem B
The Sorcerer's Donut

Input: Standard Input
Time Limit: 30 seconds

Your master went to the town for a day. You could have a relaxed day without hearing his scolding. But he ordered you to make donuts dough by the evening. Loving donuts so much, he can't live without eating tens of donuts everyday. What a chore for such a beautiful day.

But last week, you overheard a magic spell that your master was using. It was the time to try. You casted the spell on a broomstick sitting on a corner of the kitchen. With a flash of lights, the broom sprouted two arms and two legs, and became alive. You ordered him, then he brought flour from the storage, and started kneading dough. The spell worked, and how fast he kneaded it!

A few minutes later, there was a tall pile of dough on the kitchen table. That was enough for the next week. "OK, stop now." You ordered. But he didn't stop. Help! You didn't know the spell to stop him! Soon the kitchen table was filled with hundreds of pieces of dough, and he still worked as fast as he could. If you could not stop him now, you would be choked in the kitchen filled with pieces of dough.

Wait, didn't your master write his spells on his notebooks? You went to his den, and found the notebook that recorded the spell of cessation.

But it was not the end of the story. The spell written in the notebook is not easily read by others. He used a plastic model of a donut as a notebook for recording the spell. He split the surface of the donut-shaped model into square mesh (Figure B.1), and filled with the letters (Figure B.2). He hid the spell so carefully that the pattern on the surface looked meaningless. But you knew that he wrote the pattern so that the spell "appears" more than once (see the next paragraph for the precise conditions). The spell was not necessarily written in the left-to-right direction, but any of the 8 directions, namely left-to-right, right-to-left, top-down, bottom-up, and the 4 diagonal directions.

You should be able to find the spell as the longest string that appears more than once. Here, a string is considered to appear more than once if there are square sequences having the string on the donut that satisfy the following conditions.

- Each square sequence does not overlap itself. (Two square sequences can share some squares.)
- The square sequences start from different squares, and/or go to different directions.

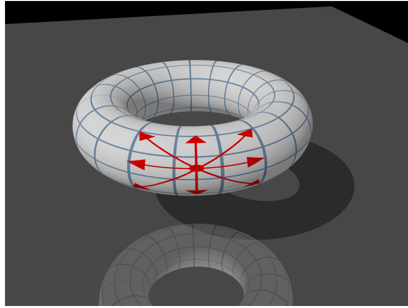


Figure B.1: The Sorcerer's Donut Before Filled with Letters, Showing the Mesh and 8 Possible Spell Directions

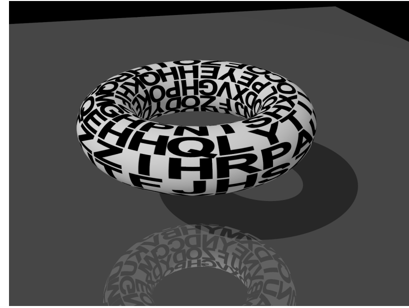


Figure B.2: The Sorcerer's Donut After Filled with Letters

Note that a palindrome (i.e., a string that is the same whether you read it backwards or forwards) that satisfies the first condition "appears" twice.

The pattern on the donut is given as a matrix of letters as follows.

```
ABCD
EFGH
IJKL
```

Note that the surface of the donut has no ends; the top and bottom sides, and the left and right sides of the pattern are respectively connected. There can be square sequences longer than both the vertical and horizontal lengths of the pattern. For example, from the letter `F` in the above pattern, the strings in the longest non-self-overlapping sequences towards the 8 directions are as follows.

```
FGHE
FKDEJCHIBGLA
FJB
FIDGJAHKBELC
FEHG
FALGBIHCJEDK
FBJ
FCLEBKHAJGDI
```

Please write a program that finds the magic spell before you will be choked with pieces of donuts dough.

Input

The input is a sequence of datasets. Each dataset begins with a line of two integers h and w, which denote the size of the pattern, followed by h lines of w uppercase letters from `A` to `Z`, inclusive, which denote the pattern on the donut. You may assume $3 \leq h \leq 10$ and $3 \leq w \leq 20$.

The end of the input is indicated by a line containing two zeros.

Output

For each dataset, output the magic spell. If there is more than one longest string of the same length, the first one in the dictionary order must be the spell. The spell is known to be at least two letters long. When no spell is found, output `0` (zero).

Sample Input

```
5 7
RRCABXT
AABMFAB
RROMJAC
APTADAB
YABADAO
3 13
ABCDEFGHIJKLM
XMADAMIMADAMY
ACEGIKMOQSUWY
3 4
DEFG
ACAB
HIJK
3 6
ABCDEF
GHIAKL
MNOPQR
10 19
JFZODYDXMZZPEYTRNCW
XVGHPOKEYNZTQFZJKOD
EYEHHQKHFZOVNRGOOLP
QFZOIHRQMGHPNISHXOC
DRGILJHSQEHHQLYTILL
NCSHQMKHTZZIHRPAUJA
NCCTINCLAUTFJHSZBVK
LPBAUJIUMBVQYKHTZCW
XMYHBVKUGNCWTLLAUID
EYNDCCWLEOODXYUMBVN
0 0
```

Output for the Sample Input

```
ABRACADABRA
MADAMIMADAM
ABAC
0
ABCDEFGHIJKLMNOPQRSTUVWXYZHHHHHABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
/*
ACM-ICPC Live Archive 5852. The Sorcerer's Donut
Find from each point on the surface and each of the 8 possible
directions, the longest word starting there in that direction that
does not overlap itself. Sort lexicographically all the words found,
and find the first two consecutive words whose common prefix is maximal.
Output that common prefix. Don't forget the case where there aren't
two words with a non-empty common prefix.
*/
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

typedef vector<string> Vs;

const int diri[8] = { -1, -1, 0, 1, 1, 1, 0, -1 };
const int dirj[8] = { 0, 1, 1, 1, 0, -1, -1, -1 };

int R, C;
Vs mapa;

int vist[30][30], T;

void fun(int x, int y, int dx, int dy, string& res) {
  res = "";
  ++T;
  while (vist[x][y] != T) {
    res += mapa[x][y];
    vist[x][y] = T;
    x += dx;
    if (x < 0) x = R - 1;
    else if (x == R) x = 0;
    y += dy;
    if (y < 0) y = C - 1;
    else if (y == C) y = 0;
  }
}

int compref(const string& a, const string& b) {
  int n = min(a.size(), b.size());
  for (int i = 0; i < n; ++i)
    if (a[i] != b[i]) return i;
  return n;
}

int main() {
  while (cin >> R >> C and R > 0) {
    mapa = Vs(R);
    for (int i = 0; i < R; ++i) cin >> mapa[i];

    Vs v(R*C*8);
    for (int i = 0; i < R; ++i)
      for (int j = 0; j < C; ++j)
        for (int k = 0; k < 8; ++k)
          fun(i, j, diri[k], dirj[k], v[8*C*i + 8*j + k]);

    int maxi = 1, p = -1;
    sort(v.begin(), v.end());
    int n = v.size();
```

```
    for (int i = 0; i + 1 < n; ++i) {
      int t = compref(v[i], v[i + 1]);
      if (t > maxi) {
        maxi = t;
        p = i;
      }
    }

    if (p == -1) cout << 0 << endl;
    else cout << v[p].substr(0, maxi) << endl;
  }
}
```

ACM-ICPC Live Archive 5854. Long Distance Taxi

ACM International Collegiate Programming Contest
Asia Regional Contest, Fukuoka, 2011–11–13

Problem D
Long Distance Taxi

Input: Standard Input
Time Limit: 30 seconds

A taxi driver, Nakamura, was so delighted because he got a passenger who wanted to go to a city thousands of kilometers away. However, he had a problem. As you may know, most taxis in Japan run on liquefied petroleum gas (LPG) because it is cheaper than gasoline. There are more than 50,000 gas stations in the country, but less than one percent of them sell LPG. Although the LPG tank of his car was full, the tank capacity is limited and his car runs 10 kilometer per liter, so he may not be able to get to the destination without filling the tank on the way. He knew all the locations of LPG stations.

Your task is to write a program that finds the best way from the current location to the destination without running out of gas.

Input

The input consists of several datasets, and each dataset is in the following format.

$N \quad M \quad cap$
$src \quad dest$
$c_{1,1} \quad c_{1,2} \quad d_1$
$c_{2,1} \quad c_{2,2} \quad d_2$
$\vdots$
$c_{N,1} \quad c_{N,2} \quad d_N$
s_1
s_2
$\vdots$
s_M

The first line of a dataset contains three integers (N, M, cap), where N is the number of roads $(1 \leq N \leq 3000)$, M is the number of LPG stations $(1 \leq M \leq 300)$, and cap is the tank capacity $(1 \leq cap \leq 200)$ in liter. The next line contains the name of the current city (src) and the name of the destination city ($dest$). The destination city is always different from the current city. The following N lines describe roads that connect cities. The road i $(1 \leq i \leq N)$ connects two different cities $c_{i,1}$ and $c_{i,2}$ with an integer distance d_i $(0 < d_i \leq 2000)$ in kilometer, and he can go from either city to the other. You can assume that no two different roads connect the same pair of cities. The columns are separated by a single space. The next M lines $(s_1, s_2, \ldots, s_M)$

11

indicate the names of the cities with LPG station. You can assume that a city with LPG station has at least one road.

The name of a city has no more than 15 characters. Only English alphabet ('A' to 'Z' and 'a' to 'z', case sensitive) is allowed for the name.

A line with three zeros terminates the input.

Output

For each dataset, output a line containing the length (in kilometer) of the shortest possible journey from the current city to the destination city. If Nakamura cannot reach the destination, output "`-1`" (without quotation marks). You must not output any other characters.

The actual tank capacity is usually a little bit larger than that on the specification sheet, so you can assume that he can reach a city even when the remaining amount of the gas becomes exactly zero. In addition, you can always fill the tank at the destination so you do not have to worry about the return trip.

Sample Input

```
6 3 34
Tokyo Kyoto
Tokyo Niigata 335
Tokyo Shizuoka 174
Shizuoka Nagoya 176
Nagoya Kyoto 195
Toyama Niigata 215
Toyama Kyoto 296
Nagoya
Niigata
Toyama
6 3 30
Tokyo Kyoto
Tokyo Niigata 335
Tokyo Shizuoka 174
Shizuoka Nagoya 176
Nagoya Kyoto 195
Toyama Niigata 215
Toyama Kyoto 296
Nagoya
Niigata
Toyama
0 0 0
```

Output for the Sample Input

```
846
-1
```

```
/*
ACM-ICPC Live Archive 5854. Long Distance Taxi
This problem can be solved with the Dijkstra algorithm.
A state is formed by a pair of two integers: the node where the
car stands, and the liters of gasoline in its tank. The objective is
to minimize the distance to the destination, no matter what the
amount of gasoline in the tank is at the end. In order to make
the algorithm fast enough, ignore a state when there exists
another one with the same city, more liters, and the same or
better distance found.
*/
#include <iostream>
#include <map>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

#define PB push_back
#define X first
#define Y second

typedef map<string, int> MAP;
typedef pair<int, int> P;
typedef priority_queue<P> PQ;
typedef vector<int> Vi;
typedef vector<Vi> Mi;

int N, M, S, C;
Vi st;
Mi net, netd;
Vi minim;

int dist[1<<23], tdist[1<<23], T;

inline int crea(int n, int c) {
  return (n<<11) + c;
}

inline int getn(int m) {
  return m>>11;
}

inline int getc(int m) {
  return m&((1<<11) - 1);
}

int main() {
  while (cin >> M >> S >> C and M > 0) {
    C *= 10;
    N = 0;

    string s1, s2;
    cin >> s1 >> s2;

    MAP mp;
    mp[s1] = N++;
    mp[s2] = N++;

    net = netd = Mi(2);
```

```
  for (int i = 0; i < M; ++i) {
    string a, b;
    int c;
    cin >> a >> b >> c;
    if (mp.count(a) == 0) {
      mp[a] = N++;
      net.PB(Vi());
      netd.PB(Vi());
    }
    if (mp.count(b) == 0) {
      mp[b] = N++;
      net.PB(Vi());
      netd.PB(Vi());
    }
    int ta = mp[a];
    int tb = mp[b];
    net[ta].PB(tb);
    netd[ta].PB(c);
    net[tb].PB(ta);
    netd[tb].PB(c);
  }

  st = Vi(N, 0);
  for (int i = 0; i < S; ++i) {
    string s;
    cin >> s;
    if (mp.count(s)) st[mp[s]] = 1;
  }

  ++T;

  int res = -1;

  int start = crea(0, C);

  dist[start] = 0;
  tdist[start] = T;

  minim = Vi(N, 0);

  PQ q;
  q.push(P(0, start));

  while (not q.empty()) {
    int d = -q.top().X;
    int current = q.top().Y;
    q.pop();

    if (d != dist[current]) continue;

    int n = getn(current);
    int c = getc(current);

    if (n == 1) {
      res = d;
      break;
    }

    minim[n] = max(minim[n], c);

    for (int i = 0; i < net[n].size(); ++i) {
```

```
        int nn = net[n][i];
        int dd = d + netd[n][i];
        int cc = c - netd[n][i];
        if (cc < 0) continue;
        if (st[nn]) cc = C;

        if (minim[nn] != 0 and cc <= minim[nn]) continue;

        int next = crea(nn, cc);
        if (tdist[next] != T or dd < dist[next]) {
          dist[next] = dd;
          tdist[next] = T;
          q.push(P(-dd, next));
        }
      }
    }

    cout << res << endl;
  }
}
```

ACM International Collegiate Programming Contest
Asia Regional Contest, Fukuoka, 2011–11–13

Problem E
Driving an Icosahedral Rover

Input: Standard Input
Time Limit: 60 seconds

After decades of fruitless efforts, one of the expedition teams of ITO (Intersolar Tourism Organization) finally found a planet that would surely provide one of the best tourist attractions within a ten light-year radius from our solar system. The most attractive feature of the planet, besides its comfortable gravity and calm weather, is the area called *Mare Triangularis.* Despite the name, the area is not covered with water but is a great plane. Its unique feature is that it is divided into equilateral triangular sections of the same size, called *trigons.* The *trigons* provide a unique impressive landscape, a must for tourism. It is no wonder the board of ITO decided to invest a vast amount on the planet.

Despite the expected secrecy of the staff, the Society of Astrogeology caught this information in no time, as always. They immediately sent their president's letter to the Institute of Science and Education of the Commonwealth Galactica claiming that authoritative academic inspections were to be completed before any commercial exploitation might damage the nature.

Fortunately, astrogeologists do not plan to practice all the possible inspections on all of the *trigons;* there are far too many of them. Inspections are planned only on some characteristic *trigons* and, for each of them, in one of twenty different scientific aspects.

To accelerate building this new tourist resort, ITO's construction machinery team has already succeeded in putting their brand-new invention in practical use. It is a rover vehicle of the shape of an *icosahedron,* a regular polyhedron with twenty faces of equilateral triangles. The machine is customized so that each of the twenty faces exactly fits each of the *trigons*. Controlling the high-tech *gyromotor* installed inside its body, the rover can roll onto one of the three *trigons* neighboring the one its bottom is on.

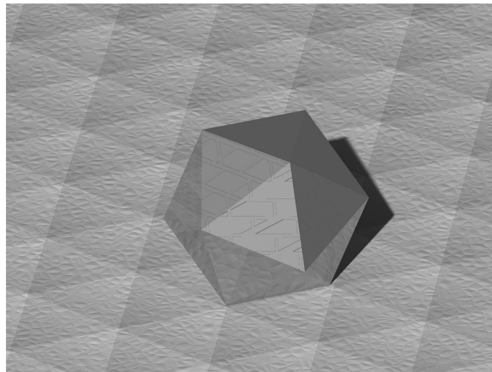


Figure E.1: The Rover on Mare Triangularis

Each of the twenty faces has its own function. The set of equipments installed on the bottom face touching the ground can be applied to the *trigon* it is on. Of course, the rover was meant

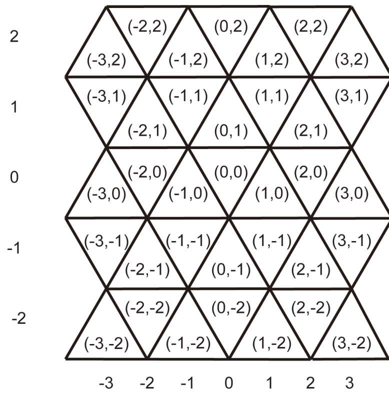


Figure E.2: The Coordinate System

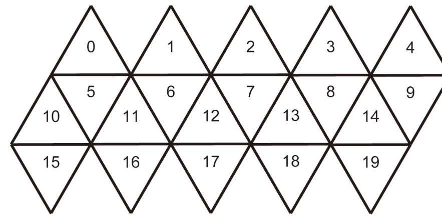


Figure E.3: Face Numbering

to accelerate construction of the luxury hotels to host rich interstellar travelers, but, changing the installed equipment sets, it can also be used to accelerate academic inspections.

You are the driver of this rover and are asked to move the vehicle onto the *trigon* specified by the leader of the scientific commission with the smallest possible steps. What makes your task more difficult is that the designated face installed with the appropriate set of equipments has to be the bottom. The direction of the rover does not matter.

The *trigons* of *Mare Triangularis* are given two-dimensional coordinates as shown in Figure E.2. Like maps used for the Earth, the x axis is from the west to the east, and the y axis is from the south to the north. Note that all the *trigons* with its coordinates (x, y) has neighboring *trigons* with coordinates $(x - 1, y)$ and $(x + 1, y)$. In addition to these, when $x + y$ is even, it has a neighbor $(x, y + 1)$; otherwise, that is, when $x + y$ is odd, it has a neighbor $(x, y - 1)$.

Figure E.3 shows a development of the skin of the rover. The top face of the development makes the exterior. That is, if the numbers on faces of the development were actually marked on the faces of the rover, they should been readable from its outside. These numbers are used to identify the faces.

When you start the rover, it is on the *trigon* $(0, 0)$ and the face 0 is touching the ground. The rover is placed so that rolling towards north onto the *trigon* $(0, 1)$ makes the face numbered 5 to be at the bottom.

As your first step, you can choose one of the three adjacent *trigons,* namely those with coordinates $(-1, 0)$, $(1, 0)$, and $(0, 1)$, to visit. The bottom will be the face numbered 4, 1, and 5, respectively. If you choose to go to $(1, 0)$ in the first rolling step, the second step can bring the rover to either of $(0, 0)$, $(2, 0)$, or $(1, -1)$. The bottom face will be either of 0, 6, or 2, correspondingly. The rover may visit any of the *trigons* twice or more, including the start and

the goal *trigons,* when appropriate.

The theoretical design section of ITO showed that the rover can reach any goal *trigon* on the specified bottom face within a finite number of steps.

Input

The input consists of a number of datasets. The number of datasets does not exceed 50.

Each of the datasets has three integers x, y, and n in one line, separated by a space. Here, (x, y) specifies the coordinates of the *trigon* to which you have to move the rover, and n specifies the face that should be at the bottom.

The end of the input is indicated by a line containing three zeros.

Output

The output for each dataset should be a line containing a single integer that gives the *minimum number* of steps required to set the rover on the specified *trigon* with the specified face touching the ground. No other characters should appear in the output.

You can assume that the maximum number of required steps does not exceed 100. *Mare Triangularis* is broad enough so that any of its edges cannot be reached within that number of steps.

Sample Input

```
0 0 1
3 5 2
-4 1 3
13 -13 2
-32 15 9
-50 50 0
0 0 0
```

Output for the Sample Input

```
6
10
9
30
47
100
```

```
/*
ACM-ICPC Live Archive 5855. Driving an Icosahedral Rover
Run a BFS algorithm in a graph where every state is not just
the position in the map where the icosahedron is located at,
but also the face which is touching the ground and the orientation
of the same. Although the implementation is a bit tricky, the
declaration of constant arrays telling what is the next
face/orientation if rolling the icosahedral in a given direction
and this kind of things, instead of having them hardcoded in the
main code, may help to simplify it.
*/
#include <iostream>
#include <queue>
using namespace std;

const int F[20][3] = {
  { 1, 5, 4 },
  { 2, 6, 0 },
  { 3, 7, 1 },
  { 4, 8, 2 },
  { 0, 9, 3 },
  { 10, 0, 11 },
  { 11, 1, 12 },
  { 12, 2, 13 },
  { 13, 3, 14 },
  { 14, 4, 10 },
  { 5, 15, 9 },
  { 6, 16, 5 },
  { 7, 17, 6 },
  { 8, 18, 7 },
  { 9, 19, 8 },
  { 19, 10, 16 },
  { 15, 11, 17 },
  { 16, 12, 18 },
  { 17, 13, 19 },
  { 18, 14, 15 }
};

int D[20][20];
int vist[1<<23], T;

inline int crea(int x, int y, int f, int d) {
  x += 128;
  y += 128;
  return d | (f<<2) | (y<<7) | (x<<15);
}

inline int getx(int m) {
  return (m>>15) - 128;
}

inline int gety(int m) {
  return ((m>>7)&255) - 128;
}

inline int getf(int m) {
  return (m>>2)&31;
}

inline int getd(int m) {
  return m&3;
```

```
}

int main() {
  D[19][15] = D[18][19] = D[17][18] = D[16][17] = D[15][16] = 1;
  D[1][0] = D[2][1] = D[3][2] = D[4][3] = D[0][4] = 1;
  for (int i = 0; i < 20; ++i)
    for (int j = 0; j < 20; ++j)
      if (D[i][j] == 1) D[j][i] = 2;

  int X, Y, N;
  while (cin >> X >> Y >> N and (X != 0 or Y != 0 or N != 0)) {
    int res = -1;

    ++T;
    queue<int> q;
    q.push(crea(0, 0, 0, 0));
    for (int step = 0; res == -1 and step <= 100; ++step) {
      for (int qq = q.size(); qq > 0; --qq) {
        int curr = q.front();
        q.pop();

        int x = getx(curr);
        int y = gety(curr);
        int f = getf(curr);
        int d = getd(curr);

        if (x == X and y == Y and f == N) {
          res = step;
          break;
        }

        if ((x + y)%2 == 0) {
          {
            int xx = x - 1;
            int yy = y;
            int ff = F[f][(d + 2)%3];
            int dd = (d + D[f][ff])%3;
            int next = crea(xx, yy, ff, dd);
            if (vist[next] != T) {
              vist[next] = T;
              q.push(next);
            }
          }
          {
            int xx = x;
            int yy = y + 1;
            int ff = F[f][(d + 1)%3];
            int dd = (d + D[f][ff])%3;
            int next = crea(xx, yy, ff, dd);
            if (vist[next] != T) {
              vist[next] = T;
              q.push(next);
            }
          }
          {
            int xx = x + 1;
            int yy = y;
            int ff = F[f][d];
            int dd = (d + D[f][ff])%3;
            int next = crea(xx, yy, ff, dd);
            if (vist[next] != T) {
```

```
              vist[next] = T;
              q.push(next);
            }
          }
        }
        else {
          {
            int xx = x - 1;
            int yy = y;
            int ff = F[f][d];
            int dd = (d + D[f][ff])%3;
            int next = crea(xx, yy, ff, dd);
            if (vist[next] != T) {
              vist[next] = T;
              q.push(next);
            }
          }
          {
            int xx = x;
            int yy = y - 1;
            int ff = F[f][(d + 1)%3];
            int dd = (d + D[f][ff])%3;
            int next = crea(xx, yy, ff, dd);
            if (vist[next] != T) {
              vist[next] = T;
              q.push(next);
            }
          }
          {
            int xx = x + 1;
            int yy = y;
            int ff = F[f][(d + 2)%3];
            int dd = (d + D[f][ff])%3;
            int next = crea(xx, yy, ff, dd);
            if (vist[next] != T) {
              vist[next] = T;
              q.push(next);
            }
          }
        }
      }
    }

    cout << res << endl;
  }
}
```

ACM International Collegiate Programming Contest
Asia Regional Contest, Fukuoka, 2011–11–13

Problem F
City Merger

Input: Standard Input
Time Limit: 60 seconds

Recent improvements in information and communication technology have made it possible to provide municipal service to a wider area more quickly and with less costs. Stimulated by this, and probably for saving their not sufficient funds, mayors of many cities started to discuss on mergers of their cities.

There are, of course, many obstacles to actually put the planned mergers in practice. Each city has its own culture of which citizens are proud. One of the largest sources of friction is with the name of the new city. All citizens would insist that the name of the new city should have the original name of their own city at least as a part of it. Simply concatenating all the original names would, however, make the name too long for everyday use.

You are asked by a group of mayors to write a program that finds the shortest possible name for the new city that includes all the original names of the merged cities. If two or more cities have common parts, they can be overlapped. For example, if "`FUKUOKA`", "`OKAYAMA`", and "`YAMAGUCHI`" cities are to be merged, "`FUKUOKAYAMAGUCHI`" is such a name that include all three of the original city names. Although this includes all the characters of the city name "`FUKUYAMA`" in this order, it does not appear as a consecutive substring, and thus "`FUKUYAMA`" is not considered to be included in the name.

Input

The input is a sequence of datasets. Each dataset begins with a line containing a positive integer n ($n \leq 14$), which denotes the number of cities to be merged. The following n lines contain the names of the cities in uppercase alphabetical letters, one in each line. You may assume that none of the original city names has more than 20 characters. Of course, no two cities have the same name.

The end of the input is indicated by a line consisting of a zero.

Output

For each dataset, output the length of the shortest possible name of the new city in one line. The output should not contain any other characters.

Sample Input

```
3
FUKUOKA
OKAYAMA
YAMAGUCHI
3
FUKUOKA
FUKUYAMA
OKAYAMA
2
ABCDE
EDCBA
4
GA
DEFG
CDDE
ABCD
2
ABCDE
C
14
AAAAA
BBBBB
CCCCC
DDDDD
EEEEE
FFFFF
GGGGG
HHHHH
IIIII
JJJJJ
KKKKK
LLLLL
MMMMM
NNNNN
0
```

Output for the Sample Input

```
16
19
9
9
5
70
```

```
/*
ACM-ICPC Live Archive 5856. City Merger
First of all, we can discard city names that appear as substring of other city
names. Now, the problem can be solved using Dynamic Programming, where a state
is the shortest possible way of merging a subset S of the city names, ending in
one particular city name E. From one state we can update states that are the
result of merging one more city name, and the increase in length can be computed
as the length of the new city name minus its maximum overlap with a suffix of E.
*/
#include <iostream>
#include <string>
#include <vector>

using namespace std;

const int INF = 1000000000;

vector<vector<int> > mem_cost;

string dec_to_bin(int n, int k = 1) {
  string ret = "";
  for (; n > 0 || k > 0; n >>= 1, --k) {
    ret = char('0'+(n&1))+ret;
  }
  return ret;
}

int compute_overlap(const string& a, const string& b) {
  for (int overlap = int(min(a.size(), b.size()))-1; overlap > 0; --overlap) {
    bool eq = true;
    for (int i = int(a.size())-overlap, j = 0; i < int(a.size()) && eq; ++i, ++j) {
      if (a[i] != b[j]) {
        eq = false;
      }
    }
    if (eq) {
      return overlap;
    }
  }
  return 0;
}

int cost(const vector<string>& names, const vector<vector<int> >& overlap, int rem,
    int cur) {
  if (rem == 0) {
    return 0;
  }

  int &ret = mem_cost[rem][cur];
  if (ret != -1) {
    return ret;
  }
  ret = INF;
  int n = int(overlap.size());
  for (int nxt = 0; nxt < n; ++nxt) if (((rem>>nxt)&1) == 1) {
    ret = min(ret, cost(names, overlap, rem&(~(1<<nxt)), nxt)+int(names[nxt].size())
        -overlap[cur][nxt]);
  }
  return ret;
}
```

```
int main() {
  for (int n; cin >> n && n > 0;) {
    vector<string> names(n);
    for (int i = 0; i < n; ++i) {
      cin >> names[i];
    }
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n && names[i].size() > 0; ++j) if (i != j) {
        if (names[i].size() < names[j].size() && names[j].find(names[i]) != string::
            npos) {
          names[i] = "";
        }
      }
    }
    vector<string> tmp;
    for (int i = 0; i < n; ++i) {
      if (names[i].size() > 0) {
        tmp.push_back(names[i]);
      }
    }
    swap(names, tmp);
    n = int(names.size());
    vector<vector<int> > overlap(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) if (i != j) {
        overlap[i][j] = compute_overlap(names[i], names[j]);
      }
    }
    mem_cost = vector<vector<int> >(1<<n, vector<int>(n, -1));
    int ans = INF;
    for (int first = 0; first < n; ++first) {
      ans = min(ans, int(names[first].size())+cost(names, overlap, ((1<<n)-1)&(~(1<<
          first)), first));
    }
    cout << ans << endl;
  }
}
```

ACM International Collegiate Programming Contest
Asia Regional Contest, Fukuoka, 2011–11–13

Problem G
Captain Q's Treasure

Input: Standard Input
Time Limit: 30 seconds

You got an old map, which turned out to be drawn by the infamous pirate "Captain Q". It shows the locations of a lot of treasure chests buried in an island.

The map is divided into square sections, each of which has a digit on it or has no digit. The digit represents the number of chests in its 9 neighboring sections (the section itself and its 8 neighbors). You may assume that there is at most one chest in each section.

Although you have the map, you can't determine the sections where the chests are buried. Even the total number of chests buried in the island is unknown. However, it is possible to calculate the minimum number of chests buried in the island. Your mission in this problem is to write a program that calculates it.

Input

The input is a sequence of datasets. Each dataset is formatted as follows.

> h w
> *map*

The first line of a dataset consists of two positive integers h and w. h is the height of the map and w is the width of the map. You may assume $1 \leq h \leq 15$ and $1 \leq w \leq 15$.

The following h lines give the map. Each line consists of w characters and corresponds to a horizontal strip of the map. Each of the characters in the line represents the state of a section as follows.

'`.`': The section is not a part of the island (water). No chest is here.

'`*`': The section is a part of the island, and the number of chests in its 9 neighbors is not known.

'`0`'–'`9`': The section is a part of the island, and the digit represents the number of chests in its 9 neighbors.

You may assume that the map is not self-contradicting, i.e., there is at least one arrangement of chests. You may also assume the number of sections with digits is at least one and at most 15.

A line containing two zeros indicates the end of the input.

Output

For each dataset, output a line that contains the minimum number of chests. The output should not contain any other character.

Sample Input

```
5 6
*2.2**
..*...
..2...
..*...
*2.2**
6 5
.*2*.
..*..
..*..
..2..
..*..
.*2*.
5 6
.1111.
**...*
33....
**...0
.*2**.
6 9
....1....
...1.1...
....1....
.1..*..1.
1.1***1.1
.1..*..1.
9 9
*********
*4*4*4*4*
*********
*4*4*4*4*
*********
*4*4*4*4*
*********
*4*4*4***
*********
0 0
```

Output for the Sample Input

```
6
5
5
6
23
```

```
/*
ACM-ICPC Live Archive 5857. Captain Q's Treasure
This problem can be written as a minimization problem with linear restrictions.
X_i,j is a variable that has a value of 1 if there is a chest in cell (i, j)
and a value of 0 otherwise. The map gives us restrictions of the form
X_i1,j1 + X_i2,j2 + ... + X_ik,jk = h.
Running the Simplex algorithm with the cost function
c(X) = sum for all i,j of X_i,j
will yield a solution for this problem except that X_i,j will not necessarily
be integer (they should be either 0 or 1, but the algorithm will sometimes
output real values between 0 and 1). However, since the problem that we want
to solve has an additional restriction, the result of the Simplex algorithm is
a lower bound on the solution.

The following part of the solution is no more than a conjecture, although the
fact that this program solved all the test cases correctly in the ACM ICPC
Live Archive online judge suggests that it might be correct:

If we run the Simplex algorithm with an additional restriction that forces the
solution (the cost function c(X)) to be greater or equal to a certain value K,
we should get an integer solution as soon as K is no less than the actual
solution to the problem. Making this assumption, we can initially set K to the
lower bound on the solution that we got from the original Simplex execution,
and increase K until the Simplex algorithm yields an integer solution.

The search on K can be improved by increasing it exponentially and then as soon
as the Simplex yields an integer solution perform a binary search in order to
find the value at which the solution goes from non-integer to integer.
*/
#include <algorithm>
#include <cmath>
#include <iostream>
#include <numeric>
#include <vector>

#define REP(i, n) for (int i = 0; i < (n); ++i)
#define ALL(x) (x).begin(),(x).end()

using namespace std;

typedef long double ld;
typedef vector<double> VD;
typedef vector<VD> mat;
typedef vector<int> VI;
typedef pair<int, int> PII;

const double INF = 1e20;
const double EPS = 1e-9;
const int DY[] = {-1, -1, 0, 1, 1, 1, 0, -1};
const int DX[] = {0, 1, 1, 1, 0, -1, -1, -1};

vector<int> mem_integer_solution;

void pivot(mat& A, VD& b, int n, int m, int bi, int nv) {
  ld pd = A[bi][nv];
  REP(j, n+m) A[bi][j] /= pd;
  b[bi] /= pd;
  REP(i, m+2) {
    if (i == bi) continue;
    ld pn = A[i][nv];
    REP(j, n+m) A[i][j] -= A[bi][j] * pn;
```

```
    b[i] -= b[bi] * pn;
  }
}

VD simplex(mat A, VD b, VD c) {
  const int n = c.size(), m = b.size(); // n vars, m eqs
  //modify b to non-negative
  REP(i, m) if (b[i] < 0) {
    REP(j, n) A[i][j] *= -1;
    b[i] *= -1;
  }
  // list of base/independent variable ids
  vector<int> bx(m), nx(n);
  REP(i, m) bx[i] = n+i; // aux in base
  REP(i, n) nx[i] = i; // real independent
  // extend A, b
  A.resize(m + 2);
  REP(i, m+2) A[i].resize(n + m, 0); // aux vars
  REP(i, m) A[i][n+i] = 1; // aux vars
  REP(i, m) REP(j, n) A[m][j] = A[m][j]+A[i][j]; // row m is phase 0 costs
  b.push_back(accumulate(ALL(b), ld(0))); // row m is phase 0 costs
  REP(j, n) A[m + 1][j] = -c[j]; // row m+1 is costs
  REP(i, m) A[m + 1][n + i] = -INF; // aux costs
  b.push_back(0);
  mat A_(A);
  VD b_(b);
  VI forbidden(n+m, false);
  // main optimization
  REP(phase, 2) {
    for (int nsteps = 0; ; ++nsteps) {
      // select an independent variable to enter
      int ni = -1;
      REP(i, n) {
        if (!forbidden[nx[i]] && A[m][nx[i]] > EPS && (ni < 0 || nx[i] < nx[ni])) {
          ni = i;
        }
      }
      if (ni == -1) break;
      int nv = nx[ni];
      // select a base variable to leave
      VD bound(m);
      REP(i, m) bound[i] = (A[i][nv] < EPS? INF : b[i]/A[i][nv]);
      if (!(*min_element(ALL(bound)) < INF)) return VD(n, INF); // unbounded
      int bi = 0;
      REP(i, m) {
        if (bound[i] < bound[bi] - EPS || (bound[i] < bound[bi] + EPS && bx[i] < bx[
            bi])) {
          bi = i;
        }
      }
      // update
      swap(nx[ni], bx[bi]);
      if (false && nsteps%100 == 0) {
        A = A_;
        b = b_;
      }
      else pivot(A, b, n, m, bi, nv);
    }
    if (phase == 0) {
      if (b[m] > EPS) return VD(0); // no solution
      REP(i, n+m) if (i >= n || A[m][i] < -EPS) forbidden[i] = true;
```

```
      swap(A[m], A[m+1]);
      swap(b[m], b[m+1]);
      swap(A_[m], A_[m+1]);
      swap(b_[m], b_[m+1]);
    }
  }
  VD x(n+m, 0);
  REP(i, m) x[bx[i]] = b[i];
  x.resize(n);
  return x;
}

bool inside(int i, int j, int h, int w) {
  return i >= 0 && j >= 0 && i < h && j < w;
}

bool is_integer(double d) {
  return abs(d-round(d)) < EPS;
}

int integer_solution(vector<vector<double> >& A, vector<double>& b, vector<double>&
    c, int k) {
  int &ret = mem_integer_solution[k];
  if (ret != -2) {
    return ret;
  }
  b[int(b.size())-1] = -k;
  vector<double> x = simplex(A, b, c);
  if (x.size() == 0) {
    return ret = -1;
  }
  for (int i = 0; i < (int(x.size())-1)/2; ++i) {
    if (!is_integer(x[i])) {
      return ret = 0;
    }
  }
  return ret = 1;
}

int main() {
  for (int h, w; cin >> h >> w && (h > 0 || w > 0);) {
    vector<string> map(h);
    for (int i = 0; i < h; ++i) {
      cin >> map[i];
    }
    vector<vector<int> > id(h, vector<int>(w, -1));
    int cnt = 0;
    for (int i = 0; i < h; ++i) {
      for (int j = 0; j < w; ++j) if (map[i][j] != '.') {
        bool potential_chest = false;
        if (map[i][j] >= '0' && map[i][j] <= '9') {
          potential_chest = true;
        }
        for (int dir = 0; dir < 8 && !potential_chest; ++dir) {
          int ai = i+DY[dir];
          int aj = j+DX[dir];
          if (inside(ai, aj, h, w) && (map[ai][aj] >= '0' && map[ai][aj] <= '9')) {
            potential_chest = true;
          }
        }
        if (potential_chest) {
```

```
        id[i][j] = cnt;
        ++cnt;
      }
    }
  }
  int m = 2*cnt+1;
  vector<vector<double> > A;
  vector<double> b;

  for (int i = 0; i < cnt; ++i) {
    vector<double> row(m, 0.0);
    row[i] = row[cnt+i] = 1.0;
    A.push_back(row);
    b.push_back(1.0);
  }

  for (int i = 0; i < h; ++i) {
    for (int j = 0; j < w; ++j) if (map[i][j] >= '0' && map[i][j] <= '9') {
      vector<double> row(m, 0.0);
      row[id[i][j]] = 1.0;
      for (int dir = 0; dir < 8; ++dir) {
        int ai = i+DY[dir];
        int aj = j+DX[dir];
        if (inside(ai, aj, h, w) && id[ai][aj] != -1) {
          row[id[ai][aj]] = 1.0;
        }
      }
      A.push_back(row);
      b.push_back(int(map[i][j]-'0'));
    }
  }

  vector<double> c(m, 0.0);
  for (int i = 0; i < cnt; ++i) {
    c[i] = 1.0;
  }

  vector<double> x = simplex(A, b, c);
  double xsum = 0.0;
  bool all_int = true;
  for (int i = 0; i < cnt; ++i) {
    xsum += x[i];
    if (!is_integer(x[i])) {
      all_int = false;
    }
  }
  int k = int(round(xsum));

  vector<double> restr(m, 0.0);
  for (int i = 0; i < cnt; ++i) {
    restr[i] = -1.0;
  }
  restr[2*cnt] = 1.0;
  A.push_back(restr);
  b.push_back(0);

  mem_integer_solution = vector<int>(cnt+1, -2);
  if (all_int) {
    mem_integer_solution[k] = 1;
  }
  int lo = k-1;
```

```
    int hi = k;
    for (; hi < cnt && integer_solution(A, b, c, hi) == 0;) {
      hi = min(cnt, lo+(hi-lo)*2);
    }
    for (; hi-lo > 1;) {
      int md = (lo+hi)/2;
      if (integer_solution(A, b, c, md) == 0) {
        lo = md;
      }
      else {
        hi = md;
      }
    }
    cout << hi << endl;
  }
}
```

ACM International Collegiate Programming Contest
Asia Regional Contest, Fukuoka, 2011–11–13

Problem J
Round Trip

Input: Standard Input
Time Limit: 30 seconds

Jim is planning to visit one of his best friends in a town in the mountain area. First, he leaves his hometown and goes to the destination town. This is called the go phase. Then, he comes back to his hometown. This is called the return phase. You are expected to write a program to find the minimum total cost of this trip, which is the sum of the costs of the go phase and the return phase.

There is a network of towns including these two towns. Every road in this network is one-way, i.e., can only be used towards the specified direction. Each road requires a certain cost to travel.

In addition to the cost of roads, it is necessary to pay a specified fee to go through each town on the way. However, since this is the visa fee for the town, it is not necessary to pay the fee on the second or later visit to the same town.

The altitude (height) of each town is given. On the go phase, the use of descending roads is inhibited. That is, when going from town a to b, the altitude of a should not be greater than that of b. On the return phase, the use of ascending roads is inhibited in a similar manner. If the altitudes of a and b are equal, the road from a to b can be used on both phases.

Input

The input consists of multiple datasets, each in the following format.

$n \quad m$
$d_2 \quad e_2$
$d_3 \quad e_3$
$\vdots$
$d_{n-1} \quad e_{n-1}$
$a_1 \quad b_1 \quad c_1$
$a_2 \quad b_2 \quad c_2$
$\vdots$
$a_m \quad b_m \quad c_m$

Every input item in a dataset is a non-negative integer. Input items in a line are separated by a space.

n is the number of towns in the network. m is the number of (one-way) roads. You can assume the inequalities $2 \le n \le 50$ and $0 \le m \le n(n-1)$ hold. Towns are numbered from 1 to n, inclusive. The town 1 is Jim's hometown, and the town n is the destination town.

d_i is the visa fee of the town i, and e_i is its altitude. You can assume $1 \le d_i \le 1000$ and $1 \le e_i \le 999$ for $2 \le i \le n-1$. The towns 1 and n do not impose visa fee. The altitude of the town 1 is 0, and that of the town n is 1000. Multiple towns may have the same altitude, but you can assume that there are no more than 10 towns with the same altitude.

The j-th road is from the town a_j to b_j with the cost c_j ($1 \le j \le m$). You can assume $1 \le a_j \le n$, $1 \le b_j \le n$, and $1 \le c_j \le 1000$. You can directly go from a_j to b_j, but not from b_j to a_j unless a road from b_j to a_j is separately given. There are no two roads connecting the same pair of towns towards the same direction, that is, for any i and j such that $i \ne j$, $a_i \ne a_j$ or $b_i \ne b_j$. There are no roads connecting a town to itself, that is, for any j, $a_j \ne b_j$.

The last dataset is followed by a line containing two zeros (separated by a space).

Output

For each dataset in the input, a line containing the minimum total cost, including the visa fees, of the trip should be output. If such a trip is not possible, output "`-1`".

Sample Input

```
3 6
3 1
1 2 1
2 3 1
3 2 1
2 1 1
1 3 4
3 1 4
3 6
5 1
1 2 1
2 3 1
3 2 1
2 1 1
1 3 4
3 1 4
4 5
3 1
3 1
1 2 5
2 3 5
3 4 5
4 2 5
```

```
3 1 5
2 1
2 1 1
0 0
```

Output for the Sample Input

```
7
8
36
-1
```

```
/*
ACM-ICPC Live Archive 5860. Round Trip
The part of the statement which says "there are no more than 10
towns with the same altitude" is essential to get to the solution.
Consider the graph with nodes (a,b) which represent the optimal paths
from city 'a' to the origin and from the origin to city 'b'. Run the
Dijkstra algorithm to calculate the minimal cost of reaching every
node (a,b). The final answer is the cost of reaching the node
(destination_city,destination_city). The trickiest part of this
is to use the correct edges with correct weights between nodes,
without counting two times the same fee. There are thre kinds of edge:
(1) (a,b)->(a,c) where 'c' is at a greater level than 'b'. The edge
b->c must exist in the original graph.
(2) (a,b)->(c,b) where 'c' is at a greater level than 'a'. The edge
c->a must exist in the original graph.
(3) (a,b)->(c,d) where 'a', 'b', 'c' and 'd' are at the same level
(a,b,c,d may not be necessarily different).
Edges of kind 1 and 2 should not suppose a problem, while edges of
kind 3 are very tricky. In the code below, dp[a][b][c][d] is the optimal
cost of travelling from node (a,b) to (c,d) when all 'a','b','c','d' are
at the same level. Look at the code to see how dp[][][][] is computed.
*/
#include <iostream>
#include <queue>
#include <utility>
#include <vector>
using namespace std;

#define PB push_back
#define X first
#define Y second

typedef pair<int, int> P;
typedef pair<int, P> PP;
typedef priority_queue<PP> PQ;
typedef vector<P> Vp;
typedef vector<Vp> Mp;
typedef vector<Mp> MMp;
typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef vector<Mi> MMi;

const int INF = 1000000000;

int N, M;
Vi fee, alt;
Mi net, netr;
Mi dis;

Mi altitude;

int ds[11][11][11][11];

int main() {
  while (cin >> N >> M and N > 0) {
    fee = alt = Vi(N, 0);
    for (int i = 1; i < N - 1; ++i) cin >> fee[i] >> alt[i];
    alt[N - 1] = 1000;

    net = netr = Mi(N);
    dis = Mi(N, Vi(N, INF));
```

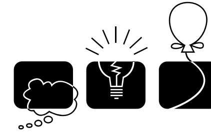
```
  for (int i = 0; i < M; ++i) {
    int a, b, c;
    cin >> a >> b >> c;
    --a; --b;
    dis[a][b] = min(dis[a][b], c);
    if (alt[a] <= alt[b]) net[a].PB(b);
    if (alt[b] <= alt[a]) netr[b].PB(a);
  }

  altitude = Mi(1001);
  for (int i = 0; i < N; ++i)
    altitude[alt[i]].PB(i);

  MMp netx(N, Mp(N));
  MMi netd(N, Mi(N));

  for (int h = 0; h <= 1000; ++h) {
    int n = altitude[h].size();
    if (n < 2) continue;

    for (int i1 = 0; i1 < n; ++i1)
      for (int i2 = 0; i2 < n; ++i2)
        for (int i3 = 0; i3 < n; ++i3)
          for (int i4 = 0; i4 < n; ++i4)
            ds[i1][i2][i3][i4] = INF;

    for (int mask = 0; mask < (1<<n); ++mask) {
      Vi v;
      for (int i = 0; i < n; ++i)
        if ((mask>>i)&1) v.PB(i);
      int m = v.size();

      int s = 0;
      for (int i = 0; i < m; ++i)
        s += fee[altitude[h][v[i]]];

      Mi d(m, Vi(m));
      for (int i = 0; i < m; ++i)
        for (int j = 0; j < m; ++j)
          d[i][j] = dis[altitude[h][v[i]]][altitude[h][v[j]]];

      for (int i = 0; i < m; ++i) dis[i][i] = 0;
      for (int k = 0; k < m; ++k)
        for (int i = 0; i < m; ++i)
          for (int j = 0; j < m; ++j)
            d[i][j] = min(d[i][j], d[i][k] + d[k][j]);

      for (int i1 = 0; i1 < m; ++i1)
        for (int i2 = 0; i2 < m; ++i2)
          for (int j1 = 0; j1 < m; ++j1)
            for (int j2 = 0; j2 < m; ++j2) {
              int t = d[i1][j1] + d[j2][i2] + s - fee[altitude[h][v[i1]]];
              if (i1 != i2) t -= fee[altitude[h][v[i2]]];
              ds[v[i1]][v[i2]][v[j1]][v[j2]] = min(ds[v[i1]][v[i2]][v[j1]][v[j2]],
                   t);
            }
    }

    for (int i1 = 0; i1 < n; ++i1)
      for (int i2 = 0; i2 < n; ++i2)
        for (int j1 = 0; j1 < n; ++j1)
```

```
        for (int j2 = 0; j2 < n; ++j2)
          if (ds[i1][i2][j1][j2] < INF) {
            netx[altitude[h][i1]][altitude[h][i2]].PB(P(altitude[h][j1],
                altitude[h][j2]));
            netd[altitude[h][i1]][altitude[h][i2]].PB(ds[i1][i2][j1][j2]);
          }
  }

  PQ q;
  q.push(PP(0, P(0, 0)));

  Mi dist(N, Vi(N, INF));
  dist[0][0] = 0;

  int res = -1;

  while (not q.empty()) {
    int d = -q.top().X;
    int x = q.top().Y.X;
    int y = q.top().Y.Y;
    q.pop();

    if (d != dist[x][y]) continue;
    if (x == N - 1 and y == N - 1) {
      res = d;
      break;
    }

    for (int i = 0; i < net[x].size(); ++i) {
      int xx = net[x][i];
      int dd = d + dis[x][xx];
      if (xx != y) dd += fee[xx];
      if (dd < dist[xx][y]) {
        dist[xx][y] = dd;
        q.push(PP(-dd, P(xx, y)));
      }
    }

    for (int j = 0; j < netr[y].size(); ++j) {
      int yy = netr[y][j];
      int dd = d + dis[yy][y];
      if (yy != x) dd += fee[yy];
      if (dd < dist[x][yy]) {
        dist[x][yy] = dd;
        q.push(PP(-dd, P(x, yy)));
      }
    }

    for (int i = 0; i < net[x].size(); ++i)
      for (int j = 0; j < netr[y].size(); ++j) {
        int xx = net[x][i], yy = netr[y][j];
        int dd = d + dis[x][xx] + dis[yy][y];
        dd += fee[xx];
        if (xx != yy) dd += fee[yy];
        if (dd < dist[xx][yy]) {
          dist[xx][yy] = dd;
          q.push(PP(-dd, P(xx, yy)));
        }
      }

    for (int i = 0; i < netx[x][y].size(); ++i) {
```

```
        int xx = netx[x][y][i].X, yy = netx[x][y][i].Y;
        int dd = d + netd[x][y][i];
        if (dd < dist[xx][yy]) {
          dist[xx][yy] = dd;
          q.push(PP(-dd, P(xx, yy)));
        }
      }
    }

    cout << res << endl;
  }
}
```

Czech Technical University in Prague
ACM ICPC sponsored by IBM

Central Europe Regional Contest 2011



Unique Encryption Keys

`unique.c`, `unique.C`, `unique.java`

The security of many ciphers strongly depends on the fact that the keys are unique and never re-used. This may be vitally important, since a relatively strong cipher may be broken if the same key is used to encrypt several different messages.

In this problem, we will try to detect repeating (duplicate) usage of keys. Given a sequence of keys used to encrypt messages, your task is to determine what keys have been used repeatedly in some specified period.

Input Specification

The input contains several cipher descriptions. Each description starts with one line containing two integer numbers M and Q separated by a space. M ($1 \leq M \leq 1\,000\,000$) is the number of encrypted messages, Q is the number of queries ($0 \leq Q \leq 1\,000\,000$).

Each of the following M lines contains one number K_i ($0 \leq K_i \leq 2^{30}$) specifying the identifier of a key used to encrypt the i-th message. The next Q lines then contain one query each. Each query is specified by two integer numbers B_j and E_j, $1 \leq B_j \leq E_j \leq M$, giving the interval of messages we want to check.

There is one empty line after each description. The input is terminated by a line containing two zeros in place of the numbers M and Q.

Output Specification

For each query, print one line of output. The line should contain the string "`OK`" if all keys used to encrypt messages between B_j and E_j (inclusive) are mutually different (that means, they have different identifiers). If some of the keys have been used repeatedly, print one identifier of *any* such key.

Print one empty line after each cipher description.

Sample Input

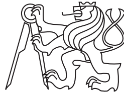
```
10 5
3
2
3
4
9
7
3
8
4
1
1 3
2 6
4 10
3 7
2 6

5 2
1
2
3
1
2
2 4
1 5

0 0
```

Output for Sample Input

```
3
OK
4
3
OK

OK
1
```

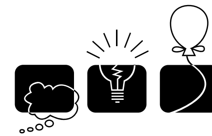
```
/*
ACM-ICPC Live Archive 5881. Unique Encryption Keys
Let R[i] be the largest integer such that the sequence V[i..R[i]] doesn't
contain repeated keys. Having R calculated, we can ask to each query of the
form [left, right] as follows: if R[left] <= right, then there aren't
repetitions in the interval [left, right], otherwise there is at least one
repetition, and one of the repeated keys is V[R[left] + 1].
The array R can be computed with a time complexity of O(N*log(N)). Look
at the code below for details about such implementation.
*/
#include <cstdio>
#include <iostream>
#include <map>
using namespace std;

typedef map<int, int> MAP;

int N, M, V[1000000], R[1000000];

int main() {
  scanf("%d%d", &N, &M);
  while (N > 0) {
    for (int i = 0; i < N; ++i) scanf("%d", &V[i]);

    MAP mp;
    R[N - 1] = N;
    mp[V[N - 1]] = N - 1;
    for (int i = N - 2; i >= 0; --i) {
      R[i] = R[i + 1];
      if (mp.count(V[i])) R[i] = min(R[i], mp[V[i]]);
      mp[V[i]] = i;
    }

    for (int i = 0; i < M; ++i) {
      int a, b;
      scanf("%d%d", &a, &b);
      --a; --b;
      if (R[a] <= b) printf("%d\n", V[R[a]]);
      else printf("OK\n");
    }
    printf("\n");

    scanf("%d%d", &N, &M);
  }
}
```

ACM-ICPC Live Archive 5888. Stack Machine Executor



Czech Technical University in Prague
ACM ICPC sponsored by IBM

Central Europe Regional Contest 2011



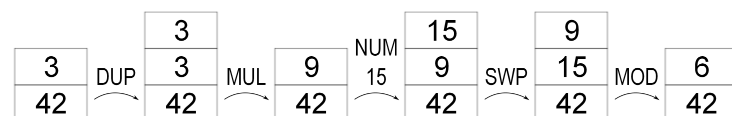
Stack Machine Executor

`execute.c`, `execute.C`, `execute.java`

Many ciphers can be computed much faster using various machines and automata. In this problem, we will focus on one particular type of machines called *stack machine*. Its name comes from the fact that the machine operates with the well-known data structure — *stack*. The later-stored values are on the top, older values at the bottom. Machine instructions typically manipulate the top of the stack only.

Our stack machine is relatively simple: It works with integer numbers only, it has no storage beside the stack (no registers etc.) and no special input or output devices. The set of instructions is as follows:

- **NUM *X***, where X is a non-negative integer number, $0 \le X \le 10^9$. The NUM instruction stores the number X on top of the stack. It is the only parametrized instruction.
- **POP**: removes the top number from the stack.
- **INV**: changes the sign of the top-most number. ($42 \to -42$)
- **DUP**: duplicates the top-most number on the stack.
- **SWP**: swaps (exchanges) the position of two top-most numbers.
- **ADD**: adds two numbers on the top of the stack.
- **SUB**: subtracts the top-most number from the "second one" (the one below).
- **MUL**: multiplies two numbers on the top of the stack.
- **DIV**: integer divison of two numbers on the top. The top-most number becomes divisor, the one below dividend. The quotient will be stored as the result.
- **MOD**: modulo operation. The operands are the same as for the division but the remainder is stored as the result.



All binary operations consider the top-most number to be the "right" operand, the second number the "left" one. All of them remove both operands from the stack and place the result on top in place of the original numbers.

If there are not enough numbers on the stack for an instruction (one or two), the execution of such an instruction will result into a program *failure*. A failure also occurs if a divisor becomes zero (for DIV or MOD) or if the result of any operation should be more than 10^9 in absolute value. This means that the machine only operates with numbers between $-1\,000\,000\,000$ and $1\,000\,000\,000$, inclusive.

To avoid ambiguities while working with negative divisors and remainders: If some operand of a division operation is negative, the absolute value of the result should always be computed with absolute values of operands, and the sign is determined as follows: The quotient is negative if (and only if) exactly one of the operands is negative. The remainder has the same sign as the dividend. Thus, $13 \text{ div } -4 = -3$, $-13 \text{ mod } 4 = -1$, $-13 \text{ mod } -4 = -1$, etc.

If a failure occurs for any reason, the machine stops the execution of the current program and no other instructions are evaluated in that program run.

Input Specification

The input contains description of several machines. Each machine is described by two parts: the program and the input section.

The program is given by a series of instructions, one per line. Every instruction is given by three uppercase letters and there must not be any other characters. The only exception is the NUM instruction, which has exactly one space after the three letters followed by a non-negative integer number between 0 and 10^9. The only allowed instructions are those defined above. Each program is terminated by a line containing the word "`END`" (and nothing else).

The input section starts with an integer N ($0 \leq N \leq 10\,000$), the number of program executions. The next N lines contain one number each, specifying an input value V_i, $0 \leq V_i \leq 10^9$. The program should be executed once for each of these values independently, every execution starting with the stack containing one number — the input value V_i.

There is one empty line at the and of each machine description. The last machine is followed by a line containing the word "`QUIT`". No program will contain more than 100 000 instructions and no program requires more than 1 000 numbers on the stack in any moment during its execution.

Output Specification

For each input value, print one line containing the output value for the corresponding execution, i.e., the one number that will be on the stack after the program executes with the initial stack containing only the input number.

If there is a program failure during the execution or if the stack size is incorrect at the end of the run (either empty or there are more numbers than one), print the word "`ERROR`" instead.

Print one empty line after each machine, including the last one.

Sample Input

```
DUP
MUL
NUM 2
ADD
END
3
1
10
50

NUM 1
NUM 1
ADD
END
2
42
43

NUM 600000000
ADD
END
3
0
600000000
1

QUIT
```

Output for Sample Input

```
3
102
2502

ERROR
ERROR

600000000
ERROR
600000001
```

```
/*
ACM-ICPC Live Archive 5888. Stack Machine Executor
In this problem, you have to make a program that simulates
the machine described in the statement. Just make every
operation to behave exactly as described, detecting and
handling the errors properly.
*/
#include <algorithm>
#include <iostream>
#include <stack>
using namespace std;

typedef long long ll;

const int NUM = 0, POP = 1, INV = 2, DUP = 3, SWP = 4, ADD = 5, SUB = 6, MUL = 7,
    DIV = 8, MOD = 9;
const int ERROR = 1100000000;

int N;
int instr[110000], param[110000];

bool check(ll x) {
  return -1000000000 <= x and x <= 1000000000;
}

ll divi(ll a, ll b) {
  if (a == 0) return 0;
  ll ta = abs(a);
  ll tb = abs(b);
  if (a*b < 0) return -(ta/tb);
  return ta/tb;
}

ll modu(ll a, ll b) {
  if (a == 0) return 0;
  ll ta = abs(a);
  ll tb = abs(b);
  ll r = (ta%tb + tb)%tb;
  if (a < 0) return -r;
  return r;
}

int simula(int input) {
  stack<int> st;
  st.push(input);
  for (int i = 0; i < N; ++i) {
    if (instr[i] == NUM) {
      st.push(param[i]);
    }
    else if (instr[i] == POP) {
      if (st.size() == 0) return ERROR;
      st.pop();
    }
    else if (instr[i] == INV) {
      if (st.size() == 0) return ERROR;
      int x = st.top();
      st.pop();
      st.push(-x);
    }
    else if (instr[i] == DUP) {
      if (st.size() == 0) return ERROR;
```

```
    int x = st.top();
    st.push(x);
  }
  else if (instr[i] == SWP) {
    if (st.size() < 2) return ERROR;
    int x = st.top();
    st.pop();
    int y = st.top();
    st.pop();
    st.push(x);
    st.push(y);
  }
  else if (instr[i] == ADD) {
    if (st.size() < 2) return ERROR;
    ll x = st.top();
    st.pop();
    ll y = st.top();
    st.pop();
    ll s = x + y;
    if (not check(s)) return ERROR;
    st.push(s);
  }
  else if (instr[i] == SUB) {
    if (st.size() < 2) return ERROR;
    ll x = st.top();
    st.pop();
    ll y = st.top();
    st.pop();
    ll s = y - x;
    if (not check(s)) return ERROR;
    st.push(s);
  }
  else if (instr[i] == MUL) {
    if (st.size() < 2) return ERROR;
    ll x = st.top();
    st.pop();
    ll y = st.top();
    st.pop();
    ll s = x*y;
    if (not check(s)) return ERROR;
    st.push(s);
  }
  else if (instr[i] == DIV) {
    if (st.size() < 2) return ERROR;
    ll x = st.top();
    st.pop();
    ll y = st.top();
    st.pop();
    if (x == 0) return ERROR;
    ll s = divi(y, x);
    if (not check(s)) return ERROR;
    st.push(s);
  }
  else if (instr[i] == MOD) {
    if (st.size() < 2) return ERROR;
    ll x = st.top();
    st.pop();
    ll y = st.top();
    st.pop();
    if (x == 0) return ERROR;
    ll s = modu(y, x);
```

```
      if (not check(s)) return ERROR;
      st.push(s);
    }
  }
  if (st.size() != 1) return ERROR;
  return st.top();
}

int main() {
  while (true) {
    N = 0;
    string s;
    while (cin >> s and s != "END") {
      if (s == "NUM") {
        cin >> param[N];
        instr[N++] = NUM;
      }
      else if (s == "POP") instr[N++] = POP;
      else if (s == "INV") instr[N++] = INV;
      else if (s == "DUP") instr[N++] = DUP;
      else if (s == "SWP") instr[N++] = SWP;
      else if (s == "ADD") instr[N++] = ADD;
      else if (s == "SUB") instr[N++] = SUB;
      else if (s == "MUL") instr[N++] = MUL;
      else if (s == "DIV") instr[N++] = DIV;
      else if (s == "MOD") instr[N++] = MOD;
      else return 0;
    }

    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
      int t;
      cin >> t;
      int r = simula(t);
      if (r == ERROR) cout << "ERROR" << endl;
      else cout << r << endl;
    }
    cout << endl;
  }
}
```

PROBLEM A — LIMIT 5 SECONDS

Good or Bad?

Description

Bikini Bottom has become inundated with tourists with super powers. Sponge Bob and Patrick are trying to figure out if a given character is good or bad, so they'll know whether to ask them to go jelly-fishing, or whether they should send Sandy, Mermaid Man, and Barnacle Boy after them.

SPONGE BOB: Wow, all these characters with super powers and we don't know whether they are good guys or bad guys.
PATRICK: Well, it's easy to tell. You just have to count up the number of g's and b's in their name. If they have more g's, they are good, if they have more b's, they are bad. Think about it, the greatest hero of them all, Algorithm Crunching Man is good since he has two g's and no b's.
SPONGE BOB: Oh, I get it. So Green Lantern is good and Boba Fett is bad!
PATRICK: Exactly! Uh, who's Boba Fett?
SPONGE BOB: Never mind. What about Superman?
PATRICK: Well he has the same number of g's as b's so he must be neutral.
SPONGE BOB: I see, no b's and no g's is the same number. Very clever Patrick! Well what about Batman? I thought he was good.
PATRICK: You clearly never saw *The Dark Knight*...
SPONGE BOB: Well what about Green Goblin? He's a baddy for sure and scary!
PATRICK: The Green Goblin is completely misunderstood. He's tormented by his past. Inside he's good and that's what counts. So the method works!
SPONGE BOB: Patrick, you are clearly on to something. But wait, are you saying that Plankton is neutral after all the terrible things he's tried to do to get the secret Crabby Patty formula?
PATRICK: Have any of his schemes ever worked?
SPONGE BOB: Hmmm, I guess not. Ultimately he's harmless and probably just needs a friend. So sure, neutral works for him.
PATRICK: Alright then, let's start taking names and figure this out.
SPONGE BOB: But Patrick, if we start counting all day, Squidward will probably get annoyed and play his clarinet and make us lose count.
PATRICK: Well, let's hire a human to do it for us on the computer. We'll pay them with Crabby Patties!
SPONGE BOB: Great idea Patrick. We're best friends forever!

Help Sponge Bob and Patrick figure out who is good and who is bad.

Input

The first line will contain an integer n ($n > 0$), specifying the number of names to process. Following this will be n names, one per line. Each name will have at least 1 character and no more than 25. Names will be composed of letters (upper or lower case) and spaces only. Spaces will only be used to separate multiple word names (e.g., there is a space between Green and Goblin).

Output

For each name read, display the name followed by a single space, followed by " is ", and then followed by either "GOOD", "A BADDY", or "NEUTRAL" based on the relation of b's to g's. Each result should be ended with a newline.

Sample Input

```
8
Algorithm Crunching Man
Green Lantern
Boba Fett
Superman
Batman
Green Goblin
Barney
Spider Pig
```

Sample Output

```
Algorithm Crunching Man is GOOD
Green Lantern is GOOD
Boba Fett is A BADDY
Superman is NEUTRAL
Batman is A BADDY
Green Goblin is GOOD
Barney is A BADDY
Spider Pig is GOOD
```

```
/*
ACM-ICPC Live Archive 5889. Good or Bad?
This is a trivial problem. Count the number of B's and
the number of G's, and compare them to print the proper
answer. Don't forget to count the lowercase and
uppercase letters as well.
*/
#include <iostream>
#include <string>
using namespace std;

int main() {
  int n;
  cin >> n;
  string line;
  getline(cin, line);
  for (int i = 0; i < n; ++i) {
    getline(cin, line);
    int m = line.size();
    int b = 0, g = 0;
    for (int i = 0; i < m; ++i) {
      if (line[i] == 'b' or line[i] == 'B') ++b;
      else if (line[i] == 'g' or line[i] == 'G') ++g;
    }
    cout << line << " is ";
    if (b < g) cout << "GOOD" << endl;
    else if (b > g) cout << "A BADDY" << endl;
    else cout << "NEUTRAL" << endl;
  }
}
```

Problem D — limit 30 seconds

Collateral Cleanup

Description

Remember the big fight where the Hulk and the Abomination threw each other through the buildings of Manhattan? Or the time when the Green Goblin smashed poor Spider-Man through a good half-dozen brick walls? Wow, they must have shattered those walls into a *million* pieces!!!

It's great that we have superheroes to bring the villains to justice, but have you ever wondered who gets to repair all the collateral damage when they're done? Well actually, as president of the Action Cleanup Management (ACM) corporation, your job is to do exactly that! After a big fight, you must take all the broken pieces of the walls and put them back together just as they were before the fierce battle began.

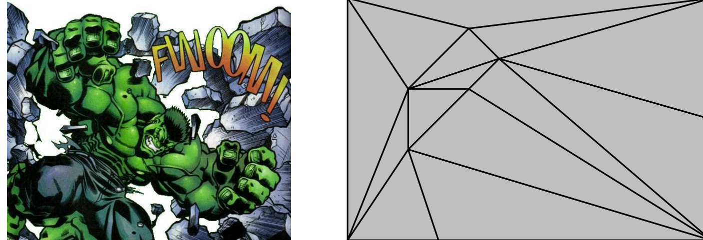
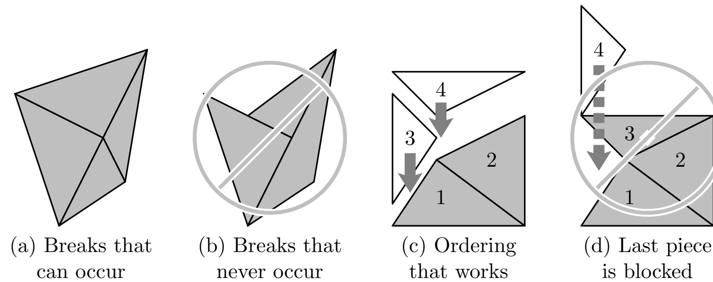


Figure 3: A wall broken into many pieces.

A wall is a perfectly rectangular region that shatters into perfectly triangular pieces when a villain is sent through it (see Figure 3). Through sophisticated visual analysis, you have ascertained where in the original structure every little piece came from. In essence, you have a blueprint that looks a lot like the picture above. Furthermore, you observe that wherever two broken pieces meet, they meet along the full length of the break (edge) that separates them, as shown in Figure 4a.



(a) Breaks that can occur | (b) Breaks that never occur | (c) Ordering that works | (d) Last piece is blocked

Figure 4: The good, the bad, and the ugly.

You have an assembly robot that can help you reconstruct a wall in place. However, the robot can only lower each piece, one at a time, straight down from the top. The robot cannot move a piece from side to side or rotate it in any way to get it where it needs to go. Thus, you must be careful regarding the order in which you tell the robot to reassemble the broken pieces, lest you inadvertently block a piece from being lowered into its proper place (see Figure 4d). Can you determine an ordering of the pieces for each wall that will allow you to fully reassemble it?

Input

An integer on the first line of the input file indicates the number of walls you must reassemble. The first line for each wall has an integer, n, indicating the number of triangular pieces the wall was broken into ($2 \le n \le 1{,}000{,}000$). Then, n lines of input follow, each describing a piece with six integers, $x_1\ y_1\ x_2\ y_2\ x_3\ y_3$, that correspond to the Cartesian (x, y) coordinates of the three corners of a triangle on the original wall from which the piece came. The first line describes piece 1, the next line piece 2, and so forth. The three points will always be given in a counterclockwise winding order and form a triangle of non-zero area. All coordinates lie between 0 and 10^9 inclusive, with the positive y direction being the "up" direction. The n pieces given will cover a rectangular region exactly, with no gaps or overlaps.

Output

For each wall, output on a single line the numbers of the pieces, separated by spaces, in an order that will allow your robot to reassemble the wall. If more than one correct solution exists, any ordering that will work is acceptable.

Sample Input

```
2
4
3 4 7 1 7 6
7 6 1 6 3 4
1 6 1 1 3 4
7 1 3 4 1 1
14
0 0 3 0 2 3
2 3 3 0 12 0
2 3 12 0 4 5
4 5 12 0 5 6
5 6 12 0 12 4
5 6 12 4 12 8
5 6 12 8 4 7
4 7 12 8 0 8
4 7 0 8 2 5
4 7 2 5 5 6
5 6 2 5 4 5
4 5 2 5 2 3
2 3 2 5 0 0
0 0 2 5 0 8
```

Sample Output

```
4 1 3 2
1 2 13 14 3 4 12 11 5 6 10 9 7 8
```

```
/*
ACM-ICPC Live Archive 5892. Collateral Cleanup
For every edge shared by two triangles, determine which one is on top (if any)
and build a directed graph where nodes correspond to triangles and there is an
edge from A and B if B is on top of A. Then perform a topological sort, which
will yield a valid ordering of the triangles.
*/
#include <algorithm>
#include <iostream>
#include <queue>
#include <vector>

#define FR first
#define SC second
#define X first
#define Y second

using namespace std;

typedef pair<int, int> PII;

int main() {
  int ncases;
  cin >> ncases;
  for (; ncases--;) {
    int n;
    cin >> n;
    vector<pair<pair<PII, PII>, int> > sides;
    vector<vector<PII> > triangles(n);
    for (int i = 0; i < n; ++i) {
      triangles[i] = vector<PII>(3);
      for (int j = 0; j < 3; ++j) {
        cin >> triangles[i][j].X >> triangles[i][j].Y;
      }
      for (int j = 0; j < 3; ++j) {
        pair<PII, PII> side(triangles[i][j], triangles[i][(j+1)%3]);
        if (side.FR > side.SC) {
          swap(side.FR, side.SC);
        }
        sides.push_back(pair<pair<PII, PII>, int>(side, -1));
      }
    }
    sort(sides.begin(), sides.end());
    sides.resize(unique(sides.begin(), sides.end())-sides.begin());
    vector<vector<int> > g(n);
    vector<int> in_deg(n, 0);
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < 3; ++j) {
        pair<PII, PII> side(triangles[i][j], triangles[i][(j+1)%3]);
        if (side.FR > side.SC) {
          swap(side.FR, side.SC);
        }
        int ind = int(lower_bound(sides.begin(), sides.end(), pair<pair<PII, PII>,
            int>(side, -1))-sides.begin());
        if (sides[ind].SC != -1) {
          int k = sides[ind].SC;
          // triangle i is on top of triangle k
          if (triangles[i][j].X < triangles[i][(j+1)%3].X) {
            g[k].push_back(i);
            ++in_deg[i];
          }
```

```
          // triangle k is on top of triangle i
          else if (triangles[i][j].X > triangles[i][(j+1)%3].X) {
            g[i].push_back(k);
            ++in_deg[k];
          }
        }
        else {
          sides[ind].SC = i;
        }
      }
    }
    queue<int> q;
    for (int i = 0; i < n; ++i) {
      if (in_deg[i] == 0) {
        q.push(i);
      }
    }
    for (bool first = true; !q.empty();) {
      int cur = q.front();
      q.pop();
      if (first) {
        first = false;
      }
      else {
        cout << " ";
      }
      cout << cur+1;
      for (int i = 0; i < int(g[cur].size()); ++i) {
        --in_deg[g[cur][i]];
        if (in_deg[g[cur][i]] == 0) {
          q.push(g[cur][i]);
        }
      }
    }
    cout << endl;
  }
}
```

PROBLEM F — LIMIT 5 SECONDS

Lightning Lessons

Description

Zeus wrung his hands nervously. "I've come to you because I agreed to duel Thor in the upcoming Godfest. You're good in a fight, Raiden; you've got to help me!"

Raiden, smiling thinly beneath the rim of his hat, replied, "What help could I provide a god as mighty as yourself? Your thunderbolts are the stuff of legends!" Zeus looked down and stammered, "I've... I've been lucky. I don't know how the thunderbolts actually work. Sometimes I turn my foe into a charred heap, but other times... weird stuff happens. If Apollo hadn't convinced the bards to keep my secret, I'd be a laughingstock."

Raiden raised his eyebrows and asked, "Weird stuff?" Zeus looked up and took a deep breath. "Sometimes it just fizzles out. Other times it rolls up and turns into a... a bunny." Raiden burst out laughing. "A bunny! That's some chi you've got there." As Zeus began to redden, Raiden held up his hand and said, "Don't worry, I'll help you out."

Raiden went on to explain. "A thunderbolt is a sequence of chi pivots, or 'zigs and zags' as the mortals call them. Each pivot has an integer amplitude—"

"Yes, I know that much.", Zeus interrupted. "But lightning is lively and unpredictable. The amplitudes go all random once the bolt hits!"

"Not all that flickers is flame. If you watch the bolt closely, you'll see it goes through 'cycles', and gets shorter by one pivot each cycle. When the bolt cycles, each successive pivot's amplitude is decreased by the amplitude of its predecessor from the end of the previous cycle, and the first pivot vanishes. If a bolt ever reaches a state of all zero amplitudes, it converges and zaps its target with power proportional to the number of preceding cycles. Your 'weird stuff' happens only when a bolt cycles down to a single non-zero amplitude. A positive amplitude just fizzles out into waste heat, but negative amplitudes produce odd low-entropy states. It's the latter you've seen hopping away in the midst of battle."

Help Zeus avoid embarrassment by writing a program that predicts how powerful a given bolt will be if it converges, or what will happen to it if it diverges.

Input

The first line of input contains a single positive integer N which denotes how many lightning bolts follow. Each bolt is specified by a line beginning with an integer M ($0 < M \leq 20$), followed by M space-delimited integers denoting the initial amplitudes of each successive pivot. No initial amplitude will have an absolute value larger than 1000.

Output

For each bolt that converges, output the letter "z" repeated P times, where P is the number of cycles encountered before the bolt converges, followed by the string "ap!" (the all-zero cycle does not count toward P).

For each bolt that fails to converge, output "*fizzle*" if the final amplitude was positive, "*bunny*" if it was negative.

Sample Input	Sample Output
4 2 1 1 5 1 3 6 10 15 5 1 2 4 8 16 2 1 0	zap! zzzap! *fizzle* *bunny*

```
/*
ACM-ICPC Live Archive 5894. Lightning Lessons
The main difficulty of this problem lies in understanding the statement.
The input is an array V of length N, and you have to simulate the
process described in the statement until the array becomes all zeroes or
only one element long, and write the proper answer acordingly.
The process that has to be applied at each step of the simulation to the
array V is as follows, where V' is the new resulting array:
V'[i] = V[i+1] - V[i] (0<=i<=N-2)
*/
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> Vi;

void next(Vi& v) {
  int n = v.size();
  for (int i = n - 1; i > 0; --i) v[i] -= v[i - 1];
  for (int i = 0; i + 1 < n; ++i) v[i] = v[i + 1];
  v.resize(n - 1);
}

bool zeros(const Vi& v) {
  int n = v.size();
  for (int i = 0; i < n; ++i)
    if (v[i]) return false;
  return true;
}

int main() {
  int n;
  cin >> n;
  for (int i = 0; i < n; ++i) {
    int m;
    cin >> m;
    Vi v(m);
    for (int j = 0; j < m; ++j) cin >> v[j];

    int res = 0;
    while (v.size() > 1 and not zeros(v)) {
      next(v);
      ++res;
    }

    if (zeros(v)) cout << string(res, 'z') << "ap!" << endl;
    else if (v[0] < 0) cout << "*bunny*" << endl;
    else cout << "*fizzle*" << endl;
  }
}
```

Problem H — limit 5 seconds

Speed Racer

Description

Speed Racer must go go go rescue Trixie at the top of Mount Domo! He must get there as quickly as possible, but his Mach 5 only holds a specific amount of fuel, and there is no way to refuel on the way. Luckily, he knows precisely how much fuel is consumed at a particular speed, taking into account air resistance, tire friction, and engine performance. For a given speed v in kilometers per hour, the amount of fuel consumed in liters per hour is

$$av^4 + bv^3 + cv^2 + dv$$

Assuming Speed travels at a constant speed, his tank holds t liters of fuel, and the top of Mount Domo is m kilometers away, how fast must he drive?

Input

The input will be one problem per line. Each line will contain six nonnegative floating point values representing a, b, c, d, m, and t, respectively. No input value will exceed 1000. The values of c, d, m, and t will be positive. There will always be a solution.

The output should be formatted as a decimal number with exactly two digits after the decimal point, and no leading zeros. The output value should be such that the Speed Racer will not run out of fuel (so truncate, rather than round, the final result). You are guaranteed that no final result will be within 10^{-6} of an integer multiple of 0.01.

Output

The required output will be a single floating point value representing the maximum speed in kilometers per hour that Speed Racer can travel to reach the top of Mount Domo without running out of gas.

Sample Input	Sample Output
0.000001 0.0001 0.029 0.2 12 100	134.41
2.8e-8 7.6e-6 0.0013 0.47 11.65 20.81	257.45
1.559e-7 1.8195e-5 0.0022233 0.31292 58.902 85.585	142.65

```
/*
ACM-ICPC Live Archive 5896. Speed Racer
The amount of fuel consumed during the whole trip in terms of the speed v is
f(v) = m*a*v^3 + m*b*v^2 + m*c*v + d*m
which is a strictly increasing function because a and b are nonnegative and
c, d, and m are positive. Therefore we can perform a binary search to find the
highest value of v such that f(v) <= t.
*/
#include <cmath>
#include <iostream>

using namespace std;

const double INF = 1e9;
const double EPS = 1e-9;

inline double used_fuel(double a, double b, double c, double d, double m, double v)
    {
  return m*a*v*v*v + m*b*v*v + m*c*v + d*m;
}

int main() {
  cout.setf(ios::fixed);
  cout.precision(2);
  for (double a, b, c, d, m, t; cin >> a >> b >> c >> d >> m >> t;) {
    double lo = 0.0, hi = INF;
    for (; lo+EPS < hi;) {
      double md = (lo+hi)/2.0;
      if (used_fuel(a, b, c, d, m, md) < t) {
        lo = md;
      }
      else {
        hi = md;
      }
    }
    cout << floor(100.0*lo)/100.0 << endl;
  }
}
```

Problem I — limit 10 seconds

The Status is Not Quo

Description

Dr. Horrible desperately wants to get into the Evil League of Evil but is having a difficult time proving his competence as the mastermind that he is. Bad Horse rules over the league with an iron hoof and is evaluating his application with extreme skepticism. Meanwhile, arch-nemesis Captain Hammer, hero of the people and corporate tool, is making life exceedingly complicated for our poor villain. But, everything is about to change. Dr. Horrible is all set to pull off a major heist; the wondeflonium needed to complete work on his freeze ray is being transported by courier van—candy from a baby. Sadly, it's not as easy as Dr. Horrible suspected. The device he created to control the van became a jumble of wires that needs to be untangled. He'd have Moist, his roommate, do it, but it's probably a bad idea to have Moist anywhere near circuitry (for obvious reasons). You better do it and do it fast, or else it's curtains for you—lacy, gently wafting curtains.

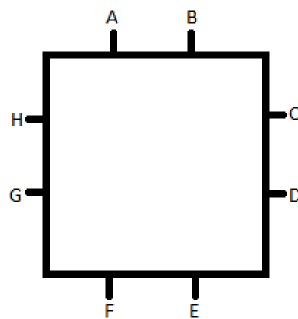


Figure 5: Connection point labels

You can help by figuring out a given wire's ending position based on its starting position for a variety of circuit boards. Here circuit boards are rectangular grids of squares, and each square has 8 connection points, two on each side. Squares also have any number of wires (between 0 and 4 inclusive) connecting one connection point of the square to another. A connection point for a square can only be used by one wire or not be used at all; there is no branching. By naming the square's connection points alphabetically from A to H (always capitalized) starting with the left connection on the top edge and traveling clockwise, you can describe each square as the collection of wires traveling from one named point to another. For example, if a square has a wire traveling from the left connection on the top edge to the bottom connection on the right edge, a wire from the left connection of the bottom edge to the right connection of the bottom edge, and a wire from the right connection of the top edge to the top connection of the left edge then you could describe the square as AD BH EF. For consistency, each wire pair is described alphabetically (BH instead of HB), and all the wire pairs for each square are listed in alphabetical order when describing a square.

Squares are aligned next to each other on all sides to make up the circuit board. For any given square, connection points A and B connect with F and E respectively with the square above it, and vice versa for the square below it. Connection points C and D connect with H and G respectively with the square to its right, and vice versa for the square to its left. If a square has a wire to any given connection point, its corresponding connection point in the adjacent square is guaranteed to continue the path of that wire from its own connection to another connection. There are no broken paths; all paths begin and end at the edge of the circuit board.

Input

Input consists of multiple puzzle sets. Each puzzle set is broken into two parts, a board description and a set of starting points. The board description begins with a single line containing two integers, h and w, both between 1 and 20 inclusive, separated by a space. These are the height and width, respectively, of the circuit board in squares. After this are n ($1 \leq n \leq h \cdot w$) lines containing square descriptions, which occur in no particular order. Each of these line describes one square and begins with a numeric designation for the square. The squares are numbered sequentially left to right, top to bottom, starting at 1; for example, the top right square is numbered w. After the number is the description of the wiring for the square as defined above. The number and all the wire descriptions are separated by single spaces. Not all squares may have a description, and a square will be described at most once per circuit board. Squares without lines have no wires connected to them.

The board description is separated from the set of starting points by a line containing only the number zero ("1"). The starting points are given on the next line, each consisting of a number and a letter together. The starting points are separated from each other by single spaces. The number of the starting point is the square and the letter is the connection point of the square to use. Only connection points on the outside of the circuit board will be given. Only connection points used by a wire will be given. Following this line is a single empty line before the start of the next puzzle set.

The end of the file is marked by two zeros separated by a space in place of the standard first line of a puzzle set.

Output

For each puzzle set, on one line output "Board" followed by a space followed by the number of the board (the number of the first puzzle starts at 1 and increments by 1 for every following puzzle) followed by a colon (":") character.

For each starting point in the set, output the ending point of that wire in the format of "{startpoint} is connected to {endpoint}" on one line. For example, if the given starting point was 1A and the end point was 9H, the output for that starting point would be "1A is connected to 9H". Capitalization matters. All wires are bidirectional, so for the same puzzle, if the starting point was 9H, the output would be "9H is connected to 1A". The letter portion of the start point and end point must be capitalized. There should be no other marks or punctuation.

The output for each puzzle set should be separated by a blank line.

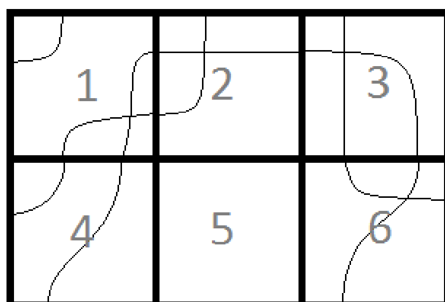


Figure 6: First sample case

Sample Input	Sample Output
2 3 4 AG BF 3 AF EH 6 AC BF 1 AH CE DF 2 AG CH 0 1A 2A 3A 4F 4 4 4 DH FG 15 BE FH 3 AD CE 13 BD 5 AD BF 6 BD FG 14 CG 16 AB 1 AE CH FG 7 BG CF DE 11 AG BC EH 9 AG DE 8 AH DG 10 AC DG 2 EH 12 DE FH 0 15F 1G 8D 3A 0 0	Board 1: 1A is connected to 1H 2A is connected to 4G 3A is connected to 6C 4F is connected to 6F Board 2: 15F is connected to 3A 1G is connected to 15E 8D is connected to 12D 3A is connected to 15F

```
/*
ACM-ICPC Live Archive 5897. The Status is Not Quo
The solution to this problem simply consists on simulating the connections in
the circuit board, and then for each query travel from the starting endpoint
through the connections until the opposite endpoint is reached.
*/
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

typedef vector<int> VI;
typedef vector<VI> V2I;
typedef vector<V2I> V3I;

const int DY[] = {-1, 0, 1, 0};
const int DX[] = {0, 1, 0, -1};

inline bool inside(int y, int x, int h, int w) {
  return y >= 0 && y < h && x >= 0 && x < w;
}

inline int opposite(int con) {
  return (con+4+1-2*(con%2))%8;
}

int main() {
  for (int ca = 1, h, w; cin >> h >> w && (h > 0 || w > 0); ++ca) {
    V3I c(h, V2I(w, VI(8, -1)));
    for (int sq; cin >> sq && sq > 0;) {
      int y = (sq-1)/w;
      int x = (sq-1)%w;
      string s;
      getline(cin, s);
      istringstream iss(s);
      for (string p; iss >> p;) {
        c[y][x][int(p[0]-'A')] = int(p[1]-'A');
        c[y][x][int(p[1]-'A')] = int(p[0]-'A');
      }
    }
    string s;
    getline(cin, s);
    getline(cin, s);
    istringstream iss(s);
    if (ca > 1) {
      cout << endl;
    }
    cout << "Board " << ca << ":" << endl;
    for (int sq; iss >> sq;) {
      int y = (sq-1)/w;
      int x = (sq-1)%w;
      char letter;
      iss >> letter;
      int con = int(letter-'A');
      do {
        int nxt_y = y+DY[c[y][x][con]/2];
        int nxt_x = x+DX[c[y][x][con]/2];
        con = opposite(c[y][x][con]);
        y = nxt_y;
```

```
        x = nxt_x;
      } while (inside(y, x, h, w));

      int nxt_y = y+DY[con/2];
      int nxt_x = x+DX[con/2];
      con = opposite(con);
      y = nxt_y;
      x = nxt_x;
      cout << sq << letter << " is connected to " << y*w+x+1 << char('A'+con) <<
          endl;
    }
  }
}
```

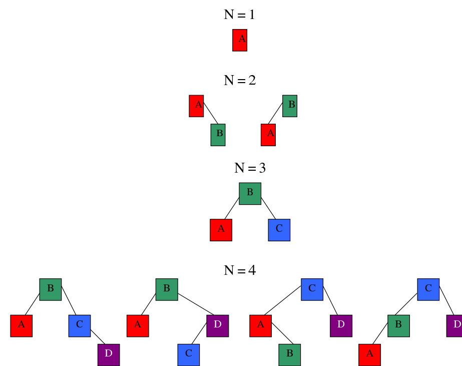
Problem K — limit 5 seconds

Tree Count

Description

Our superhero Carry Adder has uncovered the secret method that the evil Head Crash uses to generate the entrance code to his fortress in Oakland. The code is always the number of distinct binary search trees with some number of nodes, that have a specific property. To keep this code short, he only uses the least significant nine digits.

The property is that, for each node, the height of the right subtree of that node is within one of the height of the left subtree of that node. Here, the height of a subtree is the length of the longest path from the root of that subtree to any leaf of that subtree. A subtree with a single node has a height of 0, and by convention, a subtree containing no nodes is considered to have a height of -1.



Input

Input will be formatted as follows. Each test case will occur on its own line. Each line will contain only a single integer, N, the number of nodes. The value of N will be between 1 and 1427, inclusive.

Output

Your output is one line per test case, containing only the nine-digit code (note that you must print leading zeros).

Sample Input	Sample Output
1 3 6 21	000000001 000000001 000000004 000036900

```
/*
ACM-ICPC Live Archive 5899. Tree Count
In order to solve this problem it is useful to count the number of binary trees
of N nodes and height H that satisfy the property described in the problem
statement, because it can be done fairly easily using Dynamic Programming.
The number of binary trees of N nodes satisfying the property is simply the sum
for all possible heights.
*/
#include <iomanip>
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;

const int MOD = 1000000000;

vector<int> mem_mn_nodes;
vector<vector<int> > mem_trees;

inline int mx_nodes(int h) {
  return (1<<(h+1))-1;
}

int mn_nodes(int h) {
  if (h == -1) {
    return 0;
  }
  if (h == 0) {
    return 1;
  }
  int &ret = mem_mn_nodes[h];
  if (ret != -1) {
    return ret;
  }
  return ret = 1+mn_nodes(h-1)+mn_nodes(h-2);
}

int trees(int h, int n) {
  if (n < mn_nodes(h) || n > mx_nodes(h)) {
    return 0;
  }
  if (h == -1) {
    if (n == 0) {
      return 1;
    }
    return 0;
  }
  if (h == 0) {
    if (n == 1) {
      return 1;
    }
    return 0;
  }
  int &ret = mem_trees[h][n];
  if (ret != -1) {
    return ret;
  }

  ret = 0;
```

```
  // (h-1, h-1)
  for (int left = max(mn_nodes(h-1), n-1-mx_nodes(h-1)); left <= n-1; ++left) {
    if (left > mx_nodes(h-1)) {
      break;
    }
    int right = n-1-left;
    if (right < mn_nodes(h-1)) {
      break;
    }
    ret += int((ll(trees(h-1, left))*ll(trees(h-1, right)))%ll(MOD));
    ret %= MOD;
  }

  // (h-1, h-2) U (h-2, h-1)
  for (int left = max(mn_nodes(h-1), n-1-mx_nodes(h-2)); left <= n-1; ++left) {
    if (left > mx_nodes(h-1)) {
      break;
    }
    int right = n-1-left;
    if (right < mn_nodes(h-2)) {
      break;
    }
    ret += int((2*ll(trees(h-1, left))*ll(trees(h-2, right)))%ll(MOD));
    ret %= MOD;
  }

  return ret;
}

int main() {
  mem_mn_nodes = vector<int>(30, -1);
  mem_trees = vector<vector<int> >(14, vector<int>(1500, -1));
  for (int n; cin >> n;) {
    int ans = 0;
    for (int h = 0; h < 14; ++h) { // N < 1500  ==>  H < 14
      ans += trees(h, n);
      ans %= MOD;
    }
    cout << setw(9) << setfill('0') << ans << endl;
  }
}
```

C Movie collection

Mr. K. I. has a very big movie collection. He has organized his collection in a big stack. Whenever he wants to watch one of the movies, he locates the movie in this stack and removes it carefully, ensuring that the stack doesn't fall over. After he finishes watching the movie, he places it at the top of the stack.

Since the stack of movies is so big, he needs to keep track of the position of each movie. It is sufficient to know for each movie how many movies are placed above it, since, with this information, its position in the stack can be calculated. Each movie is identified by a number printed on the movie box.

Your task is to implement a program which will keep track of the position of each movie. In particular, each time Mr. K. I. removes a movie box from the stack, your program should print the number of movies that were placed above it before it was removed.

Input

On the first line a positive integer: the number of test cases, at most 100. After that per test case:

- one line with two integers n and m ($1 \leq n, m \leq 100\,000$): the number of movies in the stack and the number of locate requests.
- one line with m integers $a_1, \ldots, a_m$ ($1 \leq a_i \leq n$) representing the identification numbers of movies that Mr. K. I. wants to watch.

For simplicity, assume the initial stack contains the movies with identification numbers $1, 2, \ldots, n$ in increasing order, where the movie box with label 1 is the top-most box.

Output

Per test case:

- one line with m integers, where the i-th integer gives the number of movie boxes above the box with label a_i, immediately before this box is removed from the stack.

Note that after each locate request a_i, the movie box with label a_i is placed at the top of the stack.

Sample in- and output

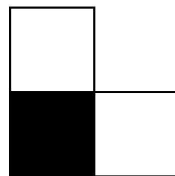
Input	Output
2 3 3 3 1 1 5 3 4 4 5	2 1 0 3 0 4

```
/*
ACM-ICPC Live Archive 5902. Movie collection
Think of having an array of length n+m, and the books located
initially on the first n positions of the array. Actually, this
arrray contains a zero or one in each position, depending on
whether there's a book at that position or not. Whenever you
must take a book from "the stack", take the book from its
current position in the array, to the position immediately at
the right of the rightmost book.
Create and update a different array whith the position of each
book in the array of books.
How to count the number of books below a given position? Instead
of having an array to store whether a position is occupied by a
book, store this information in a more proper data structure,
for example an Interval Tree, which allows us to modify a single
position in O(log(n)) and query the sum of an interval in
O(log(n)) too.
*/
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> Vi;

int N;
Vi tree;

void update(int n, int e, int d, int x, int v) {
  if (e == d) {
    tree[n] = v;
    return;
  }
  int m = (e + d)/2;
  if (x <= m) update(2*n, e, m, x, v);
  else update(2*n + 1, m + 1, d, x, v);
  tree[n] = tree[2*n] + tree[2*n + 1];
}

int query(int n, int e, int d, int x) {
  if (d <= x) return tree[n];
  if (x < e) return 0;
  int m = (e + d)/2;
  int a = query(2*n, e, m, x);
  int b = query(2*n + 1, m + 1, d, x);
  return a + b;
}

int main() {
  int tcas;
  cin >> tcas;
  for (int cas = 1; cas <= tcas; ++cas) {
    int n, m;
    cin >> n >> m;

    Vi pos(n);
    for (int i = 0; i < n; ++i)
      pos[i] = n - i - 1;

    N = n + m;
    tree = Vi(4*N, 0);
    for (int i = 0; i < n; ++i)
```

```
      update(1, 0, N - 1, i, 1);

    int p = n;
    for (int i = 0; i < m; ++i) {
      int t;
      cin >> t;
      --t;

      if (i) cout << " ";
      cout << n - query(1, 0, N - 1, pos[t]);

      update(1, 0, N - 1, pos[t], 0);
      update(1, 0, N - 1, p, 1);
      pos[t] = p++;
    }
    cout << endl;
  }
}
```

D Piece it together

Tom has developed a special kind of puzzle: it involves a whole bunch of identical puzzle pieces. The pieces have the shape of three adjoint squares in an L-shape. The corner square is black, the two adjacent squares are white.



A puzzle piece

The puzzler is given a pattern of black and white squares in a rectangular grid. The challenge is to create that pattern using these pieces. The pieces can be rotated, but must not overlap.

Tom has already designed a few nice patterns, but he needs to find out if they can be constructed with the pieces at all. Rather than trying to test this for each pattern by hand, he wants to write a computer program to determine this for him. Can you help him?

Input

On the first line a positive integer: the number of test cases, at most 100. After that per test case:

- one line with two integers n and m ($1 \leq n, m \leq 500$): the height and width of the grid containing the pattern, respectively.
- n lines, each containing m characters, denoting the grid. Each character is '`B`', '`W`', or '`.`', indicating a black, white or empty square respectively.

The grid contains at least one black or white square.

Output

Per test case:

- one line with either "`YES`" or "`NO`", indicating whether or not it is possible to construct the pattern with the puzzle pieces. You may assume that there is an infinite supply of pieces.

Sample in- and output

Input	Output
2 3 4 BWW. WWBW ..WB 3 3 W.. BW. WBW	YES NO

```
/*
ACM-ICPC Live Archive 5903. Piece it together
This problem can be solved as a 2-SAT. The variables are the edges
between adjacent cells (that is, R*(C-1) edges that join two horizontally
adjacent cells, and C*(R-1) edges vertically). A variable being true means
that the two adjacent cells which represents are contained in the same L
piece, while being false means they are not in the same piece.
The restrictions that need to be imposed are:
(1) Each white cell must have exactly one adjacent edge/variable with
its value set to true.
(2) Each black cell must have exactly one horizontally adjacent edge/variable
and exactly one vertically set to true.
*/
#include <iostream>
#include <string>
#include <vector>
using namespace std;

#define PB push_back

typedef vector<string> Vs;
typedef vector<int> Vi;
typedef vector<Vi> Mi;

int R, C;
Vs mapa;

int N;
Mi net, netr;

int T;
Vi vist, ord, comp;

inline int hor(int r, int c) { return r*(C - 1) + c; }
inline int ver(int r, int c) { return R*(C - 1) + c*(R - 1) + r; }

inline void aresta(int a, int b) {
  net[a].PB(b);
  netr[b].PB(a);
}

void dfsr(int n) {
  if (vist[n]) return;
  vist[n] = 1;
  for (int i = 0; i < netr[n].size(); ++i) dfsr(netr[n][i]);
  ord[--T] = n;
}

void dfs(int n, int m) {
  if (vist[n]) return;
  vist[n] = 1;
  comp[n] = m;
  for (int i = 0; i < net[n].size(); ++i) dfs(net[n][i], m);
}

int main() {
  int tcas;
  cin >> tcas;
  for (int cas = 1; cas <= tcas; ++cas) {
    cin >> R >> C;
    mapa = Vs(R);
```

```
  for (int i = 0; i < R; ++i) cin >> mapa[i];

  int bb = 0, ww = 0;
  for (int i = 0; i < R; ++i)
    for (int j = 0; j < C; ++j) {
      if (mapa[i][j] == 'B') ++bb;
      else if (mapa[i][j] == 'W') ++ww;
    }
  if (2*bb != ww) {
    cout << "NO" << endl;
    continue;
  }

  N = R*(C - 1) + C*(R - 1);
  net = netr = Mi(2*N);

  bool ok = true;
  for (int i = 0; i < R; ++i)
    for (int j = 0; j < C; ++j) {
      if (mapa[i][j] != 'B') continue;

      {
        bool a = i > 0 and mapa[i - 1][j] == 'W';
        bool b = i + 1 < R and mapa[i + 1][j] == 'W';
        if (a and b) {
          int ta = ver(i - 1, j);
          int tb = ver(i, j);
          aresta(N + ta, tb);
          aresta(ta, N + tb);
          aresta(N + tb, ta);
          aresta(tb, N + ta);
        }
        else if (a) {
          int ta = ver(i - 1, j);
          aresta(N + ta, ta);
        }
        else if (b) {
          int tb = ver(i, j);
          aresta(N + tb, tb);
        }
        else ok = false;
      }

      {
        bool a = j > 0 and mapa[i][j - 1] == 'W';
        bool b = j + 1 < C and mapa[i][j + 1] == 'W';
        if (a and b) {
          int ta = hor(i, j - 1);
          int tb = hor(i, j);
          aresta(N + ta, tb);
          aresta(ta, N + tb);
          aresta(N + tb, ta);
          aresta(tb, N + ta);
        }
        else if (a) {
          int ta = hor(i, j - 1);
          aresta(N + ta, ta);
        }
        else if (b) {
          int tb = hor(i, j);
          aresta(N + tb, tb);
```

```
          }
          else ok = false;
        }
      }

    for (int i = 0; i < R; ++i)
      for (int j = 0; j < C; ++j) {
        if (mapa[i][j] != 'W') continue;
        Vi v;
        if (i > 0 and mapa[i - 1][j] == 'B') v.PB(ver(i - 1, j));
        if (i + 1 < R and mapa[i + 1][j] == 'B') v.PB(ver(i, j));
        if (j > 0 and mapa[i][j - 1] == 'B') v.PB(hor(i, j - 1));
        if (j + 1 < C and mapa[i][j + 1] == 'B') v.PB(hor(i, j));

        int sz = v.size();
        if (sz == 0) ok = false;

        for (int x = 0; x < sz; ++x)
          for (int y = 0; y < sz; ++y) {
            if (x == y) continue;
            aresta(v[x], N + v[y]);
          }
      }

    if (not ok) {
      cout << "NO" << endl;
      continue;
    }

    T = 2*N;
    ord = Vi(2*N, 0);
    vist = Vi(2*N, 0);
    for (int i = 0; i < 2*N; ++i) dfsr(i);

    vist = Vi(2*N, 0);
    comp = Vi(2*N);
    int comps = 0;
    for (int i = 0; i < 2*N; ++i) {
      int x = ord[i];
      if (vist[x] == 0) dfs(x, comps++);
    }

    for (int i = 0; i < N; ++i)
      if (comp[i] == comp[N + i]) {
        ok = false;
        break;
      }

    if (ok) cout << "YES" << endl;
    else cout << "NO" << endl;
  }
}
```

E Please, go first

You are currently on a skiing trip with a group of friends. In general, it is going well: you enjoy the skiing during the day and, of course, the après-skiing during the night. However, there is one nuisance: the skiing lift. As always, it is too small, and can only serve one person every 5 seconds. To make matters worse, you and your friends generally don't arrive simultaneously at the lift, which means that you spend time waiting at the bottom of the mountain for the lift and at the top again for your friends.

The waiting at the top is especially inefficient. In fact, you realize that if your friends haven't arrived yet, you might as well let other people pass you in the queue. For you, it makes no difference, since otherwise you'd be waiting at the top. On the other hand, your actions might save them time if their friends have already arrived and are currently waiting for them at the top.

You are wondering how much time would be saved if everybody adopts this nice attitude. You have carefully observed the queue for a while and noticed which persons form groups of friends. Suppose someone lets another pass if doing this doesn't change his own total waiting time, but saves time for the other person. Do this over and over again until it can't be done anymore. How much time will this save, in total?

Input

On the first line a positive integer: the number of test cases, at most 100. After that per test case:

- one line with an integer n ($1 \leq n \leq 25\,000$): the number of people in the line for the lift.
- one line with n alphanumeric characters (uppercase and lowercase letters and numbers): the queue. The first person in this line corresponds to the person at the head of the queue. Equal characters correspond to persons from the same group of friends.

Output

Per test case:

- one line with an integer: the time saved, in seconds.

Sample in- and output

Input	Output
2 6 AABABB 10 Ab9AAb2bC2	15 45

```
/*
ACM-ICPC Live Archive 5904. Please, go first
The following arrangement is optimal:
Pick all the people belonging to the last person in the queue's team
and put them all together at the end of the queue. Then, apply the
same algorithm to the rest of the queue, and so on.
Knowing that, it shouldn't be difficult to calculate the total amount
of saved time.
*/
#include <algorithm>
#include <iostream>
#include <map>
#include <string>
#include <utility>
#include <vector>
using namespace std;

#define X first
#define Y second

typedef long long ll;
typedef pair<int, int> P;
typedef map<char, int> MAP;
typedef vector<int> Vi;
typedef vector<P> Vp;

int main() {
  int tcas;
  cin >> tcas;
  for (int cas = 1; cas <= tcas; ++cas) {
    int n;
    string s;
    cin >> n >> s;

    MAP mp;
    int m = 0;
    for (int i = 0; i < n; ++i)
      if (mp.count(s[i]) == 0) mp[s[i]] = m++;

    Vi v(n);
    for (int i = 0; i < n; ++i) v[i] = mp[s[i]];

    Vi p(m), q(m, 0);
    for (int i = 0; i < n; ++i) {
      p[v[i]] = n - i - 1;
      ++q[v[i]];
    }

    Vp ord(m);
    for (int i = 0; i < m; ++i) ord[i] = P(p[i], i);
    sort(ord.begin(), ord.end());

    ll res = 0;

    int t = 0;
    for (int i = 0; i < m; ++i) {
      int e = ord[i].Y;
      res += ll(q[e])*(t - p[e]);
      t += q[e];
    }
```

```
    cout << 5*res << endl;
  }
}
```

G Smoking gun

Andy: ”Billy the Kid fired first!”

Larry: ”No, I'm sure I heard the first shot coming from John!”

The arguments went back and forth during the trial after the big shoot-down, somewhere in the old wild west. Miraculously, everybody had survived (although there were serious injuries), but nobody could agree about the exact sequence of shots that had been fired. It was known that everybody had fired at most one shot, but everything had happened very fast. Determining the precise order of the shots was important for assigning guilt and penalties.

But then the sheriff, Willy the Wise, interrupted: ”Look, I've got a satellite image from the time of the shooting, showing exactly where everybody was located. As it turns out, Larry was located much closer to John than to Billy the Kid, while Andy was located just slightly closer to John than to Billy the Kid. Thus, because sound travels with a finite speed of 340 meters per second, Larry may have heard John's shot first, even if Billy the Kid fired first. But, although Andy was closer to John than to Billy the Kid, he heard Billy the Kid's shot first – so we know for a fact that Billy the Kid was the one who fired first!

Your task is to write a program to deduce the exact sequence of shots fired in situations like the above.

Input

On the first line a positive integer: the number of test cases, at most 100. After that per test case:

- one line with two integers n ($2 \leq n \leq 100$) and m ($1 \leq m \leq 1\,000$): the number of people involved and the number of observations.
- n lines with a string S, consisting of up to 20 lower and upper case letters, and two integers x and y ($0 \leq x, y \leq 1\,000\,000$): the unique identifier for a person and his/her position in Cartesian coordinates, in metres from the origin.
- m lines of the form “`S1 heard S2 firing before S3`”, where $S1$, $S2$ and $S3$ are identifiers among the people involved, and $S2 \neq S3$.

If a person was never mentioned as $S2$ or $S3$, then it can be assumed that this person never fired, and only acted as a witness. No two persons are located in the same position.

The test cases are constructed so that an error of less than 10^{-7} in one distance calculation will not affect the output.

Output

Per test case:

- one line with the ordering of the shooters that is compatible with all of the observations, formatted as the identifiers separated by single spaces.

If multiple distinct orderings are possible, output “`UNKNOWN`” instead. If no ordering is compatible with the observations, output “`IMPOSSIBLE`” instead.

Sample in- and output

<table>
<tr><th>Input</th><th>Output</th></tr>
<tr><td>

```
3
4 2
BillyTheKid 0 0
Andy 10 0
John 19 0
Larry 20 0
Andy heard BillyTheKid firing before John
Larry heard John firing before BillyTheKid
2 2
Andy 0 0
Beate 0 1
Andy heard Beate firing before Andy
Beate heard Andy firing before Beate
3 1
Andy 0 0
Beate 0 1
Charles 1 3
Beate heard Andy firing before Charles
```

</td><td>

```
BillyTheKid John
IMPOSSIBLE
UNKNOWN
```

</td></tr>
</table>

```
/*
ACM-ICPC Live Archive 5906. Smoking gun
Each statement of the form 'S1 heard S2 firing before S3' is telling us that
fire_time(S2)+travel_time(S2, S1) < fire_time(S3)+travel_time(S3, S1)
which can be rewritten as
travel_time(S2, S1)-travel_time(S3, S1) < fire_time(S3)-fire_time(S2)
and also as
fire_time(S2)-fire_time(S3) < travel_time(S3, S1)-travel_time(S2, S1)
therefore giving us an upper bound on one difference of firing times and a
lower bound in another difference of firing times. After all this information
has been collected we can perform a transitive closure, using Floyd-Warshall
algorithm, to get the highest possible lower bound and the lowest possible
upper bound for every difference of firing times.

If for some difference of firing times there is a contradiction (the upper bound
is lower than the lower bound), there is no possible order.
If for some difference of firing times the bound do not determine their order
(the lower bound is negative and the upper bound is positive), there are
multiple possible orderings.
If none of the previous two situations occur, the information we have computed
determines a unique ordering between the firings.
*/
#include <algorithm>
#include <cmath>
#include <iostream>
#include <map>
#include <set>
#include <string>
#include <vector>

#define FR first
#define SC second

using namespace std;

typedef pair<double, double> PDD;

const double INF = 1e9;
const double EPS = 1e-9;

double dist(double x1, double y1, double x2, double y2) {
  return sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2));
}

int main() {
  int ncases;
  cin >> ncases;
  for (; ncases--;) {
    int n, m;
    cin >> n >> m;
    vector<string> names(n);
    map<string, int> id;
    vector<double> vx(n), vy(n);
    for (int i = 0; i < n; ++i) {
      cin >> names[i] >> vx[i] >> vy[i];
      id[names[i]] = i;
    }
    vector<vector<double> > lb(n, vector<double>(n, -INF)), ub(n, vector<double>(n,
       INF));
    for (int i = 0; i < n; ++i) {
      lb[i][i] = ub[i][i] = 0.0;
```

```
  }
  set<int> sfire;
  for (int i = 0; i < m; ++i) {
    string s1, s2, s3, spam;
    cin >> s1 >> spam >> s2 >> spam >> spam >> s3;
    int id1 = id[s1], id2 = id[s2], id3 = id[s3];
    lb[id2][id3] = max(lb[id2][id3], dist(vx[id1], vy[id1], vx[id2], vy[id2])-dist
        (vx[id1], vy[id1], vx[id3], vy[id3]));
    ub[id3][id2] = min(ub[id3][id2], dist(vx[id1], vy[id1], vx[id3], vy[id3])-dist
        (vx[id1], vy[id1], vx[id2], vy[id2]));
    sfire.insert(id2);
    sfire.insert(id3);
  }
  vector<int> fire(sfire.begin(), sfire.end());

  for (int k = 0; k < n; ++k) {
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        lb[i][j] = max(lb[i][j], lb[i][k]+lb[k][j]);
        ub[i][j] = min(ub[i][j], ub[i][k]+ub[k][j]);
      }
    }
  }

  bool possible = true;
  for (int i = 0; i < int(fire.size()) && possible; ++i) {
    for (int j = 0; j < int(fire.size()); ++j) if (i != j) {
      if (lb[fire[i]][fire[j]] > EPS && lb[fire[j]][fire[i]] > EPS) {
        possible = false;
      }
    }
  }
  if (!possible) {
    cout << "IMPOSSIBLE" << endl;
    continue;
  }

  bool unknown = false;
  for (int i = 0; i < int(fire.size()) && !unknown; ++i) {
    for (int j = 0; j < int(fire.size()); ++j) if (i != j) {
      if (lb[fire[i]][fire[j]] < -EPS && ub[fire[i]][fire[j]] > EPS) {
        unknown = true;
      }
    }
  }
  if (unknown) {
    cout << "UNKNOWN" << endl;
    continue;
  }

  vector<pair<double, int> > order;
  for (int i = 0; i < int(fire.size()); ++i) {
    order.push_back(pair<double, int>(lb[fire[0]][fire[i]], fire[i]));
  }
  sort(order.begin(), order.end());
  for (int i = 0; i < int(order.size()); ++i) {
    if (i > 0) {
      cout << " ";
    }
    cout << names[order[i].SC];
  }
```

```
    cout << endl;
  }
}
```

4 Timus Online Judge

El Timus Online Judge és un jutge online creat per estudiants de la universitat russa USU (Ural State University). Conté una col·lecció de més de 800 problemes procedents de diverses competicions fetes a Rússia, com són les subregionals del NEERC (North Eastern European Regional Contest) de l'ICPC, i competicions locals d'universitats russes, entre altres.

En algunes ocasions s'hi fan concursos online oberts a tothom, amb regles similars a les de l'ICPC. En aquests concursos no es reparteixen premis ni cap tipus d'incentiu, estan pensats per servir de pràctica. Molts d'aquests concursos online són rèpliques de concursos presencials que es realitzen simultàniament.

Aquest jutge accepta solucions en 5 llenguatges de programació diferents: C, C++, C#, Java i Pascal.

A la pàgina web del Timus Online Judge hi ha un fòrum en el qual els usuaris comparteixen idees sobre els problemes, i en alguns casos els programadors experts que han resolt un determinat problema proposen jocs de prova per afegir al jutge, amb l'objectiu de fer-lo més exhaustiu.

A continuació presentem els enunciats i solucions dels 103 problemes que vam resoldre del Timus Online Judge.

Timus 1100. Final Standings

1100. Final Standings

Time Limit: 1.0 second
Memory Limit: 16 MB

Old contest software uses bubble sort for generating final standings. But now, there are too many teams and that software works too slow. You are asked to write a program, which generates exactly the same final standings as old software, but fast.

Input

The first line of input contains only integer $1 < N \le 150000$ — number of teams. Each of the next N lines contains two integers $1 \le ID \le 10^7$ and $0 \le M \le 100$. ID — unique number of team, M — number of solved problems.

Output

Output should contain N lines with two integers ID and M on each. Lines should be sorted by M in descending order using bubble sort (or analog).

Sample

input	output
8 1 2 16 3 11 2 20 3 3 5 26 4 7 1 22 4	3 5 26 4 22 4 16 3 20 3 1 2 11 2 7 1

Hint

Bubble sort works following way:

```
while (exists A[i] and A[i+1] such as A[i] < A[i+1]) do
   Swap(A[i], A[i+1]);
```

Problem Author: Pavel Atnashev
Problem Source: Tetrahedron Team Contest May 2001

```
/*
Timus 1100. Final Standings
Sort the pairs making sure that the relative order of the pairs with the same
value of M in the resulting list is the same as in the original list. The use
of the STL's stable_sort method simplifies the code.
*/
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
using namespace std;

#define X first
#define Y second

typedef long long ll;
typedef pair<ll, ll> P;
typedef vector<P> Vp;

bool sortf(P a, P b) {
  return a.Y > b.Y;
}

int main() {
  int n;
  cin >> n;

  Vp v(n);
  for (int i = 0; i < n; ++i) cin >> v[i].X >> v[i].Y;

  stable_sort(v.begin(), v.end(), sortf);
  for (int i = 0; i < n; ++i) cout << v[i].X << " " << v[i].Y << endl;
}
```

1101. Robot in the Field

Time Limit: 1.0 second
Memory Limit: 16 MB

There is a field $[-N..N]\times[-N..N]$. At initial moment, robot stands at point (0, 0). It starts moving in (1, 0) direction. Robot moves according to a program. Program is a correct boolean expression. It contains operators NOT, AND, OR (NOT has highest priority, OR - lowest), brackets '(', ')', constants 'TRUE' and 'FALSE', and registers 'A', ..., 'Z'. Initially, all robot's registers are FALSE. Robot moves forward until it reaches a fork. Then, robot evaluate the expression and turns right if it is TRUE and turns left if it is FALSE. Besides, there are some points in the field, standing on which makes one of robot's registers to invert. You are asked to print robot's route until it falls out of the field.

Input

First line contains boolean expression. The length of expression ≤ 250. Second line contains three integers $1 \le N \le 100$, $0 \le M \le 100$, $0 \le K \le 100$. M — number of forks, K — number of register inverting points. Then follows M lines, each of them contains two integers X, Y — coordinates of forks. Then follows K lines, each of them contains two integers X, Y and character C — coordinates of register inverting point and name of register, which inverts. You may assume, that there is no fork at point (0, 0). You may assume, that no two objects (forks or register inverting points) coincide. You may assume, that after some moves robot falls out of the field.

Output

You should print robot's route to output, every pair of coordinates in separate line.

Sample

input	output
NOT((A OR NOT B) AND (A OR B)) OR NOT (A AND NOT B OR TRUE) 1 5 2 1 0 1 1 1 -1 -1 -1 -1 1 0 1 A -1 0 D	0 0 1 0 1 -1 0 -1 -1 -1 -1 0 -1 1 0 1 1 1

Problem Author: Pavel Atnashev
Problem Source: Tetrahedron Team Contest May 2001

```
/*
Timus 1101. Robot in the Field
We simulate the movement of the robot, keeping track of the value of each
register and at every fork we parse and evaluate the expression using a
recursive function.
*/
#include <iostream>
#include <string>
#include <vector>
using namespace std;

#define PB push_back

typedef vector<string> Vs;
typedef vector<int> Vi;
typedef vector<char> Vc;
typedef vector<Vc> Mc;

const int dirx[4] = { 1, 0, -1, 0 };
const int diry[4] = { 0, -1, 0, 1 };

int N, M, K;

int token;
Vs expr;
Vi reg;

int eval() {

  if (expr[token] == "FALSE") {
    ++token;
    return 0;
  }
  if (expr[token] == "TRUE") {
    ++token;
    return 1;
  }

  if (expr[token].size() == 1 and 'A' <= expr[token][0] and expr[token][0] <= 'Z') {
    int t = reg[expr[token][0] - 'A'];
    ++token;
    return t;
  }
  if (expr[token] == "NOT") {
    ++token;
    return 1 - eval();
  }
  ++token;

  int r = 0;
  int t = eval();
  while (token < int(expr.size()) and (expr[token] == "OR" or expr[token] == "AND"))
      {
    if (expr[token] == "OR") {
      ++token;  // OR
      r = r | t;
      t = eval();
    }
    else {
      ++token;  // AND
      t = t & eval();
```

```
    }
  }
  r = r | t;

  ++token;
  return r;
}

int main() {
  string line;
  getline(cin, line);
  line += "      ";

  expr.PB("(");
  int n = line.size();
  int p = 0;
  while (p < n) {
    if (line[p] == ' ') ++p;
    else if (line[p] == '(' or line[p] == ')') {
      expr.PB(string(1, line[p]));
      ++p;
    }
    else if (line.substr(p, 2) == "OR") {
      expr.PB("OR");
      p += 2;
    }
    else if (line.substr(p, 3) == "NOT" or line.substr(p, 3) == "AND") {
      expr.PB(line.substr(p, 3));
      p += 3;
    }
    else if (line.substr(p, 4) == "TRUE") {
      expr.PB("TRUE");
      p += 4;
    }
    else if (line.substr(p, 5) == "FALSE") {
      expr.PB("FALSE");
      p += 5;
    }
    else {
      expr.PB(string(1, line[p]));
      ++p;
    }
  }
  expr.PB(")");

  reg = Vi(26, 0);

  cin >> N >> M >> K;

  Mc mat(2*N + 1, Vc(2*N + 1, '.'));
  for (int i = 0; i < M; ++i) {
    int x, y;
    cin >> x >> y;
    mat[x + N][y + N] = '*';
  }
  for (int i = 0; i < K; ++i) {
    int x, y;
    cin >> x >> y;
    cin >> mat[x + N][y + N];
  }
```

```
  int x = 0, y = 0, d = 0;
  while (true) {
    if (x < -N or N < x or y < -N or N < y) break;

    cout << x << " " << y << endl;

    if (mat[x + N][y + N] == '*') {
      token = 0;

      int t = eval();

      if (t) d = (d + 1)%4;
      else d = (d - 1 + 4)%4;
    }
    else if (mat[x + N][y + N] != '.') {
      reg[mat[x + N][y + N] - 'A'] = 1 - reg[mat[x + N][y + N] - 'A'];
    }

    x += dirx[d];
    y += diry[d];
  }
}
```

Timus 1102. Strange Dialog

1102. Strange Dialog

Time Limit: 1.0 second
Memory Limit: 16 MB

One entity named "one" tells with his friend "puton" and their conversation is interesting. "One" can say words "out" and "output", besides he calls his friend by name. "Puton" can say words "in", "input" and "one". They understand each other perfect and even write dialogue in strings without spaces.

You have N strings. Find which of them are dialogues.

Input

In the first line of input there is one non-negative integer $N \le 1000$. Next N lines contain non-empty strings. Each string consists of small Latin letters. Total length of all strings is no more than 10^7 characters.

Output

Output consists of N lines. Line contains word "YES", if string is some dialogue of "one" and "puton", otherwise "NO".

Sample

input	output
6 puton inonputin oneputonininputoutoutput oneininputwooutoutput outpu utput	YES NO YES NO NO NO

Problem Author: Katya Ovechkina
Problem Source: Tetrahedron Team Contest May 2001

```
/*
Timus 1102. Strange Dialog
We need to check whether the input string can be partitioned into substrings in
such a way that each of this substrings is one of the words: in, input, puton,
one, out, output.
This problem can be solved using DP. Let S be the input string, N its length
and M an array of size N + 1. Let's say that M[i] is the answer to the question:
Can the string S[i..N-1] (the suffix starting at i) be divided in the aforesaid
way? If we knew the value of M[0], then M[0] would be the answer to the problem.
In order to compute M[0], we compute each of the M[i] values starting at i=N in
decreasing order until we reach i=0. This is a very simple DP, so the details
are left to the reader. The cost of this algorithm is O(N).
The code below is not an implementation of the aforementioned DP. It is an
ad-hoc algorithm which parses the string in an on-line way from left to right.
Its cost is O(N) too.
*/
#include <iostream>
#include <cstdio>
#include <cstring>
using namespace std;

char s[10000100];

bool isok() {
    int n = strlen(s), p = 0;
    while (p < n) {
        if (s[p] == 'p') {
            if (p + 4 >= n or s[p + 1] != 'u' or s[p + 2] != 't' or
                s[p + 3] != 'o' or s[p + 4] != 'n') return false;
            p += 5;
        }
        else if (s[p] == 'o') {
            if (p + 2 >= n) return false;
            if (s[p + 1] == 'n') {
                if (s[p + 2] != 'e') return false;
                p += 3;
            }
            else {
                if (s[p + 1] != 'u' or s[p + 2] != 't') return false;
                if (p + 5 >= n or s[p + 3] != 'p' or s[p + 4] != 'u'
                    or s[p + 5] != 't') p += 3;
                else {
                    if (p + 7 >= n or s[p + 6] != 'o' or s[p + 7] != 'n') p += 6;
                    else {
                        if (p + 8 >= n or s[p + 8] != 'e') p += 3;
                        else p += 6;
                    }
                }
            }
        }
        else if (s[p] == 'i') {
            if (p + 1 >= n) return false;
            if (s[p + 1] != 'n') return false;
            if (p + 4 >= n or s[p + 2] != 'p' or s[p + 3] != 'u'
                or s[p + 4] != 't') p += 2;
            else {
                if (p + 6 >= n or s[p + 5] != 'o' or s[p + 6] != 'n') p += 5;
                else {
                    if (p + 7 >= n or s[p + 7] != 'e') p += 2;
                    else p += 5;
                }
```

```
            }
        }
        else return false;
    }
    return true;
}

int main() {
    int tcas;
    scanf("%d", &tcas);
    for (int cas = 1; cas <= tcas; ++cas) {
        scanf("%s", s);
        if (isok()) printf("YES\n");
        else printf("NO\n");
    }
}
```

Timus 1104. Don’t Ask Woman about Her Age

1104. Don’t Ask Woman about Her Age

Time Limit: 1.0 second
Memory Limit: 16 MB

Mrs Little likes digits most of all. Every year she tries to make the best number of the year. She tries to become more and more intelligent and every year studies a new digit. And the number she makes is written in numeric system which base equals to her age. To make her life more beautiful she writes only numbers that are divisible by her age minus one. Mrs Little wants to hold her age in secret.

You are given a number consisting of digits 0, …, 9 and Latin letters A, …, Z, where A equals 10, B equals 11 etc. Your task is to find the minimal number k satisfying the following condition: the given number, written in k-based system is divisible by $k-1$.

Input

Input consists of one string containing no more than 10^6 digits or uppercase Latin letters.

Output

Output the only number k, or "No solution." if for all $2 \le k \le 36$ condition written above can't be satisfied. By the way, you should write your answer in decimal system.

Sample

input	output
A1A	22

Problem Author: Igor Goldberg
Problem Source: Tetrahedron Team Contest May 2001

```
/*
Timus 1104. Don't Ask Woman about Her Age
In the implemented solution we check for each value of K (2 <= K <= 36) wether
the input number is divisible by K - 1 if interpreted in base K. In order to
check this, we compute the remainder of the division of the input number by K - 1.
Note that the input number can be very large, but we don't need any
BigInteger library in this problem, because the remainder is always very small.
Look at the code for more details about the computation of the remainder.
*/
#include <iostream>
#include <string>
using namespace std;

int num(char c) {
  if ('0' <= c and c <= '9') return c - '0';
  return 10 + c - 'A';
}

int main() {
  string s;
  cin >> s;

  int m = 0;
  int n = s.size();
  for (int i = 0; i < n; ++i)
    m = max(m, num(s[i]));

  if (m == 0) {
    cout << 2 << endl;
    return 0;
  }

  for (int k = m + 1; k <= 36; ++k) {
    int t = 0;
    for (int i = 0; i < n; ++i)
      t = (k*t + num(s[i]))%(k - 1);
    if (t == 0) {
      cout << k << endl;
      return 0;
    }
  }

  cout << "No solution." << endl;
}
```

1105. Observers Coloring

Time Limit: 0.5 second
Memory Limit: 16 MB

Nikifor told us that once he solved problem at mathematical tournament of S.Petersburg's secondary school N239 in 1994. Nikifor said that he solved a problem from the moment T_0 to the moment T_1. He remembers that N observers appeared in the room. The i-th observer entered the room at the moment $t_{0,i}$ and went out at the moment $t_{1,i}$. At every moment there was at least one observer in the room.

When the tournament was finished, Nikifor claimed that it is possible to color some observers, and the summary time when there was only one colored observer in the room is not less than 2/3 of the time when Nikifor solved problem.

You are to answer whether Nikifor right or not.

Input

The first line of input contains real numbers T_0 and T_1 ($T_0 < T_1$). The second line contains number N — number of observers ($N < 10000$). Next N lines contain real numbers $t_{0,i}$ and $t_{1,i}$ ($T_0 \le t_{0,i} < t_{1,i} \le T_1$).

Output

If Nikifor is not right output should contain the only number 0. If Nikifor is right you should write to the first line the quantity of colored observers, and next lines should contain their numbers. Do not write more than one number in a line. You may write these numbers in any order. If there are more than one solution exist you may find any of them.

Sample

input	output
0.0 20.0 7 1.0 1.5 0.0 10.0 9.0 10.0 18.0 20.0 9.0 18.0 2.72 3.14 19.0 20.0	3 2 5 7

Problem Author: Dmitry Filimonenkov feat Igor Goldberg
Problem Source: Tetrahedron Team Contest May 2001

```
/*
Timus 1105. Observers Coloring
Given a set of intervals of time which covers the interval [A, B], we need to
find a subset of intervals such that the amount of time which is covered by
exactly one interval is at least 2/3*(B - A) in the interval [A, B].
First of all, we select some of the intervals in such a way that they cover the
whole interval [A, B], and if we removed anyone of them then a gap would be
created. This can easily be done using a Greedy algorithm in O(n*log n).
Suppose we have such intervals in the array V, sorted increasingly by the first
coordinate. Now we put in a new array V1 the intervals which are in an odd
position inside V, and the rest in V2. Note that all intervals in V1 are bitwise
disjoint, and the same with V2. The amount of time covered by the intervals in
V1 equals the sum of the area covered by each of its intervals. Let's call this
area A1, and A2 the area covered by V2. If A1 >= 2/3*(B-A) or A2 >= 2/3*(B-A),
then we already have an answer to the problem. If not, then we can prove that V
is a valid solution. Here is the proof: Without loss of generality, let's suppose
B - A = 1 and that each inverval is contained inside [A, B] (we consider the
intersection of each interval with [A,B]. Given that V covers the whole interval
and A1 < 2/3, the area covered by V2 which is not covered by V1 is at least
1 - A1 > 1/3. Symmetrically, we can see that the area E1 covered by V1 and not
by V2 is at least 1 - A2 > 1/3. The area covered by exactly one interval of V
is E1 + E2 > 1/3 + 1/3 = 2/3.
*/
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
using namespace std;

#define X first
#define Y second
#define PB push_back

typedef pair<double, double> P;
typedef vector<P> Vp;
typedef vector<int> Vi;
typedef pair<P, int> PP;
typedef vector<PP> Vpp;

const double EPS = 1e-8;

int main() {
  double a, b;
  cin >> a >> b;

  int n;
  cin >> n;

  Vp v(n);
  for (int i = 0; i < n; ++i) cin >> v[i].X >> v[i].Y;

  Vpp aux(n);
  for (int i = 0; i < n; ++i) aux[i] = PP(v[i], i);
  sort(aux.begin(), aux.end());
  for (int i = 0; i < n; ++i) v[i] = aux[i].X;

  Vi res;
  double p = a;
  int m = 0;
  while (p + EPS < b) {
    int q = m;
```

```
    while (m < n and v[m].X <= p + EPS) {
      if (v[q].Y < v[m].Y) q = m;
      ++m;
    }
    res.PB(q);
    p = v[q].Y;
  }

  m = res.size();

  double r1 = 0;
  for (int i = 0; i < m; i += 2) r1 += v[res[i]].Y - v[res[i]].X;
  if (3*r1 + EPS >= 2*(b - a)) {
    cout << (m + 1)/2 << endl;
    for (int i = 0; i < m; i += 2)
      cout << aux[res[i]].Y + 1 << endl;
    return 0;
  }

  double r2 = 0;
  for (int i = 1; i < m; i += 2) r2 += v[res[i]].Y - v[res[i]].X;
  if (3*r2 + EPS >= 2*(b - a)) {
    cout << m/2 << endl;
    for (int i = 1; i < m; i += 2)
      cout << aux[res[i]].Y + 1 << endl;
    return 0;
  }

  cout << m << endl;
  for (int i = 0; i < m; ++i)
    cout << aux[res[i]].Y + 1 << endl;
}
```

Timus 1106. Two Teams

1106. Two Teams

Time Limit: 1.0 second
Memory Limit: 16 MB

The group of people consists of N members. Every member has one or more friends in the group. You are to write program that divides this group into two teams. Every member of each team must have friends in another team.

Input

The first line of input contains the only number N ($N \le 100$). Members are numbered from 1 to N. The second, the third,…and the (N+1)th line contain list of friends of the first, the second, …and the Nth member respectively. This list is finished by zero. Remember that friendship is always mutual in this group.

Output

The first line of output should contain the number of people in the first team or zero if it is impossible to divide people into two teams. If the solution exists you should write the list of the first group into the second line of output. Numbers should be divided by single space. If there are more than one solution you may find any of them.

Sample

input	output
7 2 3 0 3 1 0 1 2 4 5 0 3 0 3 0 7 0 6 0	4 2 4 5 6

Problem Author: Dmitry Filimonenkov
Problem Source: Tetrahedron Team Contest May 2001

```cpp
/*
Timus 1106. Two Teams
It is always possible to divide the group into 2 teams, as long as every person
has at least 1 friend. The division can be done by starting from some unassigned
person, placing it in one of the teams, and performing a BFS in which the people
at an even distance are in the same team and the people at an odd distance are
in the opposite team. Repeat the procedure until every person is assigned.
*/
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

#define PB push_back

typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef queue<int> Q;

int N;
Mi net;

Vi vist;
Vi res;

int bfs(int a) {
  Q q;
  q.push(a);

  vist[a] = 1;
  int vistos = 1;

  int col = 1;
  while (not q.empty()) {
    for (int qq = q.size(); qq > 0; --qq) {
      int n = q.front();
      q.pop();

      if (col) res.PB(n);

      for (int i = 0; i < int(net[n].size()); ++i) {
        int m = net[n][i];
        if (vist[m] == 0) {
          vist[m] = 1;
          ++vistos;
          q.push(m);
        }
      }
    }
    col = 1 - col;
  }

  return vistos;
}

int main() {
  cin >> N;

  net = Mi(N);
  for (int i = 0; i < N; ++i) {
    int j;
```

```
    while (cin >> j and j > 0) {
      --j;
      net[i].PB(j);
    }
  }

  bool ok = true;

  vist = Vi(N, 0);
  for (int i = 0; i < N; ++i) {
    if (vist[i] == 0) {
      if (bfs(i) < 2) ok = false;
    }
  }

  if (ok) {
    int sz = res.size();
    cout << sz << endl;
    for (int i = 0; i < sz; ++i) {
      if (i) cout << " ";
      cout << res[i] + 1;
    }
    cout << endl;
  }
  else cout << 0 << endl;
}
```

1107. Warehouse Problem

Time Limit: 1.0 second
Memory Limit: 16 MB

The are N different types of goods at the warehouse. Types are numbered by numbers $1 \ldots N$. Employees of this warehouse made K different sets of these goods. We'll say that two sets are "similar" if one of them is obtained by deleting one good form the second set or by replacing one good to another.

E.g. Set "1 2 3 4" is similar to sets "3 2 1", "1 2 5 3 4", "1 2 3 4 2" and "1 5 4 3" and is not similar to "1 2", "1 1 2 2 3 4" and "4 5 3 6".

This warehouse serves M shops ($0 < N < M < 101$), sending them sets of goods. Every two sets sent to the shop should not be similar. It is possible not to send any set to one or more shops.

You are to write program that determines how to distribute all K sets to these M shops.

Input

The first line contains numbers N, K, M. Then K lines describing every set of goods follow, $K \le 50000$. Each of these lines is started with the number of goods in the set, then numbers of goods are written. Number of goods in any set is more than 0 and less than 101. All numbers in these lines are separated by exactly one space.

Output

The first line of the output should contain word YES if the solution exists or NO contrary. If the answer is YES write the numbers of the shops where sets should be sent to. In the second line you have to write number of the shop where the first set should be sent to, the third — for the second set, etc. If there are more than one solution exist you may find any of them.

Sample

input	output
8 20 12 5 1 3 5 6 4 5 1 3 5 6 3 4 5 6 3 3 4 5 6 3 4 4 4 6 5 8 4 7 7 7 7 3 7 7 7 2 2 2 3 2 2 7 3 1 2 3 3 1 2 4 10 1 2 3 4 5 6 7 8 7 6 10 8 7 6 5 4 3 2 1 2 1 20 1 2 3 4 5 6 7 8 1 2 3 4 5 6 7 8 1 3 5 7 5 4 6 4 6 4 5 6 4 6 4 6 6 6 6 6 6 6 6 3 6 6 6 1 1 1 2	YES 2 1 9 1 6 2 4 5 3 7 8 5 4 8 7 9 1 1 2 3

Problem Author: Dmitry Filimonenkov
Problem Source: Tetrahedron Team Contest May 2001

```cpp
/*
Timus 1107. Warehouse Problem
Let N be the number of different goods, K the number of sets, M the number of
stores, and F(S) the sum of the identifiers of the products in S. It's easy to
prove that if U, V are two similar sets then F(U)%(N + 1) = F(V)%(N + 1). Given
that N < M, we get a correct solution if we assign the set S to the store
identified by F(S)%(N + 1)+1.
*/
#include <iostream>
#include <cstdio>
using namespace std;

int main() {
  int N, K, M;
  scanf("%d%d%d", &N, &K, &M);
  printf("YES\n");
  for (int i = 0; i < K; ++i) {
    int n;
    scanf("%d", &n);
    int m = 0;
    for (int j = 0; j < n; ++j) {
      int t;
      scanf("%d", &t);
      m += t;
    }
    printf("%d\n", m%(N + 1) + 1);
  }
}
```

1108. Heritage

Time Limit: 2.0 second
Memory Limit: 16 MB

Your rich uncle died recently, and the heritage needs to be divided among your relatives and the church (your uncle insisted in his will that the church must get something). There are N relatives ($N \le 18$) that were mentioned in the will. They are sorted in descending order according to their importance (the first one is the most important). Since you are the computer scientist in the family, your relatives asked you to help them. They need help, because there are some blanks in the will left to be filled. Here is how the will looks:

```
Relative #1 will get 1/... of the whole heritage,
Relative #2 will get 1/... of the whole heritage,
...
Relative #N will get 1/... of the whole heritage.
```

The logical desire of the relatives is to fill the blanks in such way that the uncle's will is preserved (i.e the fractions are non-ascending and the church gets something) and the amount of heritage left for the church is minimized.

Input

The only line of input contains the single integer N ($1 \le N \le 18$).

Output

Output the numbers that the blanks need to be filled (on separate lines), so that the heritage left for the church is minimized.

Sample

input	output
2	2 3

Problem Author: Pavlin Peev

```
/*
Timus 1108. Heritage
The i-th relative gets 1/F_i of the heritage, where F_1 = 2,
and F_i = F_1*F_2*...*F_(i-1) + 1 when i > 1.
*/
#include <iostream>
#include <vector>
#include <iomanip>
using namespace std;

typedef long long ll;
typedef vector<int> Vi;

const ll base = 1000000000;

class BI {
public:
  Vi v;
  int n;
  int sig;

  int dig(int i) const { return i<n?v[i]:0;}
  void remida(int i, int que=0) {v.resize(n=i,que);}
  void treuzeros(){
    while (--n and !v[n]);
    remida(n+1);
    if (n == 1 and !v[0]) sig = 1;
  }
  void operator +=(const BI& b) {
    if (n < b.n) remida(b.n, 0);
    int ca = 0;
    for (int i = 0; i < n; ++i) {
      v[i] += b.dig(i) + ca; ca = v[i]/base; v[i] %= base;
    }
    if (ca) remida(n+1, ca);
  }
  BI() : n(-1) {};
  BI(int x) { remida(1,x); }
  BI(const BI& b) : v(b.v), n(b.n) {}

  void operator*=(BI &b) {
    BI c; c.remida(n + b.n, 0); c.sig = sig*b.sig;
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < b.n; ++j) {
        int k = i + j;
        ll z = (ll)v[i]*b.v[j] + c.v[k], y;
        while ((y = z / base)) { c.v[k] = z % base; z = y + c.v[++k]; }
        c.v[k] = z;
      }
      c.treuzeros();
      *this = c;
  }
  friend ostream &operator<<(ostream &out, BI &b) {
    if (b.sig < 0) out << '-';
    int i = b.v.size() - 1;
    out << b.v[i];
    for (--i; i >= 0; --i) out << setw(9) << setfill('0') << b.v[i];
    return out;
  }
};
```

```
int main() {
  int n;
  cin >> n;

  BI one = 1;
  BI pr = one;

  for (int i = 0; i < n; ++i) {
    BI res = pr;
    res += one;
    pr *= res;
    cout << res << endl;
  }
}
```

1109. Conference

Time Limit: 0.5 second
Memory Limit: 16 MB

On the upcoming conference were sent *M* representatives of country *A* and *N* representatives of country *B* (*M* and *N* ≤ 1000). The representatives were identified with 1, 2, …, *M* for country A and 1, 2, …, *N* for country *B*. Before the conference *K* pairs of representatives were chosen. Every such pair consists of one member of delegation *A* and one of delegation *B*. If there exists a pair in which both member #*i* of *A* and member #*j* of *B* are included then #*i* and #*j* can negotiate. Everyone attending the conference was included in at least one pair. The CEO of the congress center wants to build direct telephone connections between the rooms of the delegates, so that everyone is connected with at least one representative of the other side, and every connection is made between people that can negotiate. The CEO also wants to minimize the amount of telephone connections. Write a program which given *M*, *N*, *K* and *K* pairs of representatives, finds the minimum number of needed connections.

Input

The first line of the input contains *M*, *N* and *K*. The following *K* lines contain the choosen pairs in the form of two integers p_1 and p_2, p_1 is member of *A* and p_2 is member of *B*.

Output

The output should contain the minimum number of needed telephone connections.

Sample

input	output
3 2 4 1 1 2 1 3 1 3 2	3

Problem Source: Bulgarian National Olympiad Day #1

```
/*
Timus 1109. Conference
We want to remove the maximum number of edges of the graph in such a way that
every node still has at least one adjacent edge. This can be solved using the
MaxFlow algorithm. Let S be the source, T be the sink, A_i the nodes of the
first group and B_i the nodes of the second one. We assign 1 unit of maximum
capacity to each edge between A_i and B_j. Now we add an edge between S and A_i
for each i, with maximum capacity of deg(A_i) - 1, where deg(v) means the
number of adjacent edges to the node v. Also, we add one edge between B_i and T
for each i, with capacity of deg(B_i) - 1. After running the MaxFlow algorithm,
the saturated edges of the original graph are the ones we can discard.
Note that each A_i node has at least one non-saturated edge (because the
capacity of the edge from S to A_i is deg(A_i) - 1) and the same for the B_i
nodes. So, the answer is the total number of edges initially minus the maximum
flow value.
*/
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

#define PB push_back

typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef queue<int> Q;
typedef vector<short> Vsh;
typedef vector<Vsh> Msh;

int N;
Msh net, flow;

int A, B, M;

int maxflow(int a, int b) {
  int maxf = 0;
  while (true) {
    Q q;
    q.push(a);

    Vi pare(N, -1);
    pare[a] = a;

    while (not q.empty() and pare[b] == -1) {
      int n = q.front();
      q.pop();
      for (int i = 0; i < int(net[n].size()); ++i) {
        int m = net[n][i];
        if (pare[m] == -1 and flow[n][m] > 0) {
          pare[m] = n;
          q.push(m);
        }
      }
    }

    if (pare[b] == -1) return maxf;

    for (int i = 0; i < int(net[b].size()); ++i) {
      int s = net[b][i];
      if (pare[s] == -1) continue;
```

```
      int f = flow[s][b];
      for (int j = s; j != a; j = pare[j]) f = min(f, int(flow[pare[j]][j]));

      flow[s][b] -= f;
      flow[b][s] += f;
      for (int j = s; j != a; j = pare[j]) {
        flow[pare[j]][j] -= f;
        flow[j][pare[j]] += f;
      }

      maxf += f;
    }
  }
}

int main() {
  cin >> A >> B >> M;

  N = A + B + 2;
  net = Msh(N);
  flow = Msh(N, Vsh(N, 0));

  for (int i = 0; i < M; ++i) {
    int a, b;
    cin >> a >> b;
    --a; --b;

    ++flow[a][A + b];
    net[a].PB(A + b);
    net[A + b].PB(a);
  }

  for (int i = 0; i < A; ++i) {
    flow[N - 2][i] = net[i].size() - 1;
    net[N - 2].PB(i);
    net[i].PB(N - 2);
  }

  for (int i = 0; i < B; ++i) {
    flow[A + i][N - 1] = net[A + i].size() - 1;
    net[A + i].PB(N - 1);
    net[N - 1].PB(A + i);
  }

  int mf = maxflow(N - 2, N - 1);

  cout << M - mf << endl;
}
```

1110. Power

Time Limit: 0.5 second
Memory Limit: 16 MB

You are given the whole numbers N, M and Y. Write a program that will find all whole numbers X in the interval $[0, M-1]$ such that $X^N \bmod M = Y$.

Input

The input contains a single line with N, M and Y ($0 < N < 999$, $1 < M < 999$, $0 < Y < 999$) separated with one space.

Output

Output all numbers X separated with space on one line. The numbers must be written in ascending order. If no such numbers exist then output -1.

Sample

input	output
2 6 4	2 4

Problem Source: Bulgarian National Olympiad Day #1

```cpp
/*
Timus 1110. Power
Given the weak constraints in this problem, one can simply check for every
x in [0, M - 1] whether x^N mod M = Y.
If N were larger, we could still solve the problem by doing fast exponentiation.
*/
#include <iostream>
using namespace std;

int main() {
  int n, m, y;
  cin >> n >> m >> y;

  bool f = true;
  for (int x = 0; x < m; ++x) {
    int t = 1;
    for (int i = 0; i < n; ++i) t = (t*x)%m;
    if (t == y) {
      if (f) f = false;
      else cout << " ";
      cout << x;
    }
  }
  if (f) cout << -1;
  cout << endl;
}
```

1111. Squares

Time Limit: 0.5 second
Memory Limit: 16 MB

You are given n ($1 \le n \le 50$) squares and point P. The distance between P and square is the shortest line segment that connects P with the contour or the internal area of the square. If P is inside the square then the distance is zero. It is possible some squares to be points i.e. to have vertices that coincide. Write a program that will sort the squares in ascending order according the distance from P.

Input

The first line contains the integer n. The following n lines contain four integers in the range $(-9999, 9999)$. The first two numbers define the x and y coordinates of one of the vertices of the square, the next two numbers define the opposite vertex. The last line contains the x and y coordinates of P.

Output

The output should be a line containing the ids of the squares sorted according to the distance from P. The ids are defined according to the order in which the squares are given in the input. Use ids to break ties i.e. if two squares are the same distance from P then write the square with the lowest id first. Using 10^{-14} precision when comparing the distances is accurate enough.

Sample

input	output
2 0 0 1 1 0 3 1 4 0 0	1 2

```
/*
Timus 1111. Squares
The geometric part of this problem is the computation of the distance from each
square to the given point. Sort the id's 1..N by the distance from the
corresponding square to the point, solving the ties properly.
*/
#include <iostream>
#include <utility>
#include <vector>
#include <cmath>
#include <algorithm>
using namespace std;

#define X first
#define Y second

typedef long double ld;
typedef pair<ld, ld> P;
typedef pair<P, P> PP;
typedef vector<PP> Vpp;
typedef pair<ld, int> Pdi;
typedef vector<Pdi> Vdi;

const ld EPS = 1e-14;

P operator*(ld a, P p) {
  return P(a*p.X, a*p.Y);
}

P operator+(P a, P b) {
  return P(a.X + b.X, a.Y + b.Y);
}

P operator-(P a, P b) {
  return P(a.X - b.X, a.Y - b.Y);
}

ld operator^(P a, P b) {
  return a.X*b.Y - a.Y*b.X;
}

ld operator*(P a, P b) {
  return a.X*b.X + a.Y*b.Y;
}

istream& operator>>(istream& in, P& p) {
  in >> p.X >> p.Y;
  return in;
}

istream& operator>>(istream& in, PP& pp) {
  in >> pp.X >> pp.Y;
  return in;
}

P left(P p) {
  return P(-p.Y, p.X);
}

ld prod(P a, P b, P c) {
  return (c - b)^(a - b);
```

```
}

ld dist(P a, P b) {
  ld x = b.X - a.X;
  ld y = b.Y - a.Y;
  return sqrt(x*x + y*y);
}

ld dist(PP square, P point) {
  P a = square.X;
  P c = square.Y;
  P b = 0.5*(a + c) + left(0.5*(a - c));
  P d = 0.5*(a + c) - left(0.5*(a - c));

  if (prod(b, point, a) >= -EPS and prod(c, point, b) >= -EPS and
      prod(d, point, c) >= -EPS and prod(a, point, d) >= -EPS) {
    return 0;
  }

  ld res = min(min(dist(a, point), dist(b, point)),
               min(dist(c, point), dist(d, point)));

  ld side = dist(a, b);

  if ((point - a)*(b - a) >= 0 and (point - b)*(a - b) >= 0)
    res = min(res, abs((b - point)^(a - point))/side);

  if ((point - b)*(c - b) >= 0 and (point - c)*(b - c) >= 0)
    res = min(res, abs((c - point)^(b - point))/side);

  if ((point - c)*(d - c) >= 0 and (point - d)*(c - d) >= 0)
    res = min(res, abs((d - point)^(c - point))/side);

  if ((point - d)*(a - d) >= 0 and (point - a)*(d - a) >= 0)
    res = min(res, abs((a - point)^(d - point))/side);

  return res;
}

bool sortf(Pdi a, Pdi b) {
  if (abs(b.X - a.X) <= EPS) return a.Y < b.Y;
  return a.X < b.X;
}

int main() {
  int n;
  cin >> n;

  Vpp v(n);
  for (int i = 0; i < n; ++i) cin >> v[i];

  P p;
  cin >> p;

  Vdi u(n);
  for (int i = 0; i < n; ++i) u[i] = Pdi(dist(v[i], p), i);
  sort(u.begin(), u.end(), sortf);

  for (int i = 0; i < n; ++i) {
    if (i) cout << " ";
    cout << u[i].Y + 1;
```

```
  }
  cout << endl;
}
```

1112. Cover

Time Limit: 0.5 second
Memory Limit: 16 MB

You are given N ($0 < N < 100$) line segments on a line. Every segment is defined with its endpoints A_i and B_i ($A_i \neq B_i$, $1 \leq i \leq N$). The endpoints are integer coordinates in the interval [−999, 999]. Some of the segments probably intersect. Write a program, which removes minimum number of the given segments, so that none of the left segments have common interior point.

Input

The first line of input contains the integer N. Each of the following N lines, contains two integers (A_i and B_i), separated with one space.

Output

On the first line write the integer P, equal to the number of segments, which are left after your program removes the excess segments. The following P lines should contain the coordinates of the left and the right endpoints of the segments which are left. These coordinates must be separated with one space. Coordinates of the left endpoints must be written in their ascending order. If the problem has more the one solution, write only one of them no matter which.

Sample

input	output
3 6 3 1 3 2 5	2 1 3 3 6

Problem Source: Bulgarian National Olympiad Day #2

```cpp
/*
Timus 1112. Cover
This problem can be solved with DP. Sort all the intervals by their starting.
Let V[i] be the i-th interval after sorting.
dp[i] will be the minimum number of intervals that need to be removed from
V[i..N] so they don't intersect. The values of dp[i] should be computed
starting from i=N to i=0, where dp[0] is the answer to the problem.
The details of the implementation are left to the reader. The implementation
presented here runs in O(N^2), but can be solved in O(N*log(N)).
*/
#include <iostream>
#include <vector>
#include <map>
using namespace std;

#define PB push_back
#define X first
#define Y second

typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef pair<int, int> P;
typedef vector<P> Vp;
typedef vector<Vp> Mp;
typedef map<int, int> MAP;
typedef MAP::iterator Mit;

int main() {
  int n;
  cin >> n;

  MAP mp;

  Vp v(n);
  for (int i = 0; i < n; ++i) {
    cin >> v[i].X >> v[i].Y;
    ++mp[v[i].X];
    ++mp[v[i].Y];
  }

  int m = 0;
  for (Mit it = mp.begin(); it != mp.end(); ++it) it->Y = m++;

  Mi mat(m);
  Mp que(m);
  for (int i = 0; i < n; ++i) {
    if (v[i].X > v[i].Y) swap(v[i].X, v[i].Y);
    mat[mp[v[i].X]].PB(mp[v[i].Y]);
    que[mp[v[i].X]].PB(v[i]);
  }

  Vi dp(m, 0);
  Vp res(m);
  for (int i = m - 2; i >= 0; --i) {
    res[i] = P(1000, 1000);
    dp[i] = dp[i + 1];
    for (int j = 0; j < int(mat[i].size()); ++j) {
      int t = 1 + dp[mat[i][j]];
      if (t > dp[i]) {
        dp[i] = t;
        res[i] = que[i][j];
```

```
      }
    }
  }

  cout << dp[0] << endl;

  int t = 0;
  while (t + 1 < m) {
    if (res[t].X == 1000) ++t;
    else {
      cout << res[t].X << " " << res[t].Y << endl;
      t = mp[res[t].Y];
    }
  }
}
```

1114. Boxes

Time Limit: 0.6 second
Memory Limit: 16 MB

N boxes are lined up in a sequence ($1 \le N \le 20$). You have *A* red balls and *B* blue balls ($0 \le A \le 15$, $0 \le B \le 15$). The red balls (and the blue ones) are exactly the same. You can place the balls in the boxes. It is allowed to put in a box, balls of the two kinds, or only from one kind. You can also leave some of the boxes empty. It's not necessary to place all the balls in the boxes. Write a program, which finds the number of different ways to place the balls in the boxes in the described way.

Input

Input contains one line with three integers *N*, *A* and *B* separated by space.

Output

The result of your program must be an integer written on the only line of output.

Sample

input	output
`2 1 1`	`9`

Problem Source: First competition for selecting the Bulgarian IOI team.

```
/*
Timus 1114. Boxes
This is an easy combinatorics problem which can be solved by DP.
In the implementation below, dp[k][i][j] is number of ways of distributing
i red balls and j blue balls among k boxes.
*/
#include <iostream>
using namespace std;

typedef unsigned long long ull;

ull dp[21][16][16];

int main() {
  for (int i = 0; i <= 15; ++i)
    for (int j = 0; j <= 15; ++j)
      dp[0][i][j] = 1;
  for (int k = 1; k <= 20; ++k)
    for (int i = 0; i <= 15; ++i)
      for (int j = 0; j <= 15; ++j)
        for (int x = 0; x <= i; ++x)
          for (int y = 0; y <= j; ++y)
            dp[k][i][j] += dp[k - 1][i - x][j - y];

  int n, a, b;
  cin >> n >> a >> b;
  cout << dp[n][a][b] << endl;
}
```

1115. Ships

Time Limit: 1.0 second
Memory Limit: 16 MB

The military intelligence of one country found out that N ($N < 100$) battle ships of neighboring enemy country are situated in M rows ($1 < M < 10$). The intelligence knows the lengths $l_1, l_2, \ldots, l_N$ of the battle ships which are whole numbers in the interval [1, 100], and wants to know in which rows the ships are situated. The only thing that is known about the M rows are their lengths — $L_1, L_2, \ldots, L_M$. Assume that the ships touch their neighbours in the rows and that every row contains at least one ship. Write program that will find one possible ordering of the ships in rows.

Input

The first line of the input contains N and M. The next N lines contain the lengths of the ships. The next M lines contain the lengths of the rows.

Output

The output should contain M pairs of lines. The first line of each pair should contain the amount of the ships in the current row, the following line should contain the lengths of the ships from the current row. The order of the M row descriptions should be the same as the order in which the rows are given in the input.

Sample

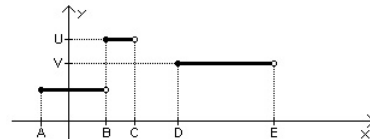
input	output
5 2 4 10 2 5 3 11 13	3 5 4 2 2 10 3

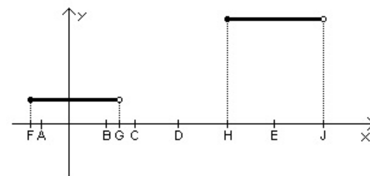
Problem Source: First competition for selecting the Bulgarian IOI team.

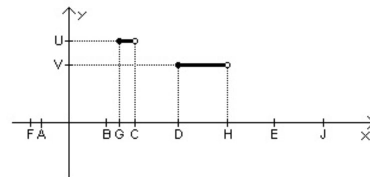
```cpp
/*
Timus 1115. Ships
This problem can be solved by brute force. We assign a row to each ship with
a standard backtracking. Also, just to improve the running time, we prune
the backtracking using a precomputed table, called dp[][] in the code below.
dp[i][j] stores whether a row of length j can be filled completely using only
ships with id greater than or equal to i.
*/
#include <iostream>
using namespace std;

int N, M;
int dp[110][11000];
int fila[10];
int ship[110];
int res[10][110];
int qt[10];

bool back(int n) {
  if (n == N) return true;

  for (int i = 0; i < M; ++i) {
    if (fila[i] < ship[n]) continue;
    fila[i] -= ship[n];
    res[i][qt[i]++] = n;

    bool ok = true;
    for (int j = 0; ok and j < M; ++j)
      if (dp[n + 1][fila[j]] == 0)
        ok = false;

    if (ok and back(n + 1)) return true;
    --qt[i];
    fila[i] += ship[n];
  }
  return false;
}

int main() {
  cin >> N >> M;
  for (int i = 0; i < N; ++i) cin >> ship[i];
  for (int i = 0; i < M; ++i) cin >> fila[i];

  dp[N][0] = 1;
  for (int n = N - 1; n >= 0; --n)
    for (int m = 0; m < 11000; ++m)
      if (dp[n + 1][m] or (ship[n] <= m and dp[n + 1][m - ship[n]]))
        dp[n][m] = 1;

  back(0);
  for (int i = 0; i < M; ++i) {
    cout << qt[i] << endl;
    for (int j = 0; j < qt[i]; ++j) {
      if (j) cout << " ";
      cout << ship[res[i][j]];
    }
    cout << endl;
  }
}
```

1116. Piecewise Constant Function

Time Limit: 1.0 second
Memory Limit: 16 MB







SKB Kontur has been developing a new *SKB Kontur Framework* for the last three months. One of the latest wrinkles is that data will be presented with the help of piecewise constant functions. Your team is to implement an operation of "subsettraction" of the functions. A function is called piecewise constant if its domain can be divided into intervals and the function is constant on each interval. We also assume that the function value at the left-end point of each interval of constancy is equal to its value on the interval. In fig.1 there is a piecewise constant function with three intervals of constancy. Note that the function value at the point B is U and at the points C, E and on the interval (C,D) - the function value is not defined.

A result of the subsettraction operation of two piecewise constant functions F_1 @ F_2 is a piecewise constant function F defined as follows:

- $F(x) = F_1(x)$ if F_1 is defined and F_2 is not defined;
- $F(x)$ is not defined if F_1 and F_2 are defined both;
- $F(x)$ is not defined if F_1 is not defined.

In Fig.3 there is the result of the operation of the subsettraction of the two functions from Fig.1 and Fig.2.

Input

contains two lines of the same format. Each line characterizes one piecewise constant function. A line begins with an integer N ($1 \le N \le 14999$). Then characterizations of constancy intervals follow in the ascending order with respect to their left ends. Each interval is given by three integer numbers A, B, Y ($|A| < 32000$, $|B| < 32000$, $|Y| \le 100$, $A < B$), where A is the left end of an interval, B is the right end of an interval and Y is the function value on the interval. It's known that that no two intervals from one line intersect. If two intervals are adjacent, the function values on the intervals are different.

Output

contains one line of the same format (see the input specification). This line should describe a result of the operation of subsettraction of the two input piecewise constant functions.

Sample

input	output
`3 -1 1 2 1 3 4 4 6 3` `2 -2 2 1 5 7 5`	`2 2 3 4 4 5 3`

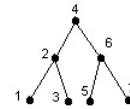
Problem Author: Oleg Kaz
Problem Source: USU Open Collegiate Programming Contest October'2001 Junior Session

```
/*
Timus 1116. Piecewise Constant Function
Store and sort all the start and end points of the intervals in the input,
we call these interesting points. Sort the intervals of each function
and iterate over the interesting points keeping one pointer for each function,
which points to the interval the current interesting point belongs to. This
allows us to identify efficiently for each interval between adjacent
interesting points whether it is defined in one function, none, or both.
*/
#include <iostream>
#include <utility>
#include <vector>
#include <set>
using namespace std;

#define X first
#define Y second
#define PB push_back

typedef long long ll;
typedef pair<int, int> P;
typedef pair<P, int> PP;
typedef vector<PP> Vpp;
typedef set<int> SET;
typedef vector<int> Vi;

int main() {
  int na;
  cin >> na;
  Vpp va(na);
  for (int i = 0; i < na; ++i)
    cin >> va[i].X.X >> va[i].X.Y >> va[i].Y;

  int nb;
  cin >> nb;
  Vpp vb(nb);
  for (int i = 0; i < na; ++i)
    cin >> vb[i].X.X >> vb[i].X.Y >> vb[i].Y;

  SET st;
  for (int i = 0; i < na; ++i) {
    st.insert(va[i].X.X);
    st.insert(va[i].X.Y);
  }
  for (int i = 0; i < nb; ++i) {
    st.insert(vb[i].X.X);
    st.insert(vb[i].X.Y);
  }

  Vpp res;
  Vi v(st.begin(), st.end());
  int m = v.size(), a = 0, b = 0;
  for (int i = 0; i + 1 < m; ++i) {
    int x = v[i], y = v[i + 1];
    while (a < na and va[a].X.Y <= x) ++a;
    while (b < nb and vb[b].X.Y <= x) ++b;
    if (a < na and va[a].X.X <= x and
        (b >= nb or x < vb[b].X.X))
      res.PB(PP(P(x, y), va[a].Y));
  }
```

```
  Vpp resok;
  {
    int i = 0;
    int n = res.size();
    while (i < n) {
      int j = 1;
      while (i + j < n and res[i + j - 1].X.Y == res[i + j].X.X and res[i].Y == res[
          i + j].Y) ++j;
      resok.PB(PP(P(res[i].X.X, res[i + j - 1].X.Y), res[i].Y));
      i += j;
    }
  }

  int sz = resok.size();
  cout << sz;
  for (int i = 0; i < sz; ++i) cout << " " << resok[i].X.X << " " << resok[i].X.Y <<
       " " << resok[i].Y;
  cout << endl;
}
```

1117. Hierarchy

Time Limit: 1.0 second
Memory Limit: 16 MB



During long years of work in SKB Kontur a certain hierarchy of employees has been developed. Each person except ordinary employees has exactly two direct subordinates and not more than one direct superior. There is no subordinate of ordinary employees (see figure).

Each employee has his own number. Of course, different employees have different numbers. It's known as well that either an employee has the maximal number or there is another employee whose number is greater by one. Similarly, either an employee has a number "1" or there is another employee whose number is less by one. The number of intermediate levels of employees between an arbitrary employee who has subordinates (an ordinary employee) and the employee who has no superiors (the main superior) is the same for all ordinary employees.

The things have come round so that each employee who has subordinates has got a number greater than the number of one of his subordinates and less than the number of the other. Moreover, if his number is greater than his superior's then the numbers of his subordinates are also greater than the number of his superior. And conversely, if his number is less, then his subordinates' numbers are less too.

A special system of intracorporate message exchange has been worked out. A message from an employee with the number i can be addressed directly only to the employees $i-1$ and $i+1$. Moreover, this is done the same day (it takes 0 days to do that) if the employees are direct superior and subordinate. Otherwise, the message delivery takes an amount of days that is equal to the number of intermediate employees between the sender and recipient. For example, a message from the employee 2 to the employee 4 is being delivered as follows. The employee 2 sends the message to the employee 3, and the employee 3 addresses it to the employee 4. This process takes one day because the first step (2->3) takes 0 days, and the second one (3->4) takes 1 day.

Input

The only line contains two positive integers: the number of an employee who sends a message and the number of the recipient. Each of the numbers doesn't exceed $2^{31}-1$.

Output

You should output the only number — the number of days necessary to deliver the message.

Sample

input	output
1 5	2

Problem Author: Alexander Somov
Problem Source: USU Open Collegiate Programming Contest October'2001 Junior Session

```cpp
/*
Timus 1117. Hierarchy
aladreta() computes the cost of traversing the tree in the way described in the
statement from node 1 to a given node. We compute this value for the first node
and for the second one, and output the difference between these two.
dp[i] is the cost of traversing a complete binary tree of height i, this value
is used by aladreta() to avoid redundant calculations.
*/
#include <iostream>
#include <algorithm>
using namespace std;

typedef long long ll;

ll dp[40];

ll aladreta(ll e, ll d, ll nd, int h) {
  ll m = (e + d)/2;
  if (nd < m) {
    ll res = aladreta(e, m - 1, nd, h - 1);
    res += h - 1;
    res += h - 1;
    res += dp[h - 1];
    return res;
  }
  else if (nd > m) {
    return aladreta(m + 1, d, nd, h - 1);
  }
  else {
    if (h == 0) return 0;
    return h - 1 + dp[h - 1];
  }
}

int main() {
  dp[0] = 0;
  for (int i = 1; i < 40; ++i) dp[i] = 2*(i - 1 + dp[i - 1]);

  ll a, b;
  cin >> a >> b;

  ll ta = aladreta(1, (1LL<<31) - 1, a, 30);
  ll tb = aladreta(1, (1LL<<31) - 1, b, 30);

  cout << abs(ta - tb) << endl;
}
```

1118. Nontrivial Numbers

Time Limit: 2.0 second
Memory Limit: 16 MB

Specialists of SKB Kontur have developed a unique cryptographic algorithm for needs of information protection while transmitting data over the Internet. The main advantage of the algorithm is that you needn't use big numbers as keys; you may easily do with natural numbers not exceeding a million. However, in order to strengthen endurance of the cryptographic system it is recommended to use special numbers - those that psychologically seem least "natural". We introduce a notion of *triviality* in order to define and emphasize those numbers.

Triviality of a natural number N is the ratio of the sum of all its proper divisors to the number itself. Thus, for example, triviality of the natural number 10 is equal to 0.8 = (1 + 2 + 5) / 10 and triviality of the number 20 is equal to 1.1 = (1 + 2 + 4 + 5 + 10) / 20. Recall that *a proper divisor* of a natural number is the divisor that is strictly less than the number.

Thus, it is recommended to use as nontrivial numbers as possible in the cryptographic protection system of SKB Kontur. You are to write a program that will find the less trivial number in a given range.

Input

The only line contains two integers I and J, $1 \le I \le J \le 10^6$, separated with a space.

Output

Output the only integer N satisfying the following conditions:

1. $I \le N \le J$;
2. N is the least trivial number among the ones that obey the first condition.

Sample

input	output
`24 28`	`25`

Problem Author: Leonid Volkov
Problem Source: USU Open Collegiate Programming Contest October'2001 Junior Session

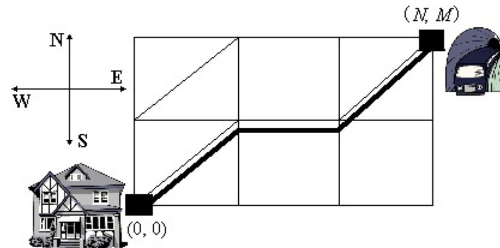
```cpp
/*
Timus 1118. Nontrivial Numbers
Compute the sum of divisors for every number in (0, J] using dynamic programming
using the fact that
#div(p_1^e_1 * p_2^e_2 * ... * p_(k-1)^e_(k-1) * p_k^e_k) =
#div(p_1^e_1 * p_2^e_2 * ... * p_(k-1)^e_(k-1))*(1 + p_k + p_k^2 + ... + p_k^e_k)
where p_i is prime for all i.
Then iterate over all N in [I, J] and output the one that minimizes
sum_of_divisors(N)/N.
*/
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;

int main() {
  int I, J;
  cin >> I >> J;
  vector<ll> factor(J+5, 0);
  factor[1] = 1;
  for (int i = 2; i*i < int(factor.size()); ++i) if (factor[i] == 0) {
    for (int j = 2*i; j < int(factor.size()); j += i) {
      factor[j] = i;
    }
  }
  vector<ll> divsum(J+5, 0);
  divsum[1] = 1;
  for (ll i = 2; i < int(divsum.size()); ++i) {
    if (factor[i] == 0) divsum[i] = 1+i;
    else {
      ll psum = 0, p = 1;
      for (; i%p == 0; p *= factor[i]) {
        psum += p;
      }
      p /= factor[i];
      divsum[i] = divsum[i/p]*psum;
    }
  }
  ll best = I;
  for (ll i = I; i <= J; ++i) {
    if ((divsum[i]-i)*best < (divsum[best]-best)*i) {
      best = i;
    }
  }
  cout << best << endl;
}
```

Timus 1119. Metro

1119. Metro

Time Limit: 0.5 second
Memory Limit: 4 MB

Many of SKB Kontur programmers like to get to work by Metro because the main office is situated quite close the station Uralmash. So, since a sedentary life requires active exercises off-duty, many of the staff — Nikifor among them — walk from their homes to Metro stations on foot.



Nikifor lives in a part of our city where streets form a grid of residential quarters. All the quarters are squares with side 100 meters. A Metro entrance is situated at one of the crossroads. Nikifor starts his way from another crossroad which is south and west of the Metro entrance. Naturally, Nikifor, starting from his home, walks along the streets leading either to the north or to the east. On his way he may cross some quarters diagonally from their south-western corners to the north-eastern ones. Thus, some of the routes are shorter than others. Nikifor wonders, how long is the shortest route.

You are to write a program that will calculate the length of the shortest route from the south-western corner of the grid to the north-eastern one.

Input

There are two integers in the first line: N and M ($0 < N,M \le 1000$) — west-east and south-north sizes of the grid. Nikifor starts his way from a crossroad which is situated south-west of the quarter with coordinates (1, 1). A Metro station is situated north-east of the quarter with coordinates (N, M). The second input line contains a number K ($0 \le K \le 100$) which is a number of quarters that can be crossed diagonally. Then K lines with pairs of numbers separated with a space follow — these are the coordinates of those quarters.

Output

Your program is to output a length of the shortest route from Nikifor's home to the Metro station in meters, rounded to the integer amount of meters.

Sample

input	output
3 2 3 1 1 3 2 1 2	383

Problem Author: Leonid Volkov
Problem Source: USU Open Collegiate Programming Contest October'2001 Junior Session

```cpp
/*
Timus 1119. Metro
Dynamic Programming can be used to compute the distance from (0, 0) to each
(i, j), using the distance from (0, 0) to (i-1, j), (i, j-1), and also
(i-1, j-1) when it is possible to move diagonally from (i-1, j-1) to (i, j).

There are (n+1)x(m+1) states, but because at every step in the algorithm only 2
rows of the (n+1)x(m+1) matrix are needed, it is enough to store only the last
2 rows.
*/
#include <cmath>
#include <iostream>
#include <vector>

using namespace std;

const double INF = 1e100;

int main() {
  cout.setf(ios::fixed);
  cout.precision(0);
  int n, m, k;
  cin >> n >> m >> k;
  vector<vector<bool> > diag(n+1, vector<bool>(m+1, false));
  for (int i = 0; i < k; ++i) {
    int y, x;
    cin >> y >> x;
    diag[y][x] = true;
  }
  vector<vector<double> > dist(2, vector<double>(m+1, INF));
  for (int i = 0; i <= n; ++i) {
    for (int j = 0; j <= m; ++j) {
      if (i == 0) dist[i%2][j] = 100.0*j;
      else if (j == 0) dist[i%2][j] = 100.0*i;
      else {
        dist[i%2][j] = min(dist[(i-1)%2][j], dist[i%2][j-1])+100.0;
        if (diag[i%2][j]) dist[i%2][j] = min(dist[i%2][j], dist[(i-1)%2][j-1]+sqrt
            (20000.0));
      }
    }
  }
  cout << dist[n%2][m] << endl;
}
```

1120. Sum of Sequential Numbers

Time Limit: 1.0 second
Memory Limit: 16 MB

There is no involute formulation concerning factitiously activity of SKB Kontur in this problem. Moreover, there is no formulation at all.

Input

There is the only number N, $1 \leq N \leq 10^9$.

Output

Your program is to output two positive integers A and P separated with a space such that:

1. $N = A + (A + 1) + \ldots + (A + P - 1)$.
2. You are to choose a pair with the maximal possible value of P.

Sample

input	output
14	2 4

Problem Author: Leonid Volkov
Problem Source: USU Open Collegiate Programming Contest October'2001 Junior Session

```cpp
/*
Timus 1120. Sum of Sequential Numbers
Check for each possible value P, whether there exists an A such that
A + A+1 + ... + A+P-1 = N. Given that, fixed P, the smallest value we can
get is: 1 + 2 + ... + P (A=1), and this one must be less than or equal to N,
we can guarantee that if 1 + 2 + ... + P > N, then no solution exists
with that fixed P. There are O(sqrt(N)) different P's to check (the P's such
that: 1 + 2 + ... + P <= N). For each positive P which satisfies the inequality,
we try to find an A such that satisfies the initial condition. In order to do so
efficiently we make a binary search over the value of A. We can do so because
f(A) = A + A+1 + ... + A+P-1 is an strictly increasing function (P is fixed).
*/
#include <iostream>
using namespace std;

typedef long long ll;

ll sum(ll a) {
  return a*(a + 1)/2;
}

ll sum(ll a, ll b) {
  return sum(b) - sum(a - 1);
}

int main() {
  ll n;
  cin >> n;

  ll resA = 0, resP = 0;
  for (ll p = 1; sum(1, p) <= n; ++p) {
    ll e = 1, d = n - p + 1, r = -1;
    while (e <= d) {
      ll m = (e + d)/2;
      ll t = sum(m, m + p - 1);
      if (t < n) e = m + 1;
      else if (t > n) d = m - 1;
      else {
        r = m;
        break;
      }
    }
    if (r != -1) {
      resA = r;
      resP = p;
    }
  }
  cout << resA << " " << resP << endl;
}
```

1121. Branches

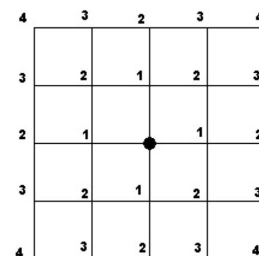
Time Limit: 1.0 second
Memory Limit: 16 MB

SKB Kontur has a lot of branches scattered all over the city. The company management decided to create a guide that will help clients to choose which of the branches they need. You are asked for help in this work.

The city is represented in the form of a grid of blocks. Each block is a square whose sides are streets and whose corners are cross-roads. We suppose that all branches are located exactly at cross-roads. The branches of SKB Kontur are of different types: service centers, warehouses, shops, training centers and so on.

Let's mark service centers with number 1, warehouses with number 2, shops with number 4, training centers with number 8 and so on. There are not more than 11 types of branches, and two branches of the same type cannot be located at the same cross-road. Each cross-road is assigned a number equal to the sum of numbers with which the branches located at this cross-road are marked. Crossroads at which there are no branches of SKB Kontur are assigned 0.

Let the distance between two cross-roads be equal to the number of street segments which one has to go from the first cross-road to the second (see picture). For example, the distance from a corner of a block to the opposite corner of this block is 2. For each cross-road at which there are no branches of SKB Kontur you have to find the sum of the numbers corresponding to the types of the branches nearest to this cross-road. For example, suppose that there are no branches at a given cross-road and at distance 1 from it, there is a branch of type 16 at distance 2, there are also two branches of type 8 and one of type 4 at distance 2 in other directions and there are no more branches at distance 2 from this cross-road. Then we should output number 28=16+8+4 for this cross-road. We do not take into consideration branches that are at distances greater than 5 from a given cross-road. Thus, if a cross-road does not have branches of SKB Kontur that are located at distances less than 6 from it then we should output 0 for this cross-road.



Input

The first line contains positive integers H and W not exceeding 150. They are numbers of "vertical" and "horizontal" streets, correspondingly. The next H lines contain W numbers each, the i-th number in the j-th line describing types of the branches located at the cross-road of the i-th "vertical" and the j-th "horizontal" street.

Output

You should output H lines containing W numbers each, the i-th number in the j-th line being equal to the sum of the numbers corresponding to the types of the branches nearest to the corresponding cross-road if there are no branches at this cross-road and −1 otherwise.

Sample

input	output
5 5 0 0 2 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 1 0 0 4 0	2 2 -1 2 -1 3 2 2 7 2 1 7 7 5 7 1 5 5 -1 5 -1 1 4 -1 4

Problem Author: Leonid Volkov, Alexander Somov
Problem Source: USU Open Collegiate Programming Contest October'2001 Junior Session

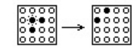
```
/*
Timus 1121. Branches
For each cross-road without any branch visit cross-roads which are at distance <= 5
from smallest to largest distance, until at some distance one or more branches are
found. Results can be computed using the logical operator OR because the IDs of
branches are powers of 2 (and therefore of the form 100..00 in binary).
*/
#include <iostream>
#include <vector>

using namespace std;

bool inside(int x, int y, int w, int h) {
  return x >= 0 && x < w && y >= 0 && y < h;
}

int main() {
  int h, w;
  cin >> h >> w;
  vector<vector<int> > b(h, vector<int>(w));
  for (int i = 0; i < h; ++i) {
    for (int j = 0; j < w; ++j) {
      cin >> b[i][j];
    }
  }
  for (int i = 0; i < h; ++i) {
    for (int j = 0; j < w; ++j) {
      if (j > 0) cout << " ";
      if (b[i][j] > 0) cout << -1;
      else {
        int c = 0;
        for (int d = 1; d <= 5 && c == 0; ++d) {
          for (int y = i-d, x = j; y < i; ++y, ++x) {
            if (inside(x, y, w, h)) c |= b[y][x];
          }
          for (int y = i, x = j+d; y < i+d; ++y, --x) {
            if (inside(x, y, w, h)) c |= b[y][x];
          }
          for (int y = i+d, x = j; y > i; --y, --x) {
            if (inside(x, y, w, h)) c |= b[y][x];
          }
          for (int y = i, x = j-d; y > i-d; --y, ++x) {
            if (inside(x, y, w, h)) c |= b[y][x];
          }
        }
        cout << c;
      }
    }
    cout << endl;
  }
}
```

Timus 1122. Game

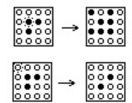
1122. Game

Time Limit: 1.0 second
Memory Limit: 16 MB

At SKB Kontur we have to work much. So there is no sin in taking a rest and playing from time to time. Consider for example the following famous one-player game.



We have a 4×4 field. There are chips with one side painted white and another side painted black on the field. Some of the chips are with their white side up and the others are with their white side down at the moment. Each move consists in turning over a chip together with all the chips that are adjacent to it vertically and horizontally (i.e. 5 chips altogether). The aim is to come to the position in which all the chips are with the same side up.



Naturally, one is easily bored with this game because interesting and unexpected positions become fewer as time goes on. That is why a modified version of the game is now more popular at SKB Kontur. In this version a move consists in turning over a fixed combination of chips within a 3×3 square. For example, a move may consist in turning over all the diagonal neighbors of a chosen chip.

The combination of chips is chosen arbitrarily; it may be assigned in the form of a 3×3 field in which the central cell corresponds to the cell at which a move as made. For example, in picture at the left the upper combination corresponds to a standard game and the lower combination is for the game described in the previous paragraph. Note that a combination can be asymmetrical. Each move is made at one of the cells of the playing field (i.e. the central cell of the 3×3 move-defining square is selected among the field's cells). Prescriptions to turn over chips at cells which are outside the 4×4 field are ignored.

In this game it would be nice to know if it is possible to reach a position in which all the chips are with the same side up and if it's possible to do this then in how many moves. You are to write a program which answers these questions.

Input

The first four lines describe the initial arrangement of chips. A symbol "W" stands for a chip which lies with its white side up and a symbol "B" stands for a chip which lies with its black side up. The next three lines describe a move: the chips that are to be turned over are shown by "1" and others are shown by "0".

Output

If it is impossible to reach the aim of the game you should output the word "Impossible", otherwise you should output the minimal number of moves necessary to come to the final position.

Sample

input	output
WWWW WBBW WBWW WWWW 101 010 101	Impossible

Problem Author: Leonid Volkov, Oleg Kats, Alexander Somov
Problem Source: USU Open Collegiate Programming Contest October'2001 Junior Session

```
/*
Timus 1122. Game
We visit all 2^(4*4) reachable states from smallest to largest number of moves and
output the first solution found. Bitmasks should be use for maximum efficiency.
*/
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

inline bool inside(int x, int y) {
  return x >= 0 && x < 4 && y >= 0 && y < 4;
}

int binary_ones(int n) {
  int ret = 0;
  for (; n > 0; n >>= 1) {
    if (n&1) ++ret;
  }
  return ret;
}

int main() {
  int b = 0;
  for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 4; ++j) {
      char c;
      cin >> c;
      if (c == 'B') b |= 1<<(4*i+j);
    }
  }
  vector<vector<bool> > m(3, vector<bool>(3));
  for (int i = 0; i < 3; ++i) {
    for (int j = 0; j < 3; ++j) {
      char c;
      cin >> c;
      if (c == '0') m[i][j] = false;
      else m[i][j] = true;
    }
  }

  vector<int> bm(1<<16);
  for (int i = 0; i < 1<<16; ++i) {
    bm[i] = i;
  }
  sort(bm.begin(), bm.end());

  int best = -1;
  for (int k = 0; k < (1<<16) && best == -1; ++k) {
    int cur = 0;
    for (int i = 0; i < 4; ++i) {
      for (int j = 0; j < 4; ++j) if ((bm[k]>>(4*i+j))&1) {
        for (int di = -1; di <= 1; ++di) {
          for (int dj = -1; dj <= 1; ++dj) if (inside(j+dj, i+di)) {
            if (m[1+di][1+dj]) cur ^= 1<<(4*(i+di)+(j+dj));
          }
        }
      }
    }
    if (b == cur || b == ((~cur)&0xFFFF)) {
```

```
      best = bm[k];
    }
  }
  if (best == -1) cout << "Impossible" << endl;
  else cout << binary_ones(best) << endl;
}
```

1124. Mosaic

Time Limit: 0.25 second
Memory Limit: 16 MB

There's no doubt that one of the most important and crucial things to do in this world is to bring up children. May be, if you study properly and reach good results at the competition you'll get a position of nanny in a kindergarten. But you are to get ready for it! Let's consider some problems that a nanny has to solve in a kindergarten.

Everyone knows the game "Mosaic". Playing the game, one is to lay out pictures of different colored pieces. Let there be M different boxes and N mosaic pieces of each of the M colors. After playing the game children rarely put the pieces back to their boxes correctly so that the color of the box and the colors of its pirces would be the same. A nanny has to do that.

Children have already put the mosaic pieces to the boxes but possibly not correctly. There are N pieces in each box. Some pieces (possibly all of them) are located in wrong boxes (i.e. boxes with pieces of a different color). Moving a hand once one can take a piece from one box to another or simply move the hand to another box. You may start from any box you like. The movement towards the first box is not taken into account. Find out the minimal number of movements one needs to put all the mosaic pieces to their boxes.

Input

The first line contains integers $2 \le M \le 500$ (the number of colors) and $2 \le N \le 50$ (the number of pieces of each color), Each of the next M lines contains N numbers in the range from 1 to M (the i+1-st line contains colors of pieces located in the i-th box). The numbers are separated with a space.

Output

the minimal possible number of hand movements that one has to make in order to take all the pieces to their boxes.

Sample

input	output
4 3 1 3 1 2 3 3 1 2 2 4 4 4	6

Problem Author: Stanislav Vasilyev
Problem Source: VI Ural State University Collegiate Programming Contest (21.10.2001)

```
/*
Timus 1124. Mosaic
Suppose we take a piece that is in a wrong box and move it to the box where it
belongs B. This means that B had some piece of the wrong color in it, and we can
take that piece and repeat the operation until at some point the gap created
during the first move in box B is filled with a piece of color B, closing the
cycle. Therefore, we just need 1 move for every piece that is in a wrong box,
but we also need additional moves to change from some box B to an unrelated box
C (i. e. there is no cycle from B that goes through C) in which there is some
piece of the wrong color. We need 1 such move for every component of related
boxes, except the first move which is not taken into account.
*/
#include <iostream>
#include <vector>

using namespace std;

void dfs(vector<vector<int> >& g, vector<int>& comp, int nd, int c) {
  comp[nd] = c;
  for (int i = 0; i < int(g[nd].size()); ++i) {
    if (comp[g[nd][i]] == -1) {
      dfs(g, comp, g[nd][i], c);
    }
  }
}

int main() {
  ios_base::sync_with_stdio(false);
  int m, n;
  cin >> m >> n;
  int ans = 0;
  vector<vector<int> > g(m);
  int ncomp = 0;
  for (int i = 0; i < m; ++i) {
    int cur = 0;
    for (int j = 0; j < n; ++j) {
      int c;
      cin >> c;
      --c;
      if (c != i) {
        g[i].push_back(c);
        ++cur;
      }
    }
    if (cur == 0) --ans;
    ans += cur;
  }
  vector<int> comp(m, -1);
  for (int i = 0; i < m; ++i) if (comp[i] == -1) {
    dfs(g, comp, i, ncomp);
    ++ncomp;
  }
  if (ans+ncomp == 0) cout << 0 << endl;
  else cout << ans+ncomp-1 << endl;
}
```

1125. Hopscotch

Time Limit: 0.25 second
Memory Limit: 16 MB

Nikifor likes to play hopscotch in the kindergarten. The playing field is a rectangle $M \times N$ partitioned into cells 1×1 meter. Nikifor hops from one cell to another possibly not adjacent cell. Each cell is colored black or white. Each time Nikifor hops into a cell, all cells whose centers are at an integer amount of meters away from Nikifor's cell center reverse their colors. You are given the final colors of the playing field cells. You also know the number of times Nikifor has been at each cell. Your task is to restore the initial colors of the cells.

Input

The first line contains two nonnegative integers M and N that do not exceed 50. The next M lines contain a character table $M \times N$. This table describes the final coloring of the field. Character 'W' denotes the white color of the cell, and 'B' denotes the black color. There are no other characters in the table. The next M lines contain a matrix with nonnegative integer elements. Each element shows how many times Nikifor has been at the corresponding cell. Numbers in the lines are separated with a space and do not exceed $2 \cdot 10^9$.

Output

should consist of M lines. The lines should contain a character table that shows the initial coloring of the playing field.

Sample

<table>
<tr><th>input</th><th>output</th></tr>
<tr><td><pre>6 6
BWBBWW
BWBBWB
BBWWBW
BBBBBW
BBWWWW
BBWBBW
2 0 12 46 2 0
3 0 0 0 0 200
4 2 1 1 4 2
4 2 1 1 4 4
0 0 0 0 0 0
2 56 24 4 2 2</pre></td><td><pre>WWBBWW
WBWWBW
WBBBBW
WBWWBW
WBWWBW
WBWWBW</pre></td></tr>
</table>

Problem Author: Dmitry Filimonenkov
Problem Source: VI Ural State University Collegiate Programming Contest (21.10.2001)

```
/*
Timus 1125. Hopscotch
The cells in which Nikifor has been an even number of times do not affect the
color of any cell, and the cells in which Nikifor has been an odd number of
times will reverse the color of the cells that are at an integer distance of
it. Simply perform the color reversal by iterating over all cells, computing
the distance to each of them, and reversing the color of the ones at integer
distance.
*/
#include <iostream>
#include <vector>
#include <cmath>
using namespace std;

typedef vector<int> Vi;
typedef vector<Vi> Mi;

int main() {
  int r, c;
  cin >> r >> c;

  Vi row(r, 0), col(c, 0);

  Mi mat(r, Vi(c));
  for (int i = 0; i < r; ++i)
    for (int j = 0; j < c; ++j) {
      char t;
      cin >> t;
      mat[i][j] = (t == 'B' ? 1 : 0);
    }

  for (int i = 0; i < r; ++i)
    for (int j = 0; j < c; ++j) {
      int t;
      cin >> t;
      if (t&1) {
        for (int x = 0; x < r; ++x) {
          for (int y = 0; y < c; ++y) {
            int p = (i - x)*(i - x) + (j - y)*(j - y);
            int q = sqrt(double(p));
            if (q*q == p) mat[x][y] ^= 1;
          }
        }
      }
    }

  for (int i = 0; i < r; ++i) {
    for (int j = 0; j < c; ++j) {
      int t = mat[i][j];
      t ^= row[i];
      t ^= col[j];
      cout << (t ? 'B' : 'W');
    }
    cout << endl;
  }
}
```

1126. Magnetic Storms

Time Limit: 0.5 second
Memory Limit: 16 MB

The directory of our kindergarten decided to be attentive to the children's health and not to let them walk outdoors during magnetic storms. Special devices that measure and display magnetic intensity were ordered. If the readout exceeded some certain level the children were told to go indoors. They disliked it because they couldn't play their games up to the end. The nannies hated it because they had to dress and undress children many times.

After a while it became clear that one could try to forecast magnetic intensity because long periods of quietude alternated with short periods of plenty of sharp peaks (this is called a magnetic storm). Hence a new modification of the devices was ordered.

The new devices were to remember the situation within several last hours and to display the maximal intensity during the period. If the intensity was low within the last 6 hours the magnetic field was regarded to be quiet; the children were let outdoors and played all the prescript time. Otherwise new peaks were probable and the children spent their time indoors.

Your task is to write a program for a new version of the device. As a matter of fact you are to solve just the main problem of modification. All the rest is already done.

You are given a number M which is length of a period (in seconds) within which peaks are to be stored and displayed. A sequence of measured magnetic intensity values is given to you as well. Each measurement is a number within the range from 0 to 100000.

Your are to output a sequence of values displayed by the device. The first number of the sequence is the maximal element of the first M input numbers, the second number is the maximal element of the 2nd, …, M+1-st input numbers and so on.

We hope that the new devices with your program won't go back on nannies and children will not walk during magnetic storms.

Input

The first line contains a number M, $1 < M \le 14000$. Then values (N integers) measured by the device follow each one in its line. There is a number −1 in the end. $M \le N \le 25000$.

Output

a sequence of readouts, each one in its line.

Sample

input	output
3 10 11 10 0 0 0 1 2 3 2 -1	11 11 10 0 1 2 3 3

Problem Author: Alexander Mironenko
Problem Source: VI Ural State University Collegiate Programming Contest (21.10.2001)

```
/*
Timus 1126. Magnetic Storms
Save the first M elements in a map or priority queue. This data structures
allows us to find efficiently the maximum among these M numbers. Move the
window one step to the right at a time by removing the first element on the
left and adding the new element to the right. This can be done in O(log(M))
at each step. The total execution time of the algorithm is O(N*log(M)),
where N is the amount of numbers in the input, and M is the length of the
window.
*/
#include <iostream>
#include <vector>
#include <queue>
#include <map>
using namespace std;

#define PB push_back

typedef vector<int> Vi;
typedef priority_queue<int> PQ;
typedef map<int, int> MAP;

int main() {
  int m;
  cin >> m;

  Vi v;
  int t;
  while (cin >> t and t != -1) v.PB(t);
  int n = v.size();

  PQ q;
  for (int i = 0; i < m; ++i) q.push(v[i]);

  cout << q.top() << endl;

  MAP mp;
  for (int i = m; i < n; ++i) {
    ++mp[v[i - m]];
    while (mp[q.top()] > 0) {
      --mp[q.top()];
      q.pop();
    }
    q.push(v[i]);
    cout << q.top() << endl;
  }
}
```

1127. Colored Bricks

Time Limit: 0.4 second
Memory Limit: 16 MB

There are lots of cubic bricks in the kindergarten. Children like to build toy brick towers and then to drop them. It is clear that the higher tower has been built the more interesting it is to drop it. The tower is built by placing bricks one onto another and aligning their sides. The tower is based on one brick. Thus the height of a tower is the number of the bricks it is built of. Each side of a brick is painted in one color. So the kids build colored towers. In order to train the children's sense of beauty nannies teach them to build the towers in such a way that each side of the tower would be one-color. Thus the kids would like to build a tower with one-color sides as high as possible.

Every nanny can easily solve this problem. Try your best to do it as well.

Input

The first line contains a number N ($1 < N \le 10^3$) — the number of bricks. The next N lines contain descriptions of bricks. Each brick is described with a string of 6 capital latin letters denoting the color of the corresponding side (A — Azure, B — Blue, C — Cyan, G — Green, O — Orange, R — Red, S — Scarlet, V — Violet, W — White, Y — Yellow). The colors of the sides are given in the following order: *front, right, left, rear, top, bottom*. A brick never has two sides of the same color.

Output

Output the only number — the maximal height of a toy tower that can be built of the given brick set.

Sample

input	output
4 GYVABW AOCGYV CABVGO OVYWGA	3

Problem Author: Ekaterina Vasilyeva
Problem Source: VI Ural State University Collegiate Programming Contest (21.10.2001)

```
/*
Timus 1127. Colored Bricks
For each cube in the input, rotate it through all possible orientations,
and keep track, for each orientation, of the strip formed by the colors on
the front, back, left and right faces. Calculate, for each possible strip,
how many cubes of the input can produce it. The strip that can be produced
by the most amount of cubes will determine the colors in the final building,
and the height of that building is the number of cubes which can produce
that strip.
*/
#include <iostream>
#include <string>
#include <set>
#include <vector>
using namespace std;

typedef vector<int> Vi;
typedef set<int> SET;
typedef SET::iterator Sit;

string R1(string s) {
  char c0 = s[0];
  s[0] = s[2];
  s[2] = s[3];
  s[3] = s[1];
  s[1] = c0;
  return s;
}

string R2(string s) {
  char c0 = s[0];
  s[0] = s[4];
  s[4] = s[3];
  s[3] = s[5];
  s[5] = c0;
  return s;
}

int num(char c) {
  if (c == 'A') return 0;
  if (c == 'B') return 1;
  if (c == 'C') return 2;
  if (c == 'G') return 3;
  if (c == 'O') return 4;
  if (c == 'R') return 5;
  if (c == 'S') return 6;
  if (c == 'V') return 7;
  if (c == 'W') return 8;
  return 9;
}

int main() {
  int n;
  cin >> n;

  Vi v(10000, 0);

  for (int i = 0; i < n; ++i) {
    string s;
    cin >> s;
```

```
    SET st;
    for (int r2 = 0; r2 < 4; ++r2) {
      for (int r1 = 0; r1 < 4; ++r1) {
        int t = num(s[0]) + 10*num(s[1]) + 100*num(s[3]) + 1000*num(s[2]);
        st.insert(t);
        s = R1(s);
      }
      s = R2(s);
    }
    s = R1(s);
    s = R2(s);
    for (int r1 = 0; r1 < 4; ++r1) {
      int t = num(s[0]) + 10*num(s[1]) + 100*num(s[3]) + 1000*num(s[2]);
      st.insert(t);
      s = R1(s);
    }
    s = R2(s);
    s = R2(s);
    for (int r1 = 0; r1 < 4; ++r1) {
      int t = num(s[0]) + 10*num(s[1]) + 100*num(s[3]) + 1000*num(s[2]);
      st.insert(t);
      s = R1(s);
    }

    for (Sit it = st.begin(); it != st.end(); ++it)
      ++v[*it];
  }

  int res = 0;
  for (int i = 0; i < 10000; ++i)
    res = max(res, v[i]);

  cout << res << endl;
}
```

1128. Partition into Groups

Time Limit: 0.5 second
Memory Limit: 16 MB

There are N children in the kindergarten. Unfortunately, the children quarrel though not often. Each child has not more than three adversaries. Is it possible to partition the children into two groups (possibly not equal), so that each child would have not more than one adversary in his or her group?

Input

The first line contains an integer N, $0 < N \leq 7163$. The next N lines contain lists of adversaries of each child. A line starts with the amount of the corresponding child's adversaries, then the numbers of the adversaries follow. The numbers in each line are separated with a space.

Output

The first line contains the number of children in the smaller group. The next line contains the list of children in the group. The numbers in the second line are separated with a space. If the groups are of the same size then you are to describe the group that contains the child number one. Note that the output may contain the only number 0. If there are several possible partitions it's sufficient to output an arbitrary one. If there's no possible partition you are to output the only string “NO SOLUTION”.

Sample

input	output
8 3 2 3 7 3 1 3 7 3 1 2 7 1 6 0 2 4 8 3 1 2 3 1 6	4 1 2 5 6

Problem Author: Dmitry Filimonenkov
Problem Source: VI Ural State University Collegiate Programming Contest (21.10.2001)

```
/*
Timus 1128. Partition into Groups
Assign each children to a group arbitrarily, then repeatedly change a child who is
in the same group as 2 or 3 of his adversaries to the other group until every child
is in the same group as at most 1 of his adversaries. This will always happen in a
O(N) number of steps because at each step the number of pairs of adversaries which
are in opposite groups increases, and it can't go over 3*N/2. Each step can be
performed in O(log N) time.
*/
#include <iostream>
#include <set>
#include <vector>

using namespace std;

int get_gadv(vector<set<int> >& children, int child) {
  for (int i = 0; i < int(children.size()); ++i) {
    if (children[i].count(child) > 0) return i;
  }
  return -1;
}

int main() {
  int n;
  cin >> n;
  vector<vector<int> > adv(n);
  for (int i = 0; i < n; ++i) {
    int a;
    cin >> a;
    adv[i] = vector<int>(a);
    for (int j = 0; j < a; ++j) {
      cin >> adv[i][j];
      --adv[i][j];
    }
  }
  vector<int> side(n, 0);
  vector<set<int> > children(4);
  for (int i = 0; i < n; ++i) {
    children[adv[i].size()].insert(i);
  }
  for (bool done = false; !done;) {
    int gadv = 3;
    for (; children[gadv].size() == 0;) --gadv;
    if (gadv < 2) done = true;
    else {
      int child = *children[gadv].begin();
      children[gadv].erase(child);
      side[child] = 1-side[child];
      for (int i = 0; i < int(adv[child].size()); ++i) {
        int agadv = get_gadv(children, adv[child][i]);
        children[agadv].erase(adv[child][i]);
        if (side[adv[child][i]] != side[child]) {
          children[agadv-1].insert(adv[child][i]);
        }
        else {
          children[agadv+1].insert(adv[child][i]);
        }
      }
      children[adv[child].size()-gadv].insert(child);
    }
  }
```

```
  int siz0 = 0;
  for (int i = 0; i < n; ++i) {
    if (side[i] == 0) ++siz0;
  }
  int group;
  if (siz0 < n-siz0 || (siz0 == n-siz0 && side[0] == 0)) {
    cout << siz0 << endl;
    group = 0;
  }
  else {
    cout << n-siz0 << endl;
    group = 1;
  }
  bool first = true;
  for (int i = 0; i < n; ++i) if (side[i] == group) {
    if (first) first = false;
    else cout << " ";
    cout << i+1;
  }
  cout << endl;
}
```

Timus 1129. Door Painting

1129. Door Painting

Time Limit: 0.25 second
Memory Limit: 16 MB

There are many rooms, corridors and doors between them in the kindergarten. Some repairs are planned to be made soon. The doors are agreed to be painted in bright cheerful colors: green and yellow. The matron of the kindergarten wants the doors to satisfy the following condition: the sides of an arbitrary door must have the different colors. The number of green doors in each of the lodgings must differ from the number of yellow doors not more than by one. Given the plan of the kindergarten suggest your scheme of door painting.

Input

The first line contains the number of lodgings $N \le 100$ in the kindergarten. The next N lines contain description of the door configuration (k+1-st line contains a description of the k-th lodging). Each of the N lines starts with the number of doors that connect this lodging with adjacent ones. Then there are numbers of adjacent lodgings separated with a space (these numbers follow in ascending order).

Output

should contain a required painting scheme or the word "Impossible" if it is impossible to satisfy the requirements. The colors of the K-th room doors should be put in the K-th line in the same order as they were in the input data. The green color is denoted by G, yellow — by Y.

Sample

input	output
5 3 2 3 4 3 1 3 5 4 1 2 4 5 3 1 3 5 3 2 3 4	G Y G Y G Y G Y Y G Y G G G Y Y

Problem Author: Magaz Asanov
Problem Source: VI Ural State University Collegiate Programming Contest (21.10.2001)

```
/*
Timus 1129. Door Painting
Repeatedly perform the following routine:
Pick one room that has some unpainted door and start paths from it always going
out through unpainted doors and painting them with the less abundant color in
the starting room (or any if both colors are equally abundant), until a room in
which all doors are painted is reached. Each path preserves the color difference
of all the rooms except the first one and the last one. After each path the
color difference of the starting room will not be above 1 because of the
criteria to choose the color. The last room will never be visited again, and its
color difference will never be above 1 because either:
a) The last room is the first room, and after the path its difference remains
the same.
b) The last room is not the first room, and its difference before the path was
0, so it can be at most 1 (+1 or -1) after the path.
*/
#include <iostream>
#include <map>
#include <set>
#include <vector>

using namespace std;

int main() {
  int n;
  cin >> n;
  vector<set<int> > g(n);
  vector<map<int, int> > sol(n);
  for (int i = 0; i < n; ++i) {
    int e;
    cin >> e;
    for (int j = 0; j < e; ++j) {
      int nd;
      cin >> nd;
      --nd;
      g[i].insert(nd);
    }
  }
  vector<int> cnt(n, 0);
  set<int> has_edges;
  for (int i = 0; i < n; ++i) {
    if (g[i].size() > 0) has_edges.insert(i);
  }
  for (; has_edges.size() > 0;) {
    int ini = *has_edges.begin();
    int sign = 1;
    if (cnt[ini] > 0) sign = -1;
    int cur = ini;
    for (; g[cur].size() > 0;) {
      int nxt = *g[cur].begin();
      cnt[cur] += sign;
      sol[cur][nxt] = sign;;
      cnt[nxt] -= sign;
      sol[nxt][cur] = -sign;;
      g[cur].erase(nxt);
      if (g[cur].size() == 0) has_edges.erase(cur);
      g[nxt].erase(cur);
      if (g[nxt].size() == 0) has_edges.erase(nxt);
      cur = nxt;
    }
  }
```

```
  for (int i = 0; i < n; ++i) {
    for (map<int, int>::iterator it = sol[i].begin(); it != sol[i].end(); ++it) {
      if (it != sol[i].begin()) cout << " ";
      if (it->second == 1) cout << "Y";
      else cout << "G";
    }
    cout << endl;
  }
}
```

Timus 1133. Fibonacci Sequence

1133. Fibonacci Sequence

Time Limit: 1.0 second
Memory Limit: 16 MB

$\{F_i\}_{-\infty}^{+\infty}$ is an infinite sequence of integers that satisfies to Fibonacci condition $F_{i+2} = F_{i+1} + F_i$ for any integer i. Write a program, which calculates the value of F_n for the given values of F_i and F_j.

Input

The input contains five integers in the following order: i, F_i, j, F_j, n.
$-1000 \le i, j, n \le 1000$, $i \ne j$,
$-2 \cdot 10^9 \le F_k \le 2 \cdot 10^9$ ($k = \min(i, j, n), \ldots, \max(i, j, n)$).

Output

The output consists of a single integer, which is the value of F_n.

Sample

input	output
`3 5 -1 4 5`	`12`

Hint

In the example you are given: $F_3 = 5$, $F_{-1} = 4$; you asked to find the value of F_5. The following Fibonacci sequence can be reconstructed using known values:

$$\ldots, F_{-1} = 4, F_0 = -1, F_1 = 3, F_2 = 2, F_3 = 5, F_4 = 7, F_5 = 12, \ldots$$

Thus, the answer is: $F_5 = 12$.

Problem Source: Quarterfinal, Central region of Russia, Rybinsk, October 17-18 2001

```
/*
Timus 1133. Fibonacci Sequence
Sigui M=[[1,1],[1,0]] (matriu 2x2, format: matriu=[fila1,fila2]), suposem i < j
Aleshores M^(j - i) * [F(i),F(i-1)] = [F(j),F(j-1)]. Coneixem M, i, j, F(i) i F(j),
tenim informacio suficient per trobar F(i-1). En la solucio implementada, F(i-1)
s'obte de la divisio de dos nombres que poden ser molt grans, per aixo hem fet
servir BigInt (llibreria per fer operacions amb nombres arbitrariament grans).
Un cop coneixem F(i-1) i F(i), ja es trivial calcular qualsevol altre valor de la
sequencia. Caldra, potser, distingir 2 casos segons si n<i o n>=i.
*/
#include <iostream>
#include <vector>
#include <sstream>
#include <iomanip>
using namespace std;

typedef vector<int> VE;
typedef long long ll;
const ll base = 1000000000;

class BI { public:
  VE v;     // el digit menys significatiu es v[0]
  int n;    // v.size()
  int sig; // 1: positiu o zero, -1: negatiu

  // GENERAL PER A TOT
  void zero() { v = VE(1, 0); n = sig = 1; }
  void remida(int i, int que=0) { v.resize(n = i, que); }

  void treuzeros(){
    while (--n and !v[n]);
    remida(n+1);
    if (n == 1 and !v[0]) sig = 1;
  }

  void parse(string num) {
    if (num[0] == '-') { sig = -1; num = num.substr(1); }
      else sig = 1;
    int m = num.size() - 1;
    v = VE(1 + m/9, 0); n = v.size();
    for (int i = m, exp = 0, pw = 1, pos = 0; i >= 0; --i, ++exp, pw *= 10) {
      if (exp == 9) { exp = 0; pw = 1; ++pos; }
      v[pos] += (num[i] - '0')*pw;
    }
    treuzeros();
  }

  // DESIGUALTATS
  // compara en valor absolut, 0: == b, positiu: > b; negatiu: < b
  inline int compabs(const BI &b) const {
    if (n != b.n) return n - b.n;
    for (int i = n - 1; i >= 0; --i)
      if (v[i] != b.v[i]) return v[i] - b.v[i];
    return 0;
  }

  // 0: == b, positiu: > b; negatiu: < b
  inline int compara(const BI &b) const {
    if (sig != b.sig) return sig - b.sig;
    return sig*compabs(b);
  }
```

```
inline bool operator==(const BI &b) const { return !compara(b); }
inline bool operator!=(const BI &b) const { return compara(b); }
inline bool operator<(const BI &b)  const { return compara(b) < 0; }
inline bool operator<=(const BI &b) const { return compara(b) <= 0; }
inline bool operator>(const BI &b)  const { return compara(b) > 0; }
inline bool operator>=(const BI &b) const { return compara(b) >= 0; }

// $|x| < 10^9$
inline bool operator==(int x) { return n == 1 and sig*v[0] == x; }

// OPERACIONS EN VALOR ABSOLUT
inline int dig(int i) { return (i < n ? v[i] : 0); }

void suma(BI &b) {
  if (n < b.n) remida(b.n, 0);
  int ca = 0;
  for (int i = 0; i < n; ++i) {
    v[i] += b.dig(i) + ca; ca = v[i]/base; v[i] %= base;
  }
  if (ca) remida(n+1, ca);
}

// suposa >= b
void resta(BI &b) {
  int ca = 0;
  for (int i = 0; i < n; ++i) {
    v[i] += base - b.dig(i) + ca; ca = v[i]/base - 1; v[i] %= base;
  }
  treuzeros();
}

// INICIALITZACIONS I COPIA
BI() { zero(); }
BI(const BI &b) { *this = b; }

// $|x| < 10^{18}$
BI(ll x) {
  sig = (x < 0 ? -1 : 1); x *= sig;
  if (x < base) { v = VE(1, x); n = 1; }
    else { v = VE(2); v[0] = x % base; v[1] = x/base; n = 2; }
}

BI(string num) { parse(num); }
void operator=(const BI &b) { v = b.v; n = b.n; sig = b.sig; }

// OPERACIONS
void operator+=(BI &b) {
  if (sig == b.sig) return suma(b);
  if (compabs(b) >= 0) return resta(b);
  BI aux(b); aux.resta(*this); *this = aux;
}

void operator-=(BI &b) {
  if (&b == this) return zero();
  b.sig *= -1; operator+=(b); b.sig *= -1;
}

// $|x| < 10^9$
void operator*=(int x) {
  if (x < 0) { sig *= -1; x *= -1; }
```

```
    remida(n + 1, 0);
    ll ca = 0;
    for (int i = 0; i < n; ++i) {
      ca += (ll)v[i]*x; v[i] = ca % base; ca /= base;
    }
    treuzeros();
  }

  void operator*=(BI &b) {
    BI c; c.remida(n + b.n, 0); c.sig = sig*b.sig;
    for (int i = 0; i < n; ++i)
      for (int j = 0; j < b.n; ++j) {
        int k = i + j;
        ll z = (ll)v[i]*b.v[j] + c.v[k], y;
        while ((y = z / base)) { c.v[k] = z % base; z = y + c.v[++k]; }
        c.v[k] = z;
      }
      c.treuzeros();
      *this = c;
  }

  // $|x| < 10^9$
  void operator/=(int x) {
    if (x < 0) { sig *= -1; x *= -1; }
    ll ca = 0;
    for (int i = n-1; i >= 0; --i ) {
      ca += v[i]; v[i] = ca/x; ca %= x; ca *= base;
    }
    treuzeros();
  }

  void operator/=(BI &b){
    if (compabs(b)<0) return zero();
    if (b.n==1) *this/=b.sig*b.v[0];
    else {
      int st = sig*b.sig, sb = b.sig; sig = b.sig = 1;
      vector<BI> VB,pot2;
      VB.push_back(b); pot2.push_back(1);
      BI curr=0;
      //primera pasada
      while (VB[VB.size()-1]<=*this){
        BI ultimo=VB[VB.size()-1];  ultimo+=ultimo; VB.push_back(ultimo);
        ultimo=pot2[pot2.size()-1]; ultimo+=ultimo; pot2.push_back(ultimo);
      }
      curr+=pot2[pot2.size()-2]; (*this)-=VB[VB.size()-2];
      //resto
      while ((*this)>=b){
        int i=0;
        while (VB[i]<=(*this)) i++;
        curr+=pot2[i-1]; (*this)-=VB[i-1];
      }
      (*this)=curr; sig = st; b.sig = sb;
    }
  }

  // $|x| < 10^9$; amb negatius funciona com C++
  ll mod(int x) {
    if (x < 0) x *= -1;
    ll ca = 0;
    for (int i = n-1; i >= 0; --i ) { ca *= base; ca += v[i]; ca %= x; }
    return ca;
```

```
  }

  void operator%=(BI &b) {
    BI r(*this); r /= b; r*= b; operator-=(r);
  }

  friend ostream &operator<<(ostream &out, BI &b) {
    if (b.sig < 0) out << '-';
    int i = b.v.size() - 1;
    out << b.v[i];
    for (--i; i >= 0; --i) out << setw(9) << setfill('0') << b.v[i];
    return out;
  }
};

typedef vector<BI> Vbi;
typedef vector<Vbi> Mbi;
typedef stringstream SS;

Mbi mult(Mbi a, Mbi b) {
  Mbi c(4, Vbi(4, 0));
  for (int i = 0; i < 2; ++i)
    for (int j = 0; j < 2; ++j)
      for (int k = 0; k < 2; ++k) {
        BI t = a[i][k];
        t *= b[k][j];
        c[i][j] += t;
      }
  return c;
}

Mbi eleva(Mbi m, int e) {
  if (e == 0) {
    Mbi r(2, Vbi(2, 0));
    r[0][0] = 1;
    r[1][1] = 1;
    return r;
  }
  Mbi r = eleva(m, e/2);
  r = mult(r, r);
  if (e&1) r = mult(m, r);
  return r;
}

int main() {
  ll i, fi, j, fj, n;
  cin >> i >> fi >> j >> fj >> n;

  if (i > j) {
    swap(i, j);
    swap(fi, fj);
  }

  Mbi mat(2, Vbi(2, 1));
  mat[1][1] = 0;
  mat = eleva(mat, j - i);

  BI a = mat[0][0];
  BI b = mat[0][1];

  BI t1 = a;
```

```
  t1 *= fi;
  BI t2 = fj;
  t2 -= t1;
  t2 /= b;

  SS ss;
  ss << t2;
  ll x;
  ss >> x;

  if (i <= n) {
    ll f1 = x;
    ll f2 = fi;
    for (int k = i; k < n; ++k) {
      ll f3 = f1 + f2;
      f1 = f2;
      f2 = f3;
    }
    cout << f2 << endl;
  }
  else {
    ll f1 = x;
    ll f2 = fi;
    for (int k = i; k > n; --k) {
      ll f0 = f2 - f1;
      f2 = f1;
      f1 = f0;
    }
    cout << f2 << endl;
  }
}
```

1134. Cards

Time Limit: 1.0 second
Memory Limit: 16 MB

Each of the n cards has numbers written on the both sides of it. The first card has 0 and 1 on it, the second has 1 and 2, …, the n-th has (n−1) and n. First-grade pupil Nick takes cards one by one in random order and reads the number on one of the sides. Nick is not very good with numbers, so it is possible that he makes a mistake. Your task is to find out if he was mistaken, i.e. if the given sequence of numbers is possible for some order of taking cards.

Input

The first line contains numbers n, the total number of cards, and m, the number of the cards that were taken. Starting with the second line, the m non-negative integers are listed (the sequence read by Nick). One or more spaces or line feeds separate the numbers.
$1 \le n \le 1000$

Output

Write YES if the given sequence of numbers is possible for some order of taking cards, NO otherwise.

Sample

input	output
`5 4` `2 0 1 2`	`NO`

Problem Source: Quarterfinal, Central region of Russia, Rybinsk, October 17-18 2001

```
/*
Timus 1134. Cards
Sort the numbers and check from left to right if it is a valid sequence by
assuming that in case of having two possible cards the leftmost one is used.
*/
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main() {
  int n, m;
  cin >> n >> m;
  vector<int> c(m);
  bool ok = true;
  for (int i = 0; i < m && ok; ++i) {
    cin >> c[i];
    if (c[i] < 0 || c[i] > n) ok = false;
  }
  if (ok && m > 0) {
    sort(c.begin(), c.end());
    int last = max(0, c[0]-1);
    for (int i = 1; i < m && ok; ++i) {
      if (last < c[i]-1) last = c[i]-1;
      else if (last < c[i] && c[i] < n) last = c[i];
      else ok = false;
    }
  }
  if (ok) cout << "YES" << endl;
  else cout << "NO" << endl;
}
```

1135. Recruits

Time Limit: 1.0 second
Memory Limit: 16 MB

N recruits are standing in front of a sergeant who orders to turn left. Some of the soldiers turn left, while the others turn right. In a second each recruit seeing the face of another recruit understands that a mistake was made and turns around. This happens at the same time to each pair of soldiers facing each other. The process continues until the formation becomes stable. Write a program, which finds out the number of times when a pair of soldiers turned around. If the process is infinite then the program should write the word “NO”.

Example:
Legend:
‘<’: a recruit facing left;
‘>’: a recruit facing right.

Formation	Comments	Number of turns
> > < < > <	Initial formation	2
> < > < < >	One second has passed	2
< > < > < >	Two seconds have passed	2
< < > < > >	Three seconds have passed	1
< < < > > >	Final formation	Total: 7

Input

The first line contains the number of recruits (*N*). The rest of the input contains only ‘<’, ‘>’ and line break characters. There is exactly *N* ‘<’ and ‘>’ characters in the input. Each line of the input may have up to 255 characters.
$1 \leq N \leq 30000$.

Output

Write the number of turns.

Sample

input	output
6 >><<><	7

Problem Source: Quarterfinal, Central region of Russia, Rybinsk, October 17-18 2001

```
/*
Timus 1135. Recruits
Un soldat que mira cap a la dreta es girara tantes vegades com soldats hi hagi a la
seva dreta mirant cap a l'esquerra. El cas dels soldats que miren a l'esquerra
es analeg. En la solucio implementada, comptem quantes vegades es giren els soldats
que miren a l'esquerra i aixo mateix es la solucio (d'aquesta forma estem comptant
tots els girs sense repetits).
*/
#include <iostream>
#include <string>
using namespace std;

typedef long long ll;

int main() {
  int n;
  cin >> n;

  string s, t;
  while (cin >> t) s += t;

  ll total = 0;
  int r = 0;
  for (int i = 0; i < n; ++i) {
    if (s[i] == '>') ++r;
    else total += r;
  }
  cout << total << endl;
}
```

Timus 1136. Parliament

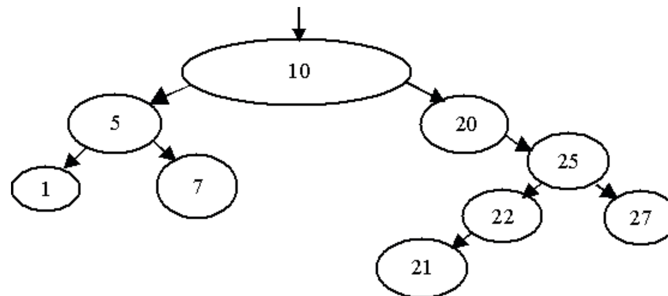
1136. Parliament

Time Limit: 1.0 second

Memory Limit: 16 MB

A new parliament is elected in the state of MMMM. Each member of the parliament gets his unique positive integer identification number during the parliament registration. The numbers were given in a random order; gaps in the sequence of numbers were also possible. The chairs in the parliament were arranged resembling a tree-like structure. When members of the parliament entered the auditorium they took seats in the following order. The first of them took the chairman's seat. Each of the following delegates headed left if his number was less than the chairman's, or right, otherwise. After that he took the empty seat and declared himself as a wing chairman. If the seat of the wing chairman has been already taken then the seating algorithm continued in the same way: the delegate headed left or right depending on the wing chairman's identification number.

The figure below demonstrates an example of the seating of the members of parliament if they entered the auditorium in the following order: 10, 5, 1, 7, 20, 25, 22, 21, 27.



During its first session the parliament decided not to change the seats in the future. The speech order was also adopted. If the number of the session was odd then the members of parliament spoke in the following order: the left wing, the right wing and the chairman. If a wing had more than one parliamentarian then their speech order was the same: the left wing, the right wing, and the wing chairman. If the number of the session was even, the speech order was different: the right wing, the left wing, and the chairman. For a given example the speech order for odd sessions will be 1, 7, 5, 21, 22, 27, 25, 20, 10; while for even sessions — 27, 21, 22, 25, 20, 7, 1, 5, 10.

Determine the speech order for an even session if the speech order for an odd session is given.

Input

The first line of the input contains *N*, the total number of parliamentarians. The following lines contain *N* integer numbers, the identification numbers of the members of parliament according to the speech order for an odd session.

The total number of the members of parliament does not exceed 3000. Identification numbers do not exceed 65535.

Output

The output should contain the identification numbers of the members of parliament in accordance with the speech order for an even session.

Sample

input	output
`9` `1 7 5 21 22 27 25 20 10`	`27 21 22 25 20 7 1 5 10`

Problem Source: Quarterfinal, Central region of Russia, Rybinsk, October 17-18 2001

```
/*
Timus 1136. Parliament
For any given subtree described in the left-right-mid order the last element is
the root (mid), and because it is a binary search tree the left and right subtrees
can be identified as the subsequences of elements that are smaller and larger,
respectively, than the root. Solve for each subtree recursively.
*/
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

void right_left_mid(vector<int>& p, int a, int b, bool first) {
  if (b == a) return;
  if (b-a == 1) {
    if (!first) cout << " ";
    cout << p[a];
    return;
  }
  int r = lower_bound(p.begin()+a, p.begin()+b-1, p[b-1])-p.begin();
  right_left_mid(p, r, b-1, first);
  right_left_mid(p, a, r, false);
  cout << " " << p[b-1];
}

int main() {
  int n;
  cin >> n;
  vector<int> p(n);
  for (int i = 0; i < n; ++i) {
    cin >> p[i];
  }
  right_left_mid(p, 0, n, true);
  cout << endl;
}
```

1137. Bus Routes

Time Limit: 1.0 second
Memory Limit: 16 MB

Several bus routes were in the city of Fishburg. None of the routes shared the same section of road, though common stops and intersections were possible. Fishburg old residents stated that it was possible to move from any stop to any other stop (probably making several transfers). The new mayor of the city decided to reform the city transportation system. He offered that there would be only one route going through all the sections where buses moved in the past. The direction of movement along the sections must be the same and no additional sections should be used.

Write a program, which creates one of the possible new routes or finds out that it is impossible.

Input

The first line of the input contains the number of old routes n. Each of the following n lines contains the description of one route: the number of stops m and the list of that stops. Bus stops are identified by positive integers not exceeding 10000. A route is represented as a sequence of $m + 1$ bus stop identifiers: $l_1, l_2, \ldots, l_m, l_{m+1} = l_1$ that are sequentially visited by a bus moving along this route. A route may be self-intersected. A route always ends at the same stop where it starts (all the routes are circular).

The number of old routes: $1 \le n \le 100$.
The number of stops: $1 \le m \le 1000$.
The number-identifier of the stop: $1 \le l \le 10000$.

Output

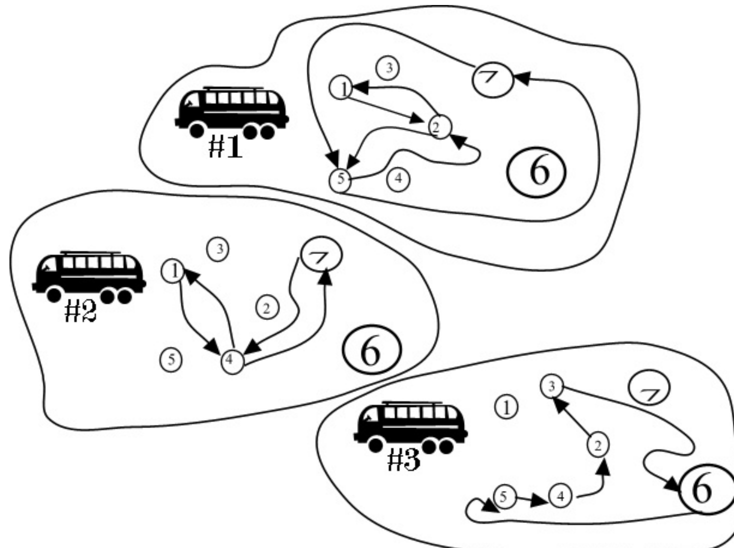
The output contains the number of stops in the new route k and the new route itself in the same format as in the input. The last (k+1)-th stop must be the same as the first. If it is impossible to make a new route according to the problem statement then write 0 (zero) to the output.

Sample

input	output
3 6 1 2 5 7 5 2 1 4 1 4 7 4 1 5 2 3 6 5 4 2	15 2 5 4 2 3 6 5 7 4 1 2 1 4 7 5 2

Hint

Here is a picture for the example:



Problem Source: Quarterfinal, Central region of Russia, Rybinsk, October 17-18 2001

```
/*
Timus 1137. Bus Routes
Ens donen un graf dirigit. Si ignorem els nodes que no tenen arestes
adjacents/incidents, el graf es connex (ens ho garanteix l'enunciat).
A mes, com que el graf es unio de cicles, tenim que tot node te el mateix
nombre d'arestes adjacents i incidents. Amb l'algorisme d'Euler sempre podem
trobar un cicle euleria que recorri totes les arestes del graf, i aixo
es una solucio. L'algorisme d'Euler es lineal respecte el nombre d'arestes
del graf. L'algorisme implementat aqui te cost O(n*logn), per la forma en
que explorem les arestes adjacents a un node, que es prou eficient donades
les restriccions del problema.
*/
#include <iostream>
#include <stack>
#include <vector>
#include <map>
using namespace std;

#define PB push_back

typedef stack<int> STACK;
typedef vector<int> Vi;
typedef map<int, int> MAP;

Vi net[11000];
MAP mat[11000];
int on[11000];
int tour[110000];

int main() {
  int n;
  cin >> n;
  for (int i = 0; i < n; ++i) {
    int m;
    cin >> m;

    int a;
    cin >> a;
    --a;
    for (int j = 0; j < m; ++j) {
      int t;
      cin >> t;
      --t;

      net[a].PB(t);
      ++mat[a][t];

      a = t;
    }
  }

  STACK st;
  for (int i = 0; i < 10000; ++i)
    if (net[i].size()) {
      st.push(i);
      break;
    }

  int s = 0;
  while (not st.empty()) {
    int nd = st.top();
```

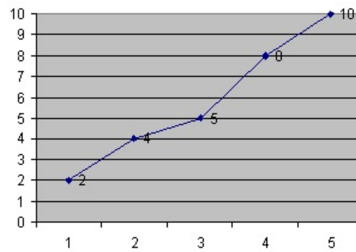
```
    st.pop();

    int sz = net[nd].size();
    while (on[nd] < sz and mat[nd][net[nd][on[nd]]] == 0) ++on[nd];

    if (on[nd] == sz) tour[s++] = nd;
    else {
      --mat[nd][net[nd][on[nd]]];
      st.push(nd);
      st.push(net[nd][on[nd]++]);
    }
  }

  cout << s - 1;
  for (int i = 0; i < s; ++i) cout << " " << tour[s - i - 1] + 1;
  cout << endl;
}
```

1138. Integer Percentage

Time Limit: 1.0 second
Memory Limit: 16 MB

Applying for a new job, programmer N. Smart required that his new salary (in rubles, positive integer) would be greater than his previous salary by integer percentage. What could be the highest possible number of previous jobs for mister Smart, if his latest salary did not exceed n rubles and his first salary was exactly s rubles?

Example. Let $n = 10$, $s = 2$, then $m = 5$. The sequence 2, 4 (+100%), 5 (+25%), 8 (+60%), 10 (+25%) is the longest (although not unique) sequence that satisfies to the problem statement. Salary increase percentage is written inside the brackets.



Input

Two integers n and s separated by one or more spaces. $1 \leq n, s \leq 10000$.

Output

A single integer m — the maximum number of N. Smart's previous jobs.

Sample

input	output
10 2	5

Hint

if $n = s$, the answer is 1.

Problem Source: Quarterfinal, Central region of Russia, Rybinsk, October 17-18 2001

```cpp
/*
Timus 1138. Integer Percentage
Compute the highest possible number of previous jobs for each salary in [s, n]
using dynamic programming and output the maximum.
*/
#include <iostream>
#include <vector>

using namespace std;

int main() {
  int n, s;
  cin >> n >> s;
  vector<int> jobs(n+1, 0);
  jobs[s] = 1;
  int ans = 0;
  for (int i = s; i < int(jobs.size()); ++i) if (jobs[i] > 0) {
    ans = max(ans, jobs[i]);
    for (int p = 1; p <= 100 && i*(100+p)/100 < int(jobs.size()); ++p) {
      if ((i*(100+p))%100 == 0) {
        jobs[i*(100+p)/100] = max(jobs[i*(100+p)/100], jobs[i]+1);
      }
    }
  }
  cout << ans << endl;
}
```

1139. City Blocks

Time Limit: 1.0 second

Memory Limit: 16 MB

The blocks in the city of Fishburg are of square form. N avenues running south to north and M streets running east to west bound them. A helicopter took off in the most southwestern crossroads and flew along the straight line to the most northeastern crossroads. How many blocks did it fly above?

Note. A block is a square of minimum area (without its borders).

Input

The input contains N and M separated by one or more spaces. $1 < N, M < 32000$.

Output

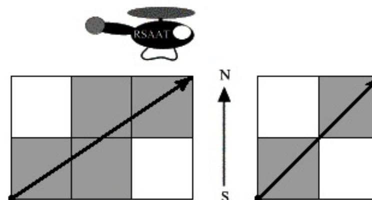
The number of blocks the helicopter flew above.

Samples

input	output
4 3	4
3 3	2

Hint

The figures for samples:

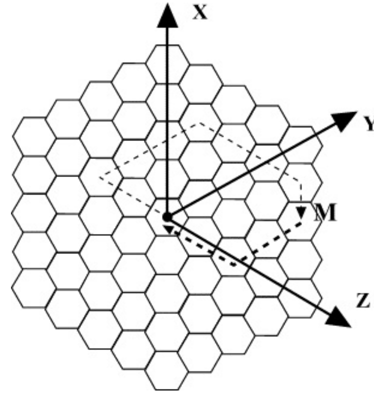


Problem Source: Quarterfinal, Central region of Russia, Rybinsk, October 17-18 2001

```
/*
Timus 1139. City Blocks
Te solution for (n, m) is the same than for (m, n), therefore we can swap n and m
if necessary so that m <= n. The slope of the line is <= 1. Add the number of
blocks that intersect with the line at each row (it can be either 1 or 2 per row).
*/
#include <iostream>

using namespace std;

int main() {
  int n, m;
  cin >> n >> m;
  if (n == m) cout << n-1 << endl;
  else {
    if (m > n) swap(n, m);
    int ans = 0;
    for (int i = 0; i < n-1; ++i) {
      int hl = i*(m-1)/(n-1);
      int hr = (i+1)*(m-1)/(n-1);
      if (hl == hr) ++ans;
      else ans += 2;
      if (((i+1)*(m-1))%(n-1) == 0) --ans;
    }
    cout << ans << endl;
  }
}
```

1140. Swamp Incident

Time Limit: 1.0 second
Memory Limit: 16 MB



An attempt was made to count all the cranberries in the swamps located in the surroundings of Rybinsk. It appeared convenient to split the surface of the swamp into regular hexagonal cells. One of these cells was considered as the central one, where the helicopter hovered. Three directions were marked (see picture).

After that a hired student landed in the central cell and wandered around for a certain time. He counted the berries and recorded his movements as he walked. Movements were recorded as a sequence of transitions from one cell to another through their common side along one of the marked (or reverse) directions. The route consisted of linear sections determined by directions (*X*, *Y*, or *Z*) and lengths (signed nonzero integers). A movement in the marked direction is represented with positive numbers, in the reverse direction — with negative numbers.

Your task is to write a program, which determines a route from the last cell visited by the student back to the central cell, having the least possible number of cells in it.

Input

The first line of the input contains n — the length of the route ($n > 0$). Each of the following n lines contains a letter denoting a direction (*X*, *Y*, or *Z*) and a signed integer l ($l \neq 0$) denoting the length of the section (in cells). The letter and the number are separated by one space.

While wandering, the student moved away from the central cell for no more than 100 cells in each of marked and reverse directions. The total length of the route does not exceed 32000 linear sections.

Output

The output must contain the description of a route from the last cell visited by the student back to the central cell, having the least possible number of cells in it.

The first line of the output must contain m — the length of the route (number of sections in the back route, $m \geq 0$). The following m lines of the output must contain the description of the sections of the back route in the same format as in the input.

Sample

input	output
4 Z -2 Y 3 Z 3 X -1	2 Y -2 Z -2

Problem Source: Quarterfinal, Central region of Russia, Rybinsk, October 17-18 2001

```
/*
Timus 1140. Swamp Incident
There will be an optimal solution of one these 3 kinds:
a) only using directions Y and Z, where each X becomes Y-Z
b) only using directions X and Z, where each Y becomes X+Z
c) only using directions X and Y, where each Z becomes -X+Y
Output the minimum of these 3.
*/
#include <cmath>
#include <iostream>

using namespace std;

int main() {
  int n;
  cin >> n;
  int x = 0, y = 0, z = 0;
  for (int i = 0; i < n; ++i) {
    char c;
    int k;
    cin >> c >> k;
    if (c == 'X') x += k;
    else if (c == 'Y') y += k;
    else if (c == 'Z') z += k;
  }
  int yz_y = y+x, yz_z = z-x;
  int xz_x = x+y, xz_z = z+y;
  int xy_x = x-z, xy_y = y+z;
  int m1 = abs(yz_y)+abs(yz_z);
  int m2 = abs(xz_x)+abs(xz_z);
  int m3 = abs(xy_x)+abs(xy_y);
  if (m1 <= m2 && m1 <= m3) {
    if (yz_y == 0 && yz_z == 0) cout << 0 << endl;
    else if (yz_y == 0 || yz_z == 0) cout << 1 << endl;
    else cout << 2 << endl;
    if (yz_y != 0) cout << "Y " << -yz_y << endl;
    if (yz_z != 0) cout << "Z " << -yz_z << endl;
  }
  else if (m2 <= m1 && m2 <= m3) {
    if (xz_x == 0 && xz_z == 0) cout << 0 << endl;
    else if (xz_x == 0 || xz_z == 0) cout << 1 << endl;
    else cout << 2 << endl;
    if (xz_x != 0) cout << "X " << -xz_x << endl;
    if (xz_z != 0) cout << "Z " << -xz_z << endl;
  }
  else {
    if (xy_x == 0 && xy_y == 0) cout << 0 << endl;
    else if (xy_x == 0 || xy_y == 0) cout << 1 << endl;
    else cout << 2 << endl;
    if (xy_x != 0) cout << "X " << -xy_x << endl;
    if (xy_y != 0) cout << "Y " << -xy_y << endl;
  }
}
```

Timus 1142. Relations

1142. Relations

Time Limit: 1.0 second

Memory Limit: 16 MB

Background

Consider a specific set of comparable objects. Between two objects *a* and *b*, there exits one of the following three classified relations:

$$a = b$$
$$a < b$$
$$b < a$$

Because relation '=' is symmetric, it is not repeated above.

So, with 3 objects (*a*, *b*, *c*), there can exist 13 classified relations:

$$a = b = c \quad a = b < c \quad c < a = b \quad a < b = c$$
$$b = c < a \quad a = c < b \quad b < a = c \quad a < b < c$$
$$a < c < b \quad b < a < c \quad b < c < a \quad c < a < b$$
$$c < b < a$$

Problem

Given *N*, determine the number of different classified relations between *N* objects.

Input

Includes many integers *N* (in the range from 2 to 10), each number on one line. Ends with −1.

Output

For each *N* of input, print the number of classified relations found, each number on one line.

Sample

input	output
2 3 -1	3 13

```
/*
Timus 1142. Relations
This is a combinatorics problem which can be solved by DP.
cn(n,k) = n!/(k! * (n - k)!) (the binomial coefficients)
fun(n) = #{classified relations between n objects} This is calculated
recursively in the implementation below.
Both cn() and fun() store the intermediate values in order to avoid
redundant calculations in future calls with the same parametres.
The number the statement asks for is fun(n), for a given n.
*/
#include <iostream>
#include <cstring>
using namespace std;

typedef long long ll;

ll dp_cn[20][20], dp_fun[20];

ll cn(int n, int k) {
  if (k == 0 or n == k) return 1;
  if (dp_cn[n][k] != -1) return dp_cn[n][k];
  return dp_cn[n][k] = cn(n - 1, k) + cn(n - 1, k - 1);
}

ll fun(int n) {
  if (n == 0) return 1;
  if (dp_fun[n] != -1) return dp_fun[n];
  ll res = 0;
  for (int k = 1; k <= n; ++k)
    res += cn(n, k)*fun(n - k);
  return dp_fun[n] = res;
}

int main() {
  memset(dp_cn, -1, sizeof(dp_cn));
  memset(dp_fun, -1, sizeof(dp_fun));

  int n;
  while (cin >> n and n != -1) cout << fun(n) << endl;
}
```

Timus 1143. Electric Path

1143. Electric Path

Time Limit: 1.0 second
Memory Limit: 16 MB

Background

At the team competition of the 10th national student informatics Olympic, which is organized at Hanoi National University, there are N teams participating. Each team is assigned to work in a camp. On the map, it can be seen that the camps are positioned on the vertices of a convex polygon with N vertices: P_1, P_2, …, P_N (the vertices are enumerated around the polygon in counter-clockwise order.) In order to achieve absolute safety providing electricity to the camps, besides an electric supplying system, the host organization set up a path from a reserved electricity generator (which is placed in one of the camps) to every camp once, and *the path's total length is minimum*.

Problem

Given the coordinates of the polygons' vertices (the camps' positions), determine the length of the electric path corresponding to the host organization's arrangement.

Input

The first line contains the integer N ($1 \le N \le 200$). The i'th line of the next N lines contains two real numbers x_i, y_i, separated by a space, with no more than 3 digits after the decimal points, are vertex P_i's coordinates on the plane (with $i = 1, 2, \ldots, N$). The length of the path connecting two vertex (x_i, y_i) and (x_j, y_j) is computed with the formula: $\text{sqrt}((x_i - x_j)^2 + (y_i - y_j)^2)$.

Output

The only line should contain real number L (written in real number format, with 3 digits after the decimal point), which is the total length of the electric path.

Sample

input	output
4 50.0 1.0 5.0 1.0 0.0 0.0 45.0 0.0	50.211

Problem Source: The competition for selecting the Vietnam IOI team

```
/*
Timus 1143. Electric Path
Let a state be an interval of consecutive vertices and which of the two
endpoints is the last visited vertex. For each state compute using dynamic
programming the length of the shortest path that goes through all the vertices
and ends in the state's last visited endpoint.
*/
#include <cmath>
#include <iostream>
#include <vector>

#define x first
#define y second

using namespace std;

typedef pair<double, double> PDD;

const double INF = 1e100;

double dist(PDD a, PDD b) {
  return sqrt((a.x-b.x)*(a.x-b.x)+(a.y-b.y)*(a.y-b.y));
}

int main() {
  cout.setf(ios::fixed);
  cout.precision(3);
  int n;
  cin >> n;
  vector<PDD> p(n);
  for (int i = 0; i < n; ++i) {
    cin >> p[i].x >> p[i].y;
  }
  vector<vector<vector<double> > > len(n, vector<vector<double> >(n, vector<double
      >(2, INF)));
  for (int i = 0; i < n; ++i) {
    len[i][i][0] = len[i][i][1] = 0.0;
  }
  for (int d = 1; d < n; ++d) {
    for (int i = 0; i < n; ++i) {
      len[i][(i+d)%n][0] = min(len[(i+1)%n][(i+d)%n][0]+dist(p[(i+1)%n], p[i]), len
          [(i+1)%n][(i+d)%n][1]+dist(p[(i+d)%n], p[i]));
      len[i][(i+d)%n][1] = min(len[i][(i+d-1)%n][0]+dist(p[i], p[(i+d)%n]), len[i][(
          i+d-1)%n][1]+dist(p[(i+d-1)%n], p[(i+d)%n]));
    }
  }
  double ans = INF;
  for (int i = 0; i < n; ++i) {
    ans = min(ans, len[i][(i+n-1)%n][0]);
    ans = min(ans, len[i][(i+n-1)%n][1]);
  }
  cout << ans << endl;
}
```

1145. Rope in the Labyrinth

Time Limit: 0.5 second
Memory Limit: 16 MB

A labyrinth with rectangular form and size $m \times n$ is divided into square cells with sides' length 1 by lines that are parallel with the labyrinth's sides. Each cell of the grid is either occupied or free. It is possible to move from one free cell to another free cells that share a common side with the cell. One cannot move beyond the labyrinth's borders. The labyrinth is designed pretty specially: for any two cells there is only one way to move from one cell to the other. There is a hook at each cell's center. In the labyrinth there are two special free cells, such that if you can connect the hooks of those two cells with a rope, the labyrinth's secret door will be automatically opened. The problem is to prepare a shortest rope that can guarantee, you always can connect the hooks of those two cells with the prepared rope regardless their position in the labyrinth.

Input

The first line contains integers n and m ($3 \le n, m \le 820$). The next lines describe the labyrinth. Each of the next m lines contains n characters. Each character is either "#" or ".", with "#" indicating an occupied cell, and "." indicating a free cell.

Output

Print out in the single line the length (measured in the number of cells) of the required rope.

Sample

<table><tr><th>input</th><th>output</th></tr><tr><td><pre>7 6
#######
#.#.###
#.#.###
#.#.#.#
#.....#
#######</pre></td><td><pre>8</pre></td></tr></table>

```
/*
Timus 1145. Rope in the Labyrinth
The constraint "for any two cells there is only one way to move from one cell
to the other" is saying that the free cells in the given labyrinth form a tree.
The answer is exactly the diameter of the tree. The diameter of a tree can be
computed by running two BFS's. Run a BFS from an arbitrarily chosen node. Then
select one node at maximum distance from the node chosen initially, and run a
second BFS from this second node. The distance of the furthest node from this
one is exactly the diameter of the tree.
*/
#include <iostream>
#include <queue>
#include <utility>
#include <string>
using namespace std;

#define X first
#define Y second

typedef pair<int, int> P;
typedef queue<P> Q;

const int diri[4] = { -1, 0, 1, 0 };
const int dirj[4] = { 0, 1, 0, -1 };

int F, C, T;
string mapa[1000];
int dist[1000][1000];

void bfs(int f, int c) {
  for (int i = 0; i < F; ++i)
    for (int j = 0; j < C; ++j)
      dist[i][j] = -1;
  dist[f][c] = 0;

  Q q;
  q.push(P(f, c));

  while (not q.empty()) {
    int x = q.front().X, y = q.front().Y;
    q.pop();

    for (int k = 0; k < 4; ++k) {
      int i = x + diri[k];
      int j = y + dirj[k];
      if (0 <= i and i < F and 0 <= j and j < C and
          mapa[i][j] == '.' and dist[i][j] == -1) {
        dist[i][j] = dist[x][y] + 1;
        q.push(P(i, j));
      }
    }
  }
}

int main() {
  cin >> C >> F;
  for (int i = 0; i < F; ++i) cin >> mapa[i];

  int sx = 0, sy = 0;
  for (int i = 0; i < F; ++i)
    for (int j = 0; j < C; ++j)
```

```
      if (mapa[i][j] == '.') {
        sx = i;
        sy = j;
        break;
      }

  bfs(sx, sy);

  int maxi = 0;
  for (int i = 0; i < F; ++i)
    for (int j = 0; j < C; ++j)
      if (dist[i][j] > maxi) {
        maxi = dist[i][j];
        sx = i;
        sy = j;
      }

  bfs(sx, sy);

  maxi = 0;
  for (int i = 0; i < F; ++i)
    for (int j = 0; j < C; ++j)
      maxi = max(maxi, dist[i][j]);
  cout << maxi << endl;
}
```

1146. Maximum Sum

Time Limit: 1.0 second
Memory Limit: 16 MB

Given a 2-dimensional array of positive and negative integers, find the sub-rectangle with the largest sum. The sum of a rectangle is the sum of all the elements in that rectangle. In this problem the sub-rectangle with the largest sum is referred to as the *maximal sub-rectangle*. A sub-rectangle is any contiguous sub-array of size 1×1 or greater located within the whole array.

As an example, the maximal sub-rectangle of the array:

0	−2	−7	0
9	2	−6	2
−4	1	−4	1
−1	8	0	−2

is in the lower-left-hand corner and has the sum of 15.

Input

The input consists of an $N \times N$ array of integers. The input begins with a single positive integer N on a line by itself indicating the size of the square two dimensional array. This is followed by N^2 integers separated by white-space (newlines and spaces). These N^2 integers make up the array in row-major order (i.e., all numbers on the first row, left-to-right, then all numbers on the second row, left-to-right, etc.). N may be as large as 100. The numbers in the array will be in the range [−127, 127].

Output

The output is the sum of the maximal sub-rectangle.

Sample

input	output
4 0 -2 -7 0 9 2 -6 2 -4 1 -4 1 -1 8 0 -2	15

```
/*
Timus 1146. Maximum Sum
The 1-dimension version of this problem where one has to find the maximal
sub-rectangle in a 1xN grid can be solved in O(N).
The current 2-dimensional problem can be reduced to solving O(N^2) 1-dimensional
problems. For each pair (i,j), where 1<=i<=j<=N (there are O(N^2) such pairs),
do the following:
Construct a sequence V of length N where the k-th element is calculated as:
V[k] = M[k][i] + M[k][i+1] + ... + M[k][j-1] + M[k][j]
Run the algorithm for the 1-dimensional problem, and this will give the answer
restricted to rectangles with sides on columns i and j.
The answer to the problem is the maximum of this value for all the pairs (i,j).
The complexity of the proposed algorithm is O(N^3).
*/
#include <iostream>
using namespace std;

int mat[110][110], sum[110][110];

inline int getsum(int f, int c1, int c2) {
  int res = sum[f][c2];
  if (c1 > 0) res -= sum[f][c1 - 1];
  return res;
}

int main() {
  int n;
  cin >> n;
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      cin >> mat[i][j];

  for (int i = 0; i < n; ++i) {
    sum[i][0] = mat[i][0];
    for (int j = 1; j < n; ++j)
      sum[i][j] = sum[i][j - 1] + mat[i][j];
  }

  int maxi = mat[0][0];
  for (int i = 0; i < n; ++i)
    for (int j = 0; j < n; ++j)
      maxi = max(maxi, mat[i][j]);

  for (int i = 0; i < n; ++i)
    for (int j = i; j < n; ++j) {
      int p = 0, s = 0;
      for (int k = 0; k < n; ++k) {
        s += getsum(k, i, j);
        while (s < 0) {
          s -= getsum(p, i, j);
          ++p;
        }
        if (p <= k) maxi = max(maxi, s);
      }
    }

  cout << maxi << endl;
}
```

1147. Shaping Regions

Time Limit: 0.5 second
Memory Limit: 16 MB

N opaque rectangles ($1 \le N \le 1000$) of various colors are placed on a white sheet of paper whose size is A wide by B long. The rectangles are put with their sides parallel to the sheet's borders. All rectangles fall within the borders of the sheet so that different figures of different colors will be seen.

The coordinate system has its origin (0, 0) at the sheet's lower left corner with axes parallel to the sheet's borders.

Input

The order of the input lines dictates the order of laying down the rectangles. The first input line is a rectangle "on the bottom". First line contains A, B and N, space separated ($1 \le A, B \le 10000$). Lines 2, …, $N + 1$ contain five integers each: *llx*, *lly*, *urx*, *ury*, color: the lower left coordinates and upper right coordinates of the rectangle whose color is *color* ($1 \le color \le 2500$) to be placed on the white sheet. The color 1 is the same color of white as the sheet upon which the rectangles are placed.

Output

The output should contain a list of all the colors that can be seen along with the total area of each color that can be seen (even if the regions of color are disjoint), ordered by increasing color. Do not display colors with no area.

Sample

input	output
20 20 3 2 2 18 18 2 0 8 19 19 3 8 0 10 19 4	1 91 2 84 3 187 4 38

```cpp
/*
Timus 1147. Shaping Regions
First of all, compress the coordinates so we have a matrix not bigger than 2Nx2N.
Now calculate, for each cell of the compressed matrix, which color will it be at
the end. This can be done in O(N^2*log(N)) (look at the implementation for more
details). Once that information is calculated, the answer can be easily obtained
using the original non-compressed coordinates.
*/
#include <iostream>
#include <queue>
#include <map>
#include <vector>
#include <set>
#include <cstdio>
using namespace std;

#define X first
#define Y second
#define PB push_back

typedef long long ll;
typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef map<int, int> MAP;
typedef MAP::iterator Mit;
typedef pair<int, int> P;
typedef priority_queue<int> PQ;
typedef set<int> SET;

int A, B, N, MX, MY;
int coordx[2100], coordy[2100];
int x1[2100], yy1[2100], x2[2100], y2[2100], color[2100];
int tx1[2100], tyy1[2100], tx2[2100], ty2[2100];
MAP mapx, mapy;

ll result[2600];

PQ cua;
int rem[2100], T;

int st[2100][1100], st_sz[2100];
int en[2100][1100], en_sz[2100];

int get() {
  while (rem[cua.top()] == T) cua.pop();
  return cua.top();
}

void del(int n) {
  rem[n] = T;
}

void ins(int n) {
  cua.push(n);
}

void clear() {
  cua = PQ();
  ++T;
}
```

```
int main() {
  scanf("%d%d%d", &A, &B, &N);
  ++mapx[0];
  ++mapy[0];

  ++mapx[A];
  ++mapy[B];

  ++N;
  x1[0] = yy1[0] = 0;
  x2[0] = A;
  y2[0] = B;
  color[0] = 0;

  for (int i = 1; i < N; ++i) {
    scanf("%d%d%d%d%d", &x1[i], &yy1[i], &x2[i], &y2[i], &color[i]);
    --color[i];

    ++mapx[x1[i]];
    ++mapy[yy1[i]];
    ++mapx[x2[i]];
    ++mapy[y2[i]];
  }

  MX = 0;
  for (Mit it = mapx.begin(); it != mapx.end(); ++it) {
    coordx[MX] = it->X;
    it->Y = MX;
    ++MX;
  }

  MY = 0;
  for (Mit it = mapy.begin(); it != mapy.end(); ++it) {
    coordy[MY] = it->X;
    it->Y = MY;
    ++MY;
  }

  for (int i = 0; i < N; ++i) tx1[i] = mapx[x1[i]];
  for (int i = 0; i < N; ++i) tyy1[i] = mapy[yy1[i]];
  for (int i = 0; i < N; ++i) tx2[i] = mapx[x2[i]];
  for (int i = 0; i < N; ++i) ty2[i] = mapy[y2[i]];

  for (int i = 0; i + 1 < MX; ++i) {
    for (int j = 0; j < MY; ++j) st_sz[j] = 0;
    for (int j = 0; j < MY; ++j) en_sz[j] = 0;

    for (int k = 0; k < N; ++k) {
      if (x1[k] <= coordx[i] and coordx[i + 1] <= x2[k]) {
        st[tyy1[k]][st_sz[tyy1[k]]++] = k;
        en[ty2[k]][en_sz[ty2[k]]++] = k;
      }
    }

    clear();
    for (int j = 0; j + 1 < MY; ++j) {
      for (int k = 0; k < en_sz[j]; ++k) del(en[j][k]);
      for (int k = 0; k < st_sz[j]; ++k) ins(st[j][k]);
      int t = get();
      result[color[t]] += ll(coordx[i + 1] - coordx[i])*(coordy[j + 1] - coordy[j]);
    }
```

```
  }

  for (int i = 0; i < 2500; ++i)
    if (result[i] > 0) printf("%d %lld\n", i + 1, result[i]);
}
```

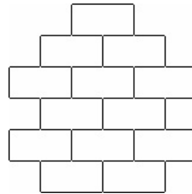
1148. Building Towers

Time Limit: 1.0 second

Memory Limit: 4 MB

There are N bricks in a toy box which have 1-unit height and 2-unit width. The teacher organizes a tower-building game. The tower is built of the bricks. The tower consists of H levels. The bottom level contains M bricks, every next level must contain exactly one brick less or greater than the level just below it.

Here is an example of a tower with H=6, M=2, N=13.



The tower with H levels can be represented by the array of H integers, which are the numbers of bricks in each level from the bottom to the top. Consider all different towers with exactly H levels and exactly M bricks in the bottom level that can be built using not more than N bricks. We can number these towers in such way that corresponding arrays will be ordered lexicographically.

Your task is to find a tower with specific number in the sense of described order.

Input

The first line of the input contains positive numbers N, H and M (N ≤ 32767, H ≤ 60, M ≤ 10). Each of the following lines contains integer K, which is the interested tower number. The last line contains number -1. The towers are numbered starting from 1.

Output

The first line of the output should contain the total number of different towers that can be built. Each following line should contain an array describing the tower with number K given in respective line of input. The numbers in the arrays should be separated by at least one space.

Sample

input	output
22 5 4 1 10 -1	10 4 3 2 1 2 4 5 4 5 4

```
/*
Timus 1148. Building Towers
This problem can be solved with simple DP. The only thing that makes this
problem complicated is the strong restriction in memory usage. We are
allowed to use at most 4MB of memory. Given that one row in the table of
the proposed DP algorithm depends only of the immediately last one, we can
store only 2 adjacent rows at a time instead of storing the entire table in
memory. In order to reconstruct the solution (thing that has to be done for
each value of K), we will have to recalculate the entire "table", because
we won't have track of the intermediate values. This makes the algorithm slower,
but it's a way to keep the memory usage within 4MB.
*/
#include <iostream>
using namespace std;

typedef long long ll;

const int HMAX = 59;
const int MMAX = 69;
const int NMAX = 2370;
const int INPMAX = 5000;

ll dp[2][MMAX + 1][NMAX + 1];
ll num[INPMAX + 1];
int ena[INPMAX];
int res[INPMAX][HMAX + 1];

void compute_dp(int H) {
  for (int m = 1; m <= MMAX; ++m) {
    dp[0][m][0] = 1;
    for (int n = 1; n <= NMAX; ++n)
      dp[0][m][n] = 0;
  }
  for (int h = 1; h <= H; ++h)
    for (int m = 1; m <= MMAX; ++m)
      for (int n = 0; n <= NMAX; ++n) {
        dp[h&1][m][n] = 0;
        if (1 <= m - 1 and 0 <= n - (m - 1)) dp[h&1][m][n] += dp[(h - 1)&1][m - 1][n
             - (m - 1)];
        if (m + 1 <= MMAX and 0 <= n - (m + 1)) dp[h&1][m][n] += dp[(h - 1)&1][m +
            1][n - (m + 1)];
      }
}

void compute_sm(int H) {
  for (int m = 1; m <= MMAX; ++m)
    for (int n = 1; n <= NMAX; ++n)
      dp[H&1][m][n] += dp[H&1][m][n - 1];
}

inline ll atmost(int h, int m, int n) {
  if (n < 0) return 0;
  if (m <= 0) return 0;
  return dp[h&1][m][n];
}

int main() {
  int N, H, M;
  scanf("%d%d%d", &N, &H, &M);
  N = min(NMAX, N);
```

```
  compute_dp(H - 1);
  compute_sm(H - 1);
  cout << dp[(H - 1)&1][M][N - M] << endl;

  int tcas = 0;
  while (scanf("%lld", &num[tcas]) != EOF and num[tcas] != -1) {
    --num[tcas];
    ++tcas;
  }

  for (int i = 0; i < tcas; ++i) res[i][0] = M;
  for (int i = 0; i < tcas; ++i) ena[i] = N - M;

  for (int i = 2; i <= H; ++i) {
    int h = H - i;
    if (h&1) compute_dp(h);
    compute_sm(h);

    for (int j = 0; j < tcas; ++j) {
      int n = ena[j];
      int m = res[j][i - 2];
      ll t = atmost(h, m - 1, n - (m - 1));
      if (num[j] < t) --m;
      else {
        num[j] -= t;
        ++m;
      }
      res[j][i - 1] = m;
      ena[j] -= m;
    }
  }

  for (int i = 0; i < tcas; ++i) {
    for (int j = 0; j < H; ++j) {
      if (j) printf(" ");
      printf("%d", res[i][j]);
    }
    printf("\n");
  }
}
```

1150. Page Numbers

Time Limit: 1.0 second
Memory Limit: 16 MB

John Smith has decided to number the pages in his notebook from 1 to N. Please, figure out the number of zeros, ones, twos, …, nines he might need.

Input

One number N ($1 \le N < 10^9$).

Output

Output 10 lines. The first line should contain the number of zeros needed, the second line should contain the number of ones needed, …, the tenth line should contain the number of nines needed.

Sample

input	output
12	1 5 2 1 1 1 1 1 1 1

Problem Author: Eugene Bryzgalov
Problem Source: Ural Collegiate Programming Contest, April 2001, Perm, English Round

```cpp
/*
Timus 1150. Page Numbers
This problem can be solved with Dynamic Programming.
In the implementation below, fun(n,z,u) returns a vector of length 10, which
contains, for each digit, the number of times it appears among all the numbers
in the set defined by the parameters n, z, u, and the global array num.
The meaning of these parameters is:
n: the number of digits the number may have
z: whether the leading zeroes must be counted
u: whether the number must be not greater than reverse(num[0..n-1])
*/
#include <iostream>
#include <vector>
using namespace std;

typedef long long ll;
typedef vector<ll> Vi;

int num[100];
Vi dp[12][2][2];

Vi fun(int n, int z, int u) {
  if (n < 0) {
    Vi v(11);
    v[10] = 1;
    return v;
  }

  if (dp[n][z][u].size()) return dp[n][z][u];

  Vi v(11, 0);
  for (int i = 0; i <= (u ? num[n] : 9); ++i) {
    Vi vv = fun(n - 1, (z == 1 and i == 0 ? 1 : 0), (u == 1 and i == num[n] ? 1 : 0)
        );
    for (int j = 0; j <= 10; ++j) v[j] += vv[j];
    if (i != 0 or z == 0) v[i] += vv[10];
  }
  return dp[n][z][u] = v;
}

Vi compute(ll n) {
  int d = 0;
  while (n >= 10) {
    num[d++] = n%10;
    n /= 10;
  }
  num[d++] = n;
  return fun(d - 1, 1, 1);
}

int main() {
  ll n;
  cin >> n;

  Vi v = compute(n);
  for (int i = 0; i < 10; ++i) cout << v[i] << endl;
}
```

1151. Radiobeacons

Time Limit: 0.5 second
Memory Limit: 16 MB

N radiobeacons are located on the plane. Their exact positions are unknown but we know that $N \le 10$ and that their coordinates are integers from 1 to 200. Each beacon produces unique signal, that distinguishes it from the other beacons.

In some different places, we should call them *checkpoints*, coordinates of which are well known, there were conducted measurements. As a result of these measurements distances from checkpoints to some of beacons became known. Here we should note, that the distance between points A and B equals $\max(|A_x - B_x|, |A_y - B_y|)$.

You need to get positions of all beacons basing on coordinates of the checkpoints and results of measurements, if that is possible.

Input

First line contains an integer M, the number of checkpoints. $1 \le M \le 20$. Then M lines follow, each of them contains an information received from one of the checkpoints, formatted as follows:

$<X_i>,<Y_i>:<ID_1>-<R_1>[,<ID_2>-<R_2>][,\ldots]$

where X_i, Y_i are coordinates of checkpoints, ID_k is ID of beacon k, R_k is a distance from checkpoint i to beacon k. Coordinates of checkpoints are integers from 1 to 200. Each checkpoint measures at least one signal. IDs of beacons are integers from 1 to 30000.

Output

Output N lines, which should look as follows:

$<ID_k>:<x_k>,<y_k>$ | UNKNOWN

Here x_k and y_k are coordinates of k'th beacon on the field. If a position of some beacon cannot be determined unambiguously, its coordinates should be replaced with a word "UNKNOWN". All lines should be ordered by ID_k in an ascending order.

Sample

input	output
2 15,15:16-7,5-3 10,10:5-2,16-2	5:12,12 16:UNKNOWN

Problem Source: Ural Collegiate Programming Contest, April 2001, Perm, English Round

```
/*
Timus 1151. Radiobeacons
Do the following for each beacon: for each integer position
inside the map check if the beacon can be at that position or not, by checking
if all the information given in the input remains consistent. If the beacon
can be at a single position, then output this position for that beacon,
otherwise output UNKNOWN.
*/
#include <iostream>
#include <utility>
#include <vector>
#include <map>
#include <sstream>
#include <algorithm>
using namespace std;

#define X first
#define Y second
#define PB push_back

typedef map<int, int> MAP;
typedef MAP::iterator Mit;
typedef pair<int, int> P;
typedef pair<P, int> PP;
typedef vector<PP> Vpp;
typedef vector<Vpp> Mpp;
typedef stringstream SS;

int N, M;
MAP mapp;
Vpp info[100];
P res[100];

int main() {
  cin >> M;
  string line;
  getline(cin, line);
  for (int i = 0; i < M; ++i) {
    getline(cin, line);
    SS ss(line);
    int x, y, id, dis;
    char spam;
    ss >> x >> spam >> y;
    while (ss >> spam >> id >> spam >> dis) {
      if (mapp.count(id) == 0) mapp[id] = N++;
      int num = mapp[id];
      info[num].PB(PP(P(x, y), dis));
    }
  }

  for (int k = 0; k < N; ++k) {
    int x = 0, y = 0;
    for (int i = 1; x != -1 and i <= 200; ++i)
      for (int j = 1; x != -1 and j <= 200; ++j) {
        bool ok = true;
        for (int t = 0; ok and t < int(info[k].size()); ++t) {
          if (max(abs(i - info[k][t].X.X), abs(j - info[k][t].X.Y)) != info[k][t].Y)
              {
            ok = false;
          }
        }
```

```
        if (ok) {
          if (x == 0) {
            x = i;
            y = j;
          }
          else {
            x = -1;
          }
        }
      }
    if (x == 0) x = -1;
    res[k] = P(x, y);
  }

  for (Mit it = mapp.begin(); it != mapp.end(); ++it) {
    cout << it->X << ":";
    if (res[it->Y].X == -1) cout << "UNKNOWN" << endl;
    else cout << res[it->Y].X << "," << res[it->Y].Y << endl;
  }
}
```

1152. False Mirrors

Time Limit: 2.0 second
Memory Limit: 16 MB

Background

We wandered in the labyrinth for twenty minutes before finally entering the large hall. The walls were covered by mirrors here as well. Under the ceiling hung small balconies where monsters stood. I had never seen this kind before. They had big bulging eyes, long hands firmly holding riffles and scaly, human-like bodies. The guards fired at me from the balconies, I shot back using my BFG-9000. The shot shattered three mirrors filling the room with silvery smoke. Bullets drummed against my body-armor knocking me down to the floor. Falling down I let go a shot, and got up as fast as I fell down by rotating on my back, like I did in my youth while break dancing, all this while shooting three more times. Three mirrors, three mirrors, three mirrors…

Sergey Lukjanenko, "The Labyrinth of Reflections"

Problem

BFG-9000 destroys three adjacent balconies per one shoot. (N-th balcony is adjacent to the first one). After the shoot the survival monsters inflict damage to Leonid (main hero of the novel) — one unit per monster. Further follows new shoot and so on until all monsters will perish. It is required to define the minimum amount of damage, which can take Leonid.

Input

The first line contains integer N, amount of balconies, on which monsters have taken a circular defense. $3 \le N \le 20$. The second line contains N integers, amount of monsters on each balcony (not less than 1 and no more than 100 on each).

Output

Output minimum amount of damage.

Sample

input	output
7 3 4 2 2 1 4 1	9

Problem Author: Eugene Bryzgalov
Problem Source: Ural Collegiate Programming Contest, April 2001, Perm, English Round

```
/*
Timus 1152. False Mirrors
This problem can be solved with Dynamic Programming using bitmasks.
The state is represented by a bitmask, each bit corresponds to one
monster, the bit represents whether the monster is alive. The number
of states grows exponentially with respect to the number of monsters.
Each state can be computed in O(N), and there are O(2^N) different
states to be computed.
*/
#include <iostream>
#include <cstring>
using namespace std;

const int INF = 1000000000;

int N, V[100], dp[1<<20];

int fun(int mask) {
  if (mask == 0) return 0;
  if (dp[mask] != -1) return dp[mask];

  int res = INF;
  int s = 0;
  for (int i = 0; i < N; ++i) if ((mask>>i)&1) s += V[i];

  for (int i = 0; i < N; ++i) {
    int j = (i + 1)%N, k = (i + 2)%N;
    if (((mask>>i)&1) or ((mask>>j)&1) or ((mask>>k)&1)) {
      int m = mask, t = s;
      if ((m>>i)&1) { m -= 1<<i; t -= V[i]; }
      if ((m>>j)&1) { m -= 1<<j; t -= V[j]; }
      if ((m>>k)&1) { m -= 1<<k; t -= V[k]; }
      t += fun(m);
      res = min(res, t);
    }
  }

  return dp[mask] = res;
}

int main() {
  memset(dp, -1, sizeof(dp));

  cin >> N;
  for (int i = 0; i < N; ++i) cin >> V[i];

  cout << fun((1<<N) - 1) << endl;
}
```

1154. Mages Contest

Time Limit: 1.0 second
Memory Limit: 16 MB

The Powers of Light and the Powers of Darkness had gathered the best elemental mages of the Middle-earth: the Lords of Fire, Earth, Air and Water.

Sometime during 24 hours there exists the Moment of Power for each one of the elements, when the mastery of corresponding mage is in its maximum point. In contrary there exists the Moment of Weakness, when the mastery of the mage is minimum. In between these moments the mage's mastery changes linearly.

There can be several mages fighting on each side. The mages cumulative mastery is defined as sum of their individual masteries. The win is given to the side, which mages' cumulative mastery is the largest. The larger the advantage of one side over another, the easier its win is, and the smaller casualties are.

The Supreme Master, who is to declare the time of the Contest secretly wishes the Powers of Light to win, and tries to make this win as easy as possible. Assume that the Contest is held momentary, and the mastery of the mages doesn't change during it. You are to help the Supreme Master in selecting the time of the Contest.

Input

The first four lines contain information about the Moments of Weakness and the Moments of Power for mages of each of the elements. Each line contains five parameters separated by spaces: element code, the time of the Moment of Power, the mastery in the Moment of Power, the time of the Moment of Weakness, the mastery in the Moment of Weakness.

Element code is one of four capital letters: “A” for Air, “E” for Earth, “F” for Fire and “W” for Water. The time is HH:MM:SS formatted and lies between 00:00:00 and 23:59:59. The Moment of Power is not equal to the Moment of Weakness. The mastery in these moments is a positive integer less than or equal to 10000.

Then two more lines follow, which determine respectively the cast of the Powers of Light and the cast of the Powers of Darkness. Each line consists of symbols “A”, “E”, “F”, “W” representing one mage of the corresponding element. The number of the mages from each side is not less than 1 and is not greater than 1000.

Output

The first line should contain HH:MM:SS formatted time of the Contest. In the second line there should be one number with two decimal points, which represents the advantage of the Powers of Light over the Powers of Darkness. The time of the Contest should lie between 00:00:00 and 23:59:59. If there exist several moments can be chosen as the time of the Contest, then choose the earliest one.

If there is no way the Powers of Light can win, then write “We can't win!”

Samples

<table>
<tr><th>input</th><th>output</th></tr>
<tr><td><pre>A 10:00:00 130 18:00:00 40
E 14:00:00 150 21:30:00 25
F 06:00:00 105 18:00:00 70
W 23:00:00 140 02:00:00 20
A
WWW</pre></td><td><pre>02:00:00
25.00</pre></td></tr>
<tr><td><pre>A 10:00:00 130 18:00:00 40
E 14:00:00 150 21:30:00 25
F 06:00:00 105 18:00:00 70
W 23:00:00 140 02:00:00 20
A
WWWF</pre></td><td><pre>We can't win!</pre></td></tr>
</table>

Problem Author: Eugene Bryzgalov
Problem Source: Ural Collegiate Programming Contest, April 2001, Perm, English Round

```
/*
Timus 1154. Mages Contest
At least one of the eight power/weakness points will be a maximum of the
advantage function. Moving towards the an earlier time when the slope is zero
takes us to other maximum points but with lower time (which we want to minimize
when tied in advantage) until we reach another power/weakness point where the
slope might change or the time 00:00:00 is reached. Therefore it is only
necessary to evaluate the function at the 8 power/weakness points and at
00:00:00 and output the time with maximum advantage.
*/
#include <algorithm>
#include <iomanip>
#include <iostream>
#include <map>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

const double EPS = 1e-9;

int hhmmss_to_s(string t) {
  stringstream ss(t);
  int h, m, s;
  char c;
  ss >> h >> c >> m >> c >> s;
  return h*60*60 + m*60 + s;
}

string s_to_hhmmss(int t) {
  int h = (t/60)/60;
  int m = (t/60)%60;
  int s = t%60;
  stringstream ss;
  ss << setfill('0') << setw(2) << h << ":" << setw(2) << m << ":" << setw(2) << s;
  string res;
  ss >> res;
  return res;
}

int dist(int a, int b) {
  if (b < a) b += 24*60*60;
  return b-a;
}

double mastery(int tp, int tw, int mp, int mw, int t) {
  if (dist(t, tp) < dist(t, tw)) return double(mw*dist(t, tp) + mp*dist(tw, t))/
      double(dist(tw, tp));
  else return double(mw*dist(tp, t) + mp*dist(t, tw))/double(dist(tp, tw));
}

double eval(vector<int>& tp, vector<int>& tw, vector<int>& mp, vector<int>& mw,
    vector<int>& light, vector<int>& dark, int t) {
  double res = 0.0;
  for (int i = 0; i < 4; ++i) {
    res += mastery(tp[i], tw[i], mp[i], mw[i], t)*double(light[i]-dark[i]);
  }
  return res;
}
```

```
int main() {
  cout.setf(ios::fixed);
  cout.precision(2);
  map<char, int> elem;
  elem['A'] = 0;
  elem['E'] = 1;
  elem['F'] = 2;
  elem['W'] = 3;
  vector<int> tp(4), tw(4), mp(4), mw(4), candidates(1, 0);
  for (int i = 0; i < 4; ++i) {
    char e;
    cin >> e;
    string sp, sw;
    cin >> sp >> mp[elem[e]] >> sw >> mw[elem[e]];
    tp[elem[e]] = hhmmss_to_s(sp);
    tw[elem[e]] = hhmmss_to_s(sw);
    candidates.push_back(tp[elem[e]]);
    candidates.push_back(tw[elem[e]]);
  }
  sort(candidates.begin(), candidates.end());
  string sl, sd;
  cin >> sl >> sd;
  vector<int> light(4, 0), dark(4, 0);
  for (int i = 0; i < int(sl.size()); ++i) {
    ++light[elem[sl[i]]];
  }
  for (int i = 0; i < int(sd.size()); ++i) {
    ++dark[elem[sd[i]]];
  }
  int best = -1;
  for (int i = 0; i < int(candidates.size()); ++i) {
    if (best == -1 || eval(tp, tw, mp, mw, light, dark, candidates[i]) > eval(tp, tw
        , mp, mw, light, dark, best)+EPS) {
      best = candidates[i];
    }
  }
  if (eval(tp, tw, mp, mw, light, dark, best) < EPS) cout << "We can't win!" << endl
      ;
  else cout << s_to_hhmmss(best) << endl << eval(tp, tw, mp, mw, light, dark, best)
      << endl;
}
```

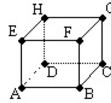
Timus 1155. Troubleduons

1155. Troubleduons

Time Limit: 0.5 second
Memory Limit: 16 MB

Archangel of the Science is reporting:
"O, Lord! Those physicists on the Earth have discovered a new elementary particle!"
"No problem, we'll add another parameter to the General Equation of the Universe."

As physics develops and moves on, scientists find more and more strange elementary particles, whose properties are more than unknown. You may have heard about muons, gluons and other strange particles. Recently scientists have found new elementary particles called troubleduons. These particles are called this way because scientists can create or annihilate them only in couples. Besides, troubleduons cause trouble to scientists, and that's why the latter want to get rid of them. You should help scientists get rid of troubleduons.



Experimental set consists of eight cameras, situated in the vertices of a cube. Cameras are named as A, B, C, …, H. It is possible to generate or annihilate two troubleduons in neighbouring cameras. You should automate the process of removing troubleduons.

Input

The only line contain eight integers ranging from 0 to 100, representing number of troubleduons in each camera of experimental set.

Output

Output sequence of actions leading to annihilating all troubleduons or "IMPOSSIBLE", if you cannot do it. Actions should be described one after another, each in a separate line, in the following way: name of the first camera, name of the second camera (it should be a neighborough to the first one), "+" if you create troubleduons, "-" if you destroy them. Number of actions in the sequence should not exceed 1000.

Samples

input	output
1 0 1 0 3 1 0 0	EF- EA- AD+ AE- DC-
0 1 0 1 2 3 2 2	IMPOSSIBLE

Problem Source: Ural Collegiate Programming Contest, April 2001, Perm, English Round

```
/*
Timus 1155. Troubleduons
Considered the cube when its vertices have been bicolored in such a way that
adjacent vertices have different color. All the possible operations preserve the
difference between the total number of troubleduons in the white vertices and
the total number of troubleduons in the black vertices. Therefore, if that
difference is not 0 it is impossible to reach a state with 0 troubleduons. In
fact, it is only impossible when the difference is other than 0 because we can
always take one white vertex X that has at least one troubleduon and one black
vertex Y that has at least one troubleduon, and remove one troubleduon from each
of them without affecting other vertices by performing the following operations:
X     V_2 -
V_2   V_3 +
V_3   V_4 -
...
V_k-1 V_k +
V_k   Y   -
where X, V_2, V_3, ..., V_k-1, V_k, Y is a path between X and Y. The number of
edges in the path is always odd because the path starts in a white vertex and
ends in a black vertex. We can make sure that the number of troubleduons will
never be negative by performing the operations that create troubleduons before
the ones that destroy them.
*/
#include <iostream>
#include <vector>

using namespace std;

char letter(int n) {
  return char('A'+n);
}

int main() {
  vector<bool> white(8, false);
  white[0] = white[2] = white[5] = white[7] = true;
  vector<int> p(8);
  int wsum = 0, particles = 0;
  for (int i = 0; i < 8; ++i) {
    cin >> p[i];
    particles += p[i];
    if (white[i]) wsum += p[i];
  }
  if (wsum != particles-wsum) {
    cout << "IMPOSSIBLE" << endl;
  }
  else {
    // cube edges
    vector<pair<int, int> > edges(12);
    edges[0] = pair<int, int>(0, 1); // AB
    edges[1] = pair<int, int>(0, 3); // AD
    edges[2] = pair<int, int>(0, 4); // AE
    edges[3] = pair<int, int>(1, 2); // BC
    edges[4] = pair<int, int>(1, 5); // BF
    edges[5] = pair<int, int>(2, 3); // CD
    edges[6] = pair<int, int>(2, 6); // CG
    edges[7] = pair<int, int>(3, 7); // DH
    edges[8] = pair<int, int>(4, 5); // EF
    edges[9] = pair<int, int>(4, 7); // EH
    edges[10] = pair<int, int>(5, 6); // FG
    edges[11] = pair<int, int>(6, 7); // GH
```

```
    // opposite vertex
    vector<int> opp(8);
    opp[0] = 6;
    opp[1] = 7;
    opp[2] = 4;
    opp[3] = 5;
    opp[4] = 2;
    opp[5] = 3;
    opp[6] = 0;
    opp[7] = 1;

    // path to the opposite vertex
    vector<vector<int> > path(8, vector<int>(3));
    path[0][0] = 0;  path[0][1] = 3;  path[0][2] = 6; // AB, BC, CG
    path[1][0] = 3;  path[1][1] = 5;  path[1][2] = 7; // BC, CD, DH
    path[2][0] = 3;  path[2][1] = 4;  path[2][2] = 8; // CB, BF, FE
    path[3][0] = 1;  path[3][1] = 0;  path[3][2] = 4; // DA, AB, BF
    path[4] = path[2];
    path[5] = path[3];
    path[6] = path[0];
    path[7] = path[1];
    for (int i = 0; i < 12; ++i) {
      for (; p[edges[i].first] > 0 && p[edges[i].second] > 0;) {
        --p[edges[i].first];
        --p[edges[i].second];
        particles -= 2;
        cout << letter(edges[i].first) << letter(edges[i].second) << "-" << endl;
      }
    }
    for (int i = 0; i < 8; ++i) {
      for (; p[i] > 0 && p[opp[i]] > 0;) {
        --p[i];
        --p[opp[i]];
        cout << letter(edges[path[i][1]].first) << letter(edges[path[i][1]].second)
            << "+" << endl;
        cout << letter(edges[path[i][0]].first) << letter(edges[path[i][0]].second)
            << "-" << endl;
        cout << letter(edges[path[i][2]].first) << letter(edges[path[i][2]].second)
            << "-" << endl;
      }
    }
  }
}
```

1156. Two Rounds

Time Limit: 2.0 second
Memory Limit: 16 MB

There are two rounds in the Urals Championship. The competitors have to solve N problems on each round. The jury had been working hard and finally managed to prepare $2N$ problems for the championship. But it appeared that among those problems there were some, which have the analogous solutions. One shouldn't assign such a problems for the same round. Please, help the jury form sets of tasks for each of the rounds.

Input

First line contains two numbers: N, the number of tasks for a round, and M, the number of pairs of tasks which should not be assigned for one round ($1 \le N \le 50$; $0 \le M \le 100$). Then M lines follow, each of them contains two numbers of analogous tasks.

Output

Output two lines, containing numbers of tasks assigned for each round. If there is no solution, output the only word "IMPOSSIBLE". If there are more than one solution you may assume anyone of them.

Sample

input	output
2 3 1 3 2 1 4 3	1 4 2 3

Problem Author: Eugene Bryzgalov
Problem Source: Ural Collegiate Programming Contest, April 2001, Perm, English Round

```
/*
Timus 1156. Two Rounds
Consider a graph where there is a node for each problem, and an edge between
every pair of similar problems. Try to bicolorate each connected component.
If there is at least one connected component which is not bipartite,
then the answer is IMPOSSIBLE, otherwise count the number of black nodes
and white nodes in each connected component.
Now, we must check if it is possible to make the total number of white nodes
equal to the total number of black nodes by inverting the colors of some
of the components. This can be done with Dynamic Programming.
If that is not possible, then the answer is IMPOSSIBLE,
otherwise the answer exists and it should be easy to construct it.
*/
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
using namespace std;

#define X first
#define Y second
#define PB push_back

typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef pair<int, int> P;
typedef vector<P> Vp;

int N, M, C;
Mi net;
Vi color;
Vi component;
Vp comp;

int dp[110][110][110];

bool bicolor(int n, int c, int com) {
  if (color[n] != -1) return c == color[n];
  color[n] = c;
  component[n] = com;
  for (int i = 0; i < int(net[n].size()); ++i)
    if (not bicolor(net[n][i], 1 - c, com)) return false;
  return true;
}

int main() {
  cin >> N >> M;
  net = Mi(2*N);
  for (int i = 0; i < M; ++i) {
    int a, b;
    cin >> a >> b;
    --a; --b;
    net[a].PB(b);
    net[b].PB(a);
  }

  color = Vi(2*N, -1);
  component = Vi(2*N);

  for (int i = 0; i < 2*N; ++i) {
    if (color[i] != -1) continue;
```

```
    if (not bicolor(i, 0, C++)) {
      cout << "IMPOSSIBLE" << endl;
      return 0;
    }
  }

  comp = Vp(C, P(0, 0));
  for (int i = 0; i < 2*N; ++i) {
    if (color[i] == 0) ++comp[component[i]].X;
    else ++comp[component[i]].Y;
  }

  dp[C][0][0] = 1;
  for (int n = C - 1; n >= 0; --n)
    for (int i = 0; i <= N; ++i)
      for (int j = 0; j <= N; ++j) {
        if (comp[n].X <= i and comp[n].Y <= j and dp[n + 1][i - comp[n].X][j - comp[
            n].Y]) {
          dp[n][i][j] = 1;
        }
        else if (comp[n].Y <= i and comp[n].X <= j and dp[n + 1][i - comp[n].Y][j -
            comp[n].X]) {
          dp[n][i][j] = 2;
        }
      }
  if (dp[0][N][N] == 0) {
    cout << "IMPOSSIBLE" << endl;
    return 0;
  }

  Vi quin(C);
  int x = N, y = N;
  for (int c = 0; c < C; ++c) {
    if (dp[c][x][y] == 1) {
      quin[c] = 0;
      x -= comp[c].X;
      y -= comp[c].Y;
    }
    else {
      quin[c] = 1;
      x -= comp[c].Y;
      y -= comp[c].X;
    }
  }

  Vi va, vb;
  for (int i = 0; i < 2*N; ++i) {
    if ((color[i] + quin[component[i]])%2 == 0) va.PB(i);
    else vb.PB(i);
  }

  for (int i = 0; i < N; ++i) {
    if (i) cout << " ";
    cout << va[i] + 1;
  }
  cout << endl;

  for (int i = 0; i < N; ++i) {
    if (i) cout << " ";
    cout << vb[i] + 1;
  }
```

```
  cout << endl;
}
```

1157. Young Tiler

Time Limit: 1.0 second
Memory Limit: 16 MB

One young boy had many-many identical square tiles. He loved putting all his tiles to form a rectangle more, than anything in the world — he has learned the number of all rectangles he can form using his tiles. On his birthday he was presented a number of new tiles. Naturally, he started forming rectangles from these tiles — the thing he loved most of all! Soon he has learned all rectangles he could form with a new number of tiles.

Here we should notice that boy can easily count the number of rectangles, but he has difficulty counting the number of tiles — there are too much of them for such a young boy. But it will not be difficult for you to determine how many tiles he has now, knowing how many rectangles he could form before, how many rectangles he can form now, and how many tiles he got as a birthday present.

You are given numbers M, N and K. You should find the smallest number L, such as you can form N different rectangles using all L tiles, and form M rectangles using $L - K$ tiles.

Input

One line containing three integers: M, N, K ($1 \le M, N, K \le 10000$).

Output

If L is less than or equal to 10000, then print that number (if there is a number of such L, you should print the smallest one). If there is no solution or smallest L is greater than 10000, print 0.

Sample

input	output
`2 3 1`	`16`

Problem Source: Ural Collegiate Programming Contest, April 2001, Perm, English Round

```
/*
Timus 1157. Young Tiler
The problem asks what is the smallest number L > K such that
ceil(#div(L)/2) = n
and
ceil(#div(L-K)/2) = m
where ceil(x) is the smallest integer greater or equal than x, and #div(n) is
the number of divisors of n.

First of all we use a modified Sieve of Eratosthenes to compute a prime factor
of each number smaller than a sufficiently large amount (20000 in the following
implementation).

The number of divisors can be computed using Dynamic Programming: for a natural
number k, assuming we have already computed the number of divisors of every
i < k, take any prime factor p of k and calculate its multiplicity m (p^m
divides k but p^(m+1) does not). The number of divisors of k can be obtained as:
#div(k) = (m+1)*#div(k/p^m)
because for every divisor d of k/p^m there are (m+1) divisors of k: d, d*p,
d*p^2, ..., d*p^m.

Once the number of divisors have been computed, we can just loop over natural
numbers from 2 to 10000 increasingly, looking for the smallest that satisfies
the conditions.
*/
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<int> factor(20000, 0);
  factor[1] = 1;
  for (int i = 2; i*i < int(factor.size()); ++i) if (factor[i] == 0) {
    for (int j = 2*i; j < int(factor.size()); j += i) {
      factor[j] = i;
    }
  }
  int m, n, k;
  cin >> m >> n >> k;
  vector<int> ndiv(10001, 0);
  ndiv[1] = 1;
  int ans = 0;
  for (int i = 2; i <= 10000 && ans == 0; ++i) {
    if (factor[i] == 0) ndiv[i] = 2;
    else {
      int s = i, e = 0;
      for (; s%factor[i] == 0; s /= factor[i]) {
        ++e;
      }
      ndiv[i] = ndiv[s]*(e+1);
    }
    if (i-k > 0 && (ndiv[i]+1)/2 == n && (ndiv[i-k]+1)/2 == m) {
      ans = i;
    }
  }
  cout << ans << endl;
}
```

Timus 1158. Censored!

1158. Censored!

Time Limit: 2.0 second
Memory Limit: 16 MB

The alphabet of Freeland consists of exactly N letters. Each sentence of Freeland language (also known as Freish) consists of exactly M letters without word breaks. So, there exist exactly N^M different Freish sentences.

But after recent election of Mr. Grass Jr. as Freeland president some words offending him were declared unprintable and all sentences containing at least one of them were forbidden. The sentence S contains a word W if W is a substring of S i.e. exists such k >= 1 that S[k] = W[1], S[k+1] = W[2], ... , S[k+len(W)-1] = W[len(W)], where k+len(W)-1 <= M and len(W) denotes length of W. Everyone who uses a forbidden sentence is to be put to jail for 10 years.

Find out how many different sentences can be used now by freelanders without risk to be put to jail for using it.

Input

The first line contains three integer numbers: N - the number of letters in Freish alphabet, M - the length of all Freish sentences and P - the number of forbidden words ($1 \le N \le 50$, $1 \le M \le 50$, $0 \le P \le 10$).

The second line contains exactly N different characters - the letters of the Freish alphabet (all with ASCII code greater than 32).

The following P lines contain forbidden words, each not longer than min(M, 10) characters, all containing only letters of Freish alphabet.

Output

Output the only integer number - the number of different sentences freelanders can safely use.

Sample

input	output
3 3 3 QWE QQ WEE Q	7

Problem Author: Nick Durov
Problem Source: ACM ICPC 2001. Northeastern European Region, Northern Subregion

```
/*
Timus 1158. Censored!
First, create an automata to recognize censored sentences.
Once the automata is created, the problem can be solved with DP.
fun(n, v) returns the number of different non-censored sentences of length n
assuming we start in the state v of the automata. The DP counts the number
of different sentences which don't reach any accepting state of the automata.
*/
#include <iostream>
#include <map>
#include <vector>
#include <string>
#include <iomanip>
#include <algorithm>
#include <utility>
#include <cmath>
using namespace std;

#define PB push_back

typedef vector<int> VE;
typedef long long ll;
const ll base = 1000000000;

class BI { public:
  VE v;
  int n;
  int sig;

  void zero() {
    v = VE(1, 0);
    n = sig = 1;
  }

  void remida(int i, int que=0) {
    v.resize(n = i, que);
  }

  void treuzeros() {
    while (--n and !v[n]);
    remida(n + 1);
    if (n == 1 and !v[0]) sig = 1;
  }

  inline int compabs(const BI& b) const {
    if (n != b.n) return n - b.n;
    for (int i = n - 1; i >= 0; --i)
      if (v[i] != b.v[i]) return v[i] - b.v[i];
    return 0;
  }

  inline int dig(int i) {
    return (i < n ? v[i] : 0);
  }

  void suma(BI &b) {
    if (n < b.n) remida(b.n, 0);
    int ca = 0;
    for (int i = 0; i < n; ++i) {
      v[i] += b.dig(i) + ca;
      ca = v[i]/base;
```

```
      v[i] %= base;
    }
    if (ca) remida(n + 1, ca);
  }

  void resta(BI &b) {
    int ca = 0;
    for (int i = 0; i < n; ++i) {
      v[i] += base - b.dig(i) + ca;
      ca = v[i]/base - 1;
      v[i] %= base;
    }
    treuzeros();
  }

  BI() {
    zero();
  }

  BI(const BI &b) {
    *this = b;
  }

  BI(ll x) {
    sig = (x < 0 ? -1 : 1);
    x *= sig;
    if (x < base) {
      v = VE(1, int(x));
      n = 1;
    }
    else {
      v = VE(2);
      v[0] = x%base;
      v[1] = x/base;
      n = 2;
    }
  }

  void operator+=(BI &b) {
    if (sig == b.sig) return suma(b);
    if (compabs(b) >= 0) return resta(b);
    BI aux(b); aux.resta(*this); *this = aux;
  }

  friend ostream &operator<<(ostream &out, BI &b) {
    if (b.sig < 0) out << '-';
    int i = b.v.size() - 1;
    out << b.v[i];
    for (--i; i >= 0; --i) out << setw(9) << setfill('0') << b.v[i];
    return out;
  }
};

typedef vector<bool> Vb;
typedef vector<int> Vi;
typedef vector<Vi> Mi;
typedef map<Vi, int> MAP;
typedef vector<BI> Vbi;
typedef vector<Vbi> Mbi;

int N, M, P, nodes;
```

```
string word[10];
char alph[50];
MAP mapp;
Mi mat;
Mi rev;
Vb proh;
Mbi dp;
Mi tdp;

void next(Vi v, char c, Vi& res) {
  res = Vi(P);
  for (int i = 0; i < P; ++i) {
    string s = word[i].substr(0, v[i]) + c;
    res[i] = 0;
    for (int j = min(s.size(), word[i].size()); res[i] == 0 and j > 0; --j) {
      if (word[i].substr(0, j) == s.substr(s.size() - j, j))
        res[i] = j;
    }
  }
}

bool prohibit(Vi v) {
  for (int i = 0; i < P; ++i)
    if (v[i] == int(word[i].size()))
      return true;
  return false;
}

int node(Vi v) {
  if (mapp.count(v) != 0) return mapp[v];
  rev.PB(v);
  proh.PB(prohibit(v));
  mat.PB(Vi(N, -1));
  mapp[v] = nodes;
  dp.PB(Vbi(M + 1));
  tdp.PB(Vi(M + 1, 0));
  return nodes++;
}

BI fun(int n, int nd) {
  if (proh[nd]) {
    BI zero = 0;
    return zero;
  }
  if (n == 0) {
    BI one = 1;
    return one;
  }
  if (tdp[nd][n]) return dp[nd][n];
  BI res = 0;
  for (int i = 0; i < N; ++i) {
    int nx = mat[nd][i];
    if (nx == -1) {
      Vi t1 = rev[nd];
      Vi t2;
      next(t1, alph[i], t2);
      int t3 = node(t2);
      nx = mat[nd][i] = t3;
    }
    BI tmp = fun(n - 1, nx);
    res += tmp;
```

```
  }
  tdp[nd][n] = 1;
  return dp[nd][n] = res;
}

int main() {
  cin >> N >> M >> P;
  for (int i = 0; i < N; ++i) cin >> alph[i];
  for (int i = 0; i < P; ++i) cin >> word[i];

  BI tmp = fun(M, node(Vi(P, 0)));
  cout << tmp << endl;
}
```

1159. Fence

Time Limit: 1.0 second
Memory Limit: 16 MB

Workers are going to enclose a new working region with a fence. For their convenience the enclosed area has to be as large as possible. They have N rectangular blocks to build the fence. The length of the i-th block is L_i meters. All blocks have the same height of 1 meter. The workers are not allowed to break blocks into parts. All blocks must be used to build the fence.

Input

The first line contains one integer N ($3 \le N \le 100$). The following N lines describe fence blocks. Each block is represented by its length in meters (integer number, $1 \le L_i \le 100$).

Output

Write one non-negative number S - maximal possible area of the working region (in square meters). S must be written with two digits after the decimal point. If it is not possible to construct the fence from the specified blocks, write 0.00.

Sample

input	output
4 10 5 5 4	28.00

Problem Author: Nick Durov
Problem Source: ACM ICPC 2001. Northeastern European Region, Northern Subregion

```
/*
Timus 1159. Fence
The optimal arrangement is a polygon inscribed in a circle, and the
order of the blocks is not relevant.
In order to know the area of the polygon, first we calculate the radius
of the circumscribed circle. Fixed an arbitrary radius, it's easy to check
if the required radius should be smaller or bigger, here is the idea:
each side of the polygon determines an angle (vertex1 - center of the circle -
vertex 2), if the sum of all such angles is below 2pi, then the circle is
too big. Therefore, we can get a good approximation of the right radius
doing a binary search over the radius.
The polygon is impossible to construct when one side's length is bigger than
the sum of the rest of the sides' length.
*/
#include <algorithm>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> Vi;

const double PI = 2*acos(double(0));

int main() {
  cout.setf(ios::fixed);
  cout.precision(2);

  int n;
  cin >> n;

  Vi v(n);
  for (int i = 0; i < n; ++i) cin >> v[i];
  sort(v.begin(), v.end());

  int s = 0, m = 0;
  for (int i = 0; i < n; ++i) s += v[i];
  for (int i = 0; i < n; ++i) m = max(m, v[i]);

  if (s <= 2*m) {
    cout << double(0) << endl;
    return 0;
  }

  double esq = 0.5*m, dre = 1e+18, res = 0;
  for (int step = 0; step <= 200; ++step) {
    double rad = 0.5*esq + 0.5*dre;
    double ang = 0;
    for (int i = 0; i < n - 1; ++i) ang += 2*asin(0.5*v[i]/rad);

    if (2*PI <= ang) esq = rad;
    else {
      double dis = 2*rad*sin(0.5*ang);
      if (dis < v[n - 1]) esq = rad;
      else dre = rad;
    }

    double area = 0;
    for (int i = 0; i < n; ++i) {
      double a = 0.5*v[i]*sqrt(rad*rad - 0.25*v[i]*v[i]);
```

```
      if (i < n - 1 or ang > PI) area += a;
      else area -= a;
    }
    res = area;
  }

  cout << res << endl;
}
```

1160. Network

Time Limit: 1.0 second
Memory Limit: 16 MB

Andrew is working as system administrator and is planning to establish a new network in his company. There will be N hubs in the company, they can be connected to each other using cables. Since each worker of the company must have access to the whole network, each hub must be accessible by cables from any other hub (with possibly some intermediate hubs).

Since cables of different types are available and shorter ones are cheaper, it is necessary to make such a plan of hub connection, that the maximum length of a single cable is minimal. There is another problem - not each hub can be connected to any other one because of compatibility problems and building geometry limitations. Of course, Andrew will provide you all necessary information about possible hub connections.

You are to help Andrew to find the way to connect hubs so that all above conditions are satisfied.

Input

The first line contains two integer: N - the number of hubs in the network ($2 \le N \le 1000$) and M — the number of possible hub connections ($1 \le M \le 15000$). All hubs are numbered from 1 to N. The following M lines contain information about possible connections - the numbers of two hubs, which can be connected and the cable length required to connect them. Length is a positive integer number that does not exceed 10^6. There will be no more than one way to connect two hubs. A hub cannot be connected to itself. There will always be at least one way to connect all hubs.

Output

Output first the maximum length of a single cable in your hub connection plan (the value you should minimize). Then output your plan: first output P - the number of cables used, then output P pairs of integer numbers - numbers of hubs connected by the corresponding cable. Separate numbers by spaces and/or line breaks.

Sample

input	output
4 6 1 2 1 1 3 1 1 4 2 2 3 1 3 4 1 2 4 1	1 4 1 2 1 3 2 3 3 4

Problem Author: Andrew Stankevich
Problem Source: ACM ICPC 2001. Northeastern European Region, Northern Subregion

```cpp
/*
Timus 1160. Network
In this problem we need to find a minimum spanning tree of a
given graph. This can be solved, for example, with the well-known
Kruskal's algorithm.
*/
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
using namespace std;

#define X first
#define Y second
#define PB push_back

typedef vector<int> Vi;
typedef pair<int, int> P;
typedef pair<int , P> PP;
typedef vector<P> Vp;
typedef vector<PP> Vpp;

Vi tree;
int root(int n) {
  if (tree[n] == -1) return n;
  return tree[n] = root(tree[n]);
}

int main() {
  int n, m;
  cin >> n >> m;

  Vpp v(m);
  for (int i = 0; i < m; ++i) {
    cin >> v[i].Y.X >> v[i].Y.Y >> v[i].X;
    --v[i].Y.X;
    --v[i].Y.Y;
  }
  sort(v.begin(), v.end());

  int mini = 0;
  Vp res;
  int a = n - 1;
  tree = Vi(n, -1);
  for (int i = 0; i < m and a > 0; ++i) {
    int a = v[i].Y.X, b = v[i].Y.Y;
    int ra = root(a), rb = root(b);
    if (ra != rb) {
      --a;
      res.PB(v[i].Y);
      mini = max(mini, v[i].X);
      tree[rb] = ra;
    }
  }

  cout << mini << endl;
  cout << res.size() << endl;
  for (int i = 0; i < int(res.size()); ++i)
    cout << res[i].X + 1 << " " << res[i].Y + 1 << endl;
}
```

1161. Stripies

Time Limit: 1.0 second
Memory Limit: 16 MB

Our chemical biologists have invented a new very useful form of life called *stripies* (in fact, they were first called in Russian - *polosatiki*, but the scientists had to invent an English name to apply for an international patent). The stripies are transparent amorphous amebiform creatures that live in flat colonies in a jelly-like nutrient medium. Most of the time the stripies are moving. When two of them collide a new stripie appears instead of them. Long observations made by our scientists enabled them to establish that the weight of the new stripie isn't equal to the sum of weights of two disappeared stripies that collided; nevertheless, they soon learned that when two stripies of weights m_1 and m_2 collide the weight of resulting stripie equals to $2 \cdot \mathrm{sqrt}(m_1 m_2)$. Our chemical biologists are very anxious to know to what limits can decrease the total weight of a given colony of stripies.

You are to write a program that will help them to answer this question. You may assume that 3 or more stipies never collide together.

Input

The first line contains one integer N ($1 \le N \le 100$) - the number of stripies in a colony. Each of next N lines contains one integer ranging from 1 to 10000 - the weight of the corresponding stripie.

Output

The output must contain one line with the minimal possible total weight of colony with the accuracy of two decimal digits after the point.

Sample

input	output
3 72 30 50	120.00

Problem Author: Nick Durov
Problem Source: ACM ICPC 2001. Northeastern European Region, Northern Subregion

```
/*
Timus 1161. Stripies
The optimal way to proceed here is to choose at every step the
two biggest stripies and join them. The implementation is straightforward.
*/
#include <cmath>
#include <iostream>
#include <queue>
using namespace std;

typedef priority_queue<double> PQ;

int main() {
  cout.setf(ios::fixed);
  cout.precision(2);

  int n;
  cin >> n;

  PQ q;
  for (int i = 0; i < n; ++i) {
    double t;
    cin >> t;
    q.push(t);
  }

  while (q.size() > 1) {
    double a = q.top(); q.pop();
    double b = q.top(); q.pop();
    double c = 2*sqrt(a*b);
    q.push(c);
  }

  cout << q.top() << endl;
}
```

1162. Currency Exchange

Time Limit: 1.0 second
Memory Limit: 16 MB

Several currency exchange points are working in our city. Let us suppose that each point specializes in two particular currencies and performs exchange operations only with these currencies. There can be several points specializing in the same pair of currencies. Each point has its own exchange rates, exchange rate of A to B is the quantity of B you get for 1A. Also each exchange point has some commission, the sum you have to pay for your exchange operation. Commission is always collected in source currency.

For example, if you want to exchange 100 US Dollars into Russian Rubles at the exchange point, where the exchange rate is 29.75, and the commission is 0.39 you will get (100 - 0.39) * 29.75 = 2963.3975RUR.

You surely know that there are N different currencies you can deal with in our city. Let us assign unique integer number from 1 to N to each currency. Then each exchange point can be described with 6 numbers: integer A and B - numbers of currencies it exchanges, and real RAB, CAB, RBA and CBA - exchange rates and commissions when exchanging A to B and B to A respectively.

Nick has some money in currency S and wonders if he can somehow, after some exchange operations, increase his capital. Of course, he wants to have his money in currency S in the end. Help him to answer this difficult question. Nick must always have non-negative sum of money while making his operations.

Input

The first line contains four numbers: N - the number of currencies, M - the number of exchange points, S - the number of currency Nick has and V - the quantity of currency units he has. The following M lines contain 6 numbers each - the description of the corresponding exchange point - in specified above order. Numbers are separated by one or more spaces. $1 \le S \le N \le 100$, $1 \le M \le 100$, V is real number, $0 \le V \le 10^3$.

For each point exchange rates and commissions are real, given with at most two digits after the decimal point, $10^{-2} \le \text{rate} \le 10^2$, $0 \le \text{commission} \le 10^2$.

Let us call some sequence of the exchange operations simple if no exchange point is used more than once in this sequence. You may assume that ratio of the numeric values of the sums at the end and at the beginning of any simple sequence of the exchange operations will be less than 10^4.

Output

If Nick can increase his wealth, output YES, in other case output NO.

Sample

input	output
3 2 1 20.0 1 2 1.00 1.00 1.00 1.00 2 3 1.10 1.00 1.10 1.00	YES

Problem Author: Nick Durov
Problem Source: ACM ICPC 2001. Northeastern European Region, Northern Subregion

```
/*
Timus 1162. Currency Exchange
The currencies and the possible changes can be interpreted as nodes and
edges of a graph. We want to calculate the maximum amount we can get
of each currency. If we can get as much money as we want in any currency,
then the answer is YES, otherwise is NO. This happens when there is a
cycle in the graph that increases the amount of money of some currency, i.e.
if we start with money in a currency of that cycle, and make the changes
of the cycle we finish having more money than initially. We can check this with
the Bellman-Ford algorithm.
*/
#include <iostream>
using namespace std;

const double EPS = 1e-8;

int N, M;
int A[200], B[200];
double R[200], C[200];
double money[100];

int main() {
  int st;
  cin >> N >> M >> st;
  M *= 2;
  --st;

  for (int i = 0; i < N; ++i) money[i] = 0;
  cin >> money[st];

  for (int i = 0; i < M; i += 2) {
    cin >> A[i] >> B[i] >> R[i] >> C[i];
    --A[i]; --B[i];
    A[i + 1] = B[i];
    B[i + 1] = A[i];
    cin >> R[i + 1] >> C[i + 1];
  }

  bool yes = false;
  for (int k = 0; k <= N; ++k) {
    for (int i = 0; i < M; ++i) {
      int a = A[i], b = B[i];
      double c = C[i], r = R[i];
      if (c < money[a]) {
        double t = (money[a] - c)*r;
        if (money[b] + EPS < t) {
          money[b] = t;
          if (k == N) yes = true;
        }
      }
    }
  }

  if (yes) cout << "YES" << endl;
  else cout << "NO" << endl;
}
```

1164. Fillword

Time Limit: 1.0 second
Memory Limit: 16 MB

Alex likes solving fillwords. Fillword is a word game with very simple rules. The author of the fillword takes rectangular grid (M cells width, N cells height) and P words. Then he writes letters in the cells of the grid (one letter in one cell) so that each word can be found on the grid and the following conditions are met:

- no cell belongs to more than one word
- no cell belongs to any word more than once

Some word W (let us consider its length being k) is found on the grid if you can find such sequence of cells (x_1, y_1), (x_2, y_2), …, (x_k, y_k) that:

- (x_i, y_i) and (x_{i+1}, y_{i+1}) are neighbors ($|x_i-x_{i+1}| + |y_i-y_{i+1}| = 1$) for each i = 1, 2, …, k-1
- W[i] is written in the cell with coordinates (x_i, y_i).

The task is to find all the words on the grid. After they are found, you see that the letters in some cells are not used (they do not belong to any found word). You make up a secret word using these letters and win a big prize.

Your task is to help Alex to solve fillwords. You should find out which letters will be left after he finds all the words on the grid. The most difficult task - to make up a secret word out of them - we still reserve to Alex.

Input

The first line contains three integers - N, M ($2 \le M, N \le 10$) and P ($P \le 100$). Next N lines contain M characters each, and represent the grid. The following P lines contain words that are to be found on the fillword grid.

Fillword will always have at least one solution. All characters occurring in fillword will be capital English letters.

Output

Output letters from, which a secret word should be made up. Letters should be output in lexicographical order.

Sample

input	output
3 3 2 EBG GEE EGE BEG GEE	EEG

Problem Author: Alex Selivanov
Problem Source: ACM ICPC 2001. Northeastern European Region, Northern Subregion

```
/*
Timus 1164. Fillword
The key observation here is that one doen't need to deal with
the arrangement of the letters inside the matrix or within the words,
just with the frequency of appearance of each letter.
*/
#include <iostream>
#include <string>
#include <vector>
using namespace std;

typedef vector<int> Vi;

int main() {
  int f, c, k;
  cin >> f >> c >> k;

  Vi freq(26, 0);
  for (int i = 0; i < f; ++i) {
    string s;
    cin >> s;
    for (int j = 0; j < c; ++j) ++freq[s[j] - 'A'];
  }
  for (int i = 0; i < k; ++i) {
    string s;
    cin >> s;
    int n = s.size();
    for (int j = 0; j < n; ++j) --freq[s[j] - 'A'];
  }

  for (int i = 0; i < 26; ++i) cout << string(freq[i], 'A' + i);
  cout << endl;
}
```

1165. Subnumber

Time Limit: 1.0 second
Memory Limit: 16 MB

George likes arithmetics very much. Especially he likes the integers series. His most favourite thing is the infinite sequence of digits, which results as the concatenation of all positive integers in ascending order. The beginning of this sequence is 1234567891011121314… Let us call this sequence S. Then S[1] = 1, S[2] = 2, …, S[10] = 1, S[11] = 0, …, and so on.

George takes a sequence of digits A and wants to know when it first appears in S. Help him to solve this difficult problem.

Input

The first line contains A - the given sequence of digits. The number of digits in A does not exceed 200.

Output

Output the only number - the least k such that A[1] = S[k], A[2] = S[k+1], ... A[len(A)] = S[k + len(A) – 1], where len(A) denotes the length of A (i.e. the number of digits in it).

Sample

input	output
101	10

Problem Author: Nikita Shamgunov
Problem Source: ACM ICPC 2001. Northeastern European Region, Northern Subregion

```
/*
Timus 1165. Subnumber
This is a quite tricky problem.
The main idea is: guess what characters of the given string form the
first number that appears entirely in the string, then check if
the rest of the string is consistent with that guess by generating
the next few numbers needed to fill the string completely. Among all
those guesses choose the "smallest" one. There are some tricky cases
you must have into account.
*/
#include <iostream>
#include <vector>
#include <iomanip>
#include <sstream>
using namespace std;

typedef vector<int> VE;
typedef long long ll;
const ll base = 1000000000;

class BI { public:
  VE v;
  int n;
  int sig;

  void zero() {
    v = VE(1, 0);
    n = sig = 1;
  }

  void remida(int i, int que=0) {
    v.resize(n = i, que);
  }

  void treuzeros() {
    while (--n and !v[n]);
    remida(n + 1);
    if (n == 1 and !v[0]) sig = 1;
  }

  void parse(string num) {
    if (num[0] == '-') {
      sig = -1;
      num = num.substr(1);
    }
    else sig = 1;

    int m = num.size() - 1;
    v = VE(1 + m/9, 0); n = v.size();
    for (int i = m, exp = 0, pw = 1, pos = 0; i >= 0; --i, ++exp, pw *= 10) {
      if (exp == 9) { exp = 0; pw = 1; ++pos; }
      v[pos] += (num[i] - '0')*pw;
    }
    treuzeros();
  }

  inline int compabs(const BI& b) const {
    if (n != b.n) return n - b.n;
    for (int i = n - 1; i >= 0; --i)
      if (v[i] != b.v[i]) return v[i] - b.v[i];
    return 0;
```

```
}

inline int compara(const BI &b) const {
  if (sig != b.sig) return sig - b.sig;
  return sig*compabs(b);
}

inline bool operator<(const BI& b) const {
  return compara(b) < 0;
}

inline int dig(int i) {
  return (i < n ? v[i] : 0);
}

void suma(BI &b) {
  if (n < b.n) remida(b.n, 0);
  int ca = 0;
  for (int i = 0; i < n; ++i) {
    v[i] += b.dig(i) + ca;
    ca = v[i]/base;
    v[i] %= base;
  }
  if (ca) remida(n + 1, ca);
}

void resta(BI &b) {
  int ca = 0;
  for (int i = 0; i < n; ++i) {
    v[i] += base - b.dig(i) + ca;
    ca = v[i]/base - 1;
    v[i] %= base;
  }
  treuzeros();
}

BI() {
  zero();
}

BI(const BI &b) {
  *this = b;
}

BI(ll x) {
  sig = (x < 0 ? -1 : 1);
  x *= sig;
  if (x < base) {
    v = VE(1, int(x));
    n = 1;
  }
  else {
    v = VE(2);
    v[0] = x%base;
    v[1] = x/base;
    n = 2;
  }
}

BI(string num) {
  parse(num);
```

```
  }

  void operator=(const BI &b) {
    v = b.v;
    n = b.n;
    sig = b.sig;
  }

  void operator+=(BI &b) {
    if (sig == b.sig) return suma(b);
    if (compabs(b) >= 0) return resta(b);
    BI aux(b); aux.resta(*this); *this = aux;
  }

  void operator-=(BI &b) {
    if (&b == this) return zero();
    b.sig *= -1;
    operator+=(b); b.sig *= -1;
  }

  void operator*=(int x) {
    if (x < 0) {
      sig *= -1;
      x *= -1;
    }
    remida(n + 1, 0);
    ll ca = 0;
    for (int i = 0; i < n; ++i) {
      ca += ll(v[i])*x;
      v[i] = ca%base;
      ca /= base;
    }
    treuzeros();
  }
};

ostream &operator<<(ostream &out, const BI &b) {
  if (b.sig < 0) out << '-';
  int i = b.v.size() - 1;
  out << b.v[i];
  for (--i; i >= 0; --i) out << setw(9) << setfill('0') << b.v[i];
  return out;
}


typedef stringstream SS;

string tostring(const BI& b) {
  SS ss;
  ss << b;
  string s;
  ss >> s;
  return s;
}

int maxmatch(string a, string b) {
  int na = a.size();
  int nb = b.size();

  for (int i = max(1, na - nb); i < na; ++i)
    if (a.substr(i, na - i) == b.substr(0, na - i))
```

```
      return na - i;
  return 0;
}

BI startpos(string s) {
  int n = s.size();
  if (n == 1) return s[0] - '0' - 1;

  BI res = 0;
  BI nou = 9;
  for (int i = 1; i < n; ++i) {
    BI tmp = nou;
    tmp *= i;
    res += tmp;

    nou *= 10;
  }

  BI zero = 0;
  BI one = 1;

  if (s[0] == '1') s = s.substr(1, s.size() - 1);
  else --s[0];

  BI tmp(s);
  tmp *= n;
  res += tmp;
  return res;
}

int N;
string S;

int naive(string s) {
  string t;
  for (int i = 1; i < 10000; ++i) {
    SS ss;
    ss << i;
    string num;
    ss >> num;
    t += num;
  }
  int n = s.size(), m = t.size();
  for (int i = 0; i + n <= m; ++i) {
    bool ok = true;
    for (int j = 0; ok and j < n; ++j)
      if (s[j] != t[i + j]) ok = false;
    if (ok) return i;
  }
  return -1;
}

int main() {
  BI one = 1;

  cin >> S;
  N = S.size();

  int nv = naive(S);
  if (nv != -1) {
    cout << nv + 1 << endl;
```

```
    return 0;
  }

  if (S == string(N, '0')) {
    BI tmp = startpos("1" + S);
    tmp += one;
    tmp += one;
    cout << tmp << endl;
    return 0;
  }


  bool res = true;
  BI minim = 0;

  // cas 1
  for (int mida = 3; mida <= N; ++mida)
    for (int start = 0; start + mida <= N; ++start) {
      if (S[start] == '0') continue;

      string s = S.substr(start, mida);
      int n = s.size();

      bool ok = true;

      BI bi(s);
      int p = start + n;
      while (ok and p < N) {
        bi += one;
        string t = tostring(bi);
        int m = t.size();
        if (S.substr(p, min(N - p, m)) != t.substr(0, min(N - p, m))) ok = false;
        p += m;
      }

      bi = BI(s);
      p = start;
      while (ok and p > 0) {
        bi -= one;
        string t = tostring(bi);
        int m = t.size();
        p -= m;
        if (S.substr(max(0, p), min(p + m, m)) != t.substr(m - min(p + m, m), min(p
            + m, m))) ok = false;
      }

      if (ok) {
        BI tmp = startpos(s);
        BI tmp2 = start;
        tmp -= tmp2;
        if (res or tmp < minim) {
          minim = tmp;
          res = false;
        }
      }
    }

  // cas 2
  for (int start = 1; start < N; ++start) {
    if (S[start] == '0') continue;
```

```
    string s = S.substr(0, start);
    BI bi(s);
    bi += one;
    s = tostring(bi);
    if (s.size() < start) s = string(start - s.size(), '0') + s;
    s = s.substr(s.size() - start, start);

    string t = S.substr(start, N - start);

    int mx = maxmatch(t, s);

    string r = t + s.substr(mx, s.size() - mx);
    BI tmp = startpos(r);
    BI tmp2 = start;
    tmp -= tmp2;
    if (res or tmp < minim) {
      minim = tmp;
      res = false;
    }
  }

  minim += one;
  cout << minim << endl;
}
```

1167. Bicolored Horses

Time Limit: 1.0 second
Memory Limit: 16 MB

Every day, farmer Ion (this is a Romanian name) takes out all his horses, so they may run and play. When they are done, farmer Ion has to take all the horses back to the stables. In order to do this, he places them in a straight line and they follow him to the stables. Because they are very tired, farmer Ion decides that he doesn't want to make the horses move more than they should. So he develops this algorithm: he places the 1st P_1 horses in the first stable, the next P_2 in the 2nd stable and so on. Moreover, he doesn't want any of the **K** stables he owns to be empty, and no horse must be left outside. Now you should know that farmer Ion only has black or white horses, which don't really get along too well. If there are **i** black horses and **j** white horses in one stable, then the coefficient of unhappiness of that stable is **i*j**. The total coefficient of unhappiness is the sum of the coefficients of unhappiness of every of the K stables.

Determine a way to place the **N** horses into the **K** stables, so that the total coefficient of unhappiness is minimized.

Input

On the 1st line there are 2 numbers: **N** ($1 \le N \le 500$) and **K** ($1 \le K \le N$). On the next N lines there are N numbers. The i-th of these lines contains the color of the i-th horse in the sequence: 1 means that the horse is black, 0 means that the horse is white.

Output

You should only output a single number, which is the minimum possible value for the total coefficient of unhappiness.

Sample

input	output
6 3 1 1 0 1 0 1	2

Hint

Place the first 2 horses in the first stable, the next 3 horses in the 2nd stable and the last horse in the 3rd stable.

Problem Author: Mugurel Ionut Andreica
Problem Source: Romanian Open Contest, December 2001

```
/*
Timus 1167. Bicolored Horses
This problem can be solved with Dynamic Programming.
The time complexity of the implemented algorithm is O(K*N^2).
dp[n][k] is minimum total value of unhappiness after distributing optimally
the horses in positions n..N-1 among K-k stables.
*/
#include <iostream>
using namespace std;

const int INF = 1000000000;

int dp[501][501];
int horse[500];

int main() {
  int N, K;
  cin >> N >> K;

  for (int i = 0; i < N; ++i) cin >> horse[i];

  for (int k = 0; k < K; ++k) dp[N][k] = INF;
  dp[N][K] = 0;

  for (int n = N - 1; n >= 0; --n) {
    for (int k = 0; k < K; ++k) {
      dp[n][k] = INF;
      if (N - n < K - k) continue;
      int a = 0, b = 0;
      for (int i = n; i < N; ++i) {
        if (horse[i] == 0) ++a;
        else ++b;
        dp[n][k] = min(dp[n][k], a*b + dp[i + 1][k + 1]);
      }
    }
    dp[n][K] = INF;
  }

  cout << dp[0][0] << endl;
}
```

Timus 1168. Radio Stations

1168. Radio Stations

Time Limit: 1.0 second
Memory Limit: 16 MB

Along the surface of Romania, there are **K** radio stations positioned at different points and altitudes. Each of these radio stations has a given broadcast radius, that is, the maximum distance it can send its signal to. The government wants to place a radio receiver somewhere on the map, so that it will receive the signals from all the K radio stations: this means that the distance to every radio station should be less or equal to the broadcast radius of that radio station.

The map of Romania is given as a **M*N** matrix, where the value in row i and column j represents the altitude of the corresponding zone. The side of a square in this matrix is 1. All the K radio stations are placed at distinct coordinates on the map and at the same height as the corresponding zone (plus, they are placed exactly in the center of their square). The radio receiver can be placed in the center of any square not occupied by a radio station, at the same altitude of the square or it can be placed higher with an integer number of meters. The radio receiver cannot be placed in a square occupied by a radio station.

Your task is to decide how many possibilities to place the radio receiver the government has. Note that if the radio receiver may be placed in row **i** and column **j** at altitudes **h1** and **h2** (h1 ≠ h2), this counts as 2 different possibilities.

Input

The first line of input contains 3 integers: **M**, **N** (1 ≤ M, N ≤ 50) and **K** (1 ≤ K ≤ min(M*N-1, 1000)), representing the dimensions of the map and the number of radio stations. Next there are M lines each containing N integers, which are the altitudes of the zones on the map (no altitude will be higher than 32000 or lower than 0). After that, there will be K lines, each containing 3 numbers: **i, j** and **R**. i and j will be the location of the radio station on the map and R will be its broadcast radius (R is a real number, not larger than 100000 and not less than 0).

No two radio stations will be placed on the same square.

Output

You should output one integer number, which is the total number of valid possibilities to place the radio receiver on the map.

Sample

input	output
5 5 3 1 2 3 4 5 6 7 8 9 10 1 2 3 4 5 6 7 8 9 10 5 4 3 2 1 1 1 4.3 5 5 4.3 5 1 4.3	4

Hint

The radio receiver can be placed at position (3, 2), with extra height 0 and extra height 1, and at position (3, 3), with extra height 0 and extra height 1. So, there are 4 possible ways to place the receiver.

When you compute distances, be aware that they are distances in a 3D space.

Problem Author: Mugurel Ionut Andreica
Problem Source: Romanian Open Contest, December 2001

```cpp
/*
Timus 1168. Radio Stations
For each cell where a receiver can be positioned, calculate the minimum
and maximum height it can be positioned at, all the intermediate heights
are valid positions too. The time complexity of the proposed algorithm
is O(K*N^2).
*/
#include <cmath>
#include <iostream>
using namespace std;

const double EPS = 1e-8;
const int INF = 1000000000;

int M[50][50];
int SX[1100], SY[1100];
double SR[1100];

int main() {
  int R, C, N;
  cin >> R >> C >> N;

  for (int i = 0; i < R; ++i)
    for (int j = 0; j < C; ++j)
      cin >> M[i][j];

  for (int i = 0; i < N; ++i) {
    cin >> SX[i] >> SY[i] >> SR[i];
    --SX[i]; --SY[i];
  }

  int res = 0;
  for (int i = 0; i < R; ++i)
    for (int j = 0; j < C; ++j) {
      bool ok = true;
      int a = M[i][j], b = INF;
      for (int k = 0; k < N; ++k) {
        int x = SX[k] - i, y = SY[k] - j;
        if (x == 0 and y == 0) {
          ok = false;
          break;
        }
        int dd = x*x + y*y;
        double t = SR[k]*SR[k] - dd;
        if (t < 0) {
          ok = false;
          break;
        }
        int h = int(sqrt(t) + EPS);
        a = max(a, M[SX[k]][SY[k]] - h);
        b = min(b, M[SX[k]][SY[k]] + h);
      }
      if (ok and a <= b) res += b - a + 1;
    }
  cout << res << endl;
}
```

1169. Pairs

Time Limit: 1.0 second
Memory Limit: 16 MB

A Romanian software company has bought **N** computers, which are going to be connected so that they may form a network. A connection can be made between any 2 distinct computers and is bidirectional (if the 2 computers are labeled i and j, then data can be sent both from i to j and from j to i). Your job is to determine a way to connect all the N computers, in such a way that every 2 computers will be able to send data between them (directly or using other computers as intermediate devices).

There is only one extra requirement: the network must contain exactly **K** critical pairs. A pair (i, j) is critical if there exists a connection which, if removed, data communication between i and j will become impossible.

Input

The input consists of 2 integer numbers: N (1 ≤ N ≤ 100), the number of computers the network will contain and K (0 ≤ K ≤ N*(N-1)/2), the number of critical pairs the network will contain.

Output

You should output the connections which form the network, one connection per line. A connection is described by a pair (i, j), which means that i and j are directly connected. The 2 numbers of the pair should be separated by a blank. If you cannot build a network which contains exactly K critical pairs, then you should output -1.

Sample

input	output
7 12	1 2 1 3 2 3 3 4 4 5 4 6 4 7 5 6 5 7 6 7

Problem Author: Mugurel Ionut Andreica
Problem Source: Romanian Open Contest, December 2001

```
/*
Timus 1169. Pairs
Two nodes are related to each other if and only if there is no edge in the graph
such that after removing the edge the two nodes are disconnected. This is an
equivalence relation. Therefore in order to count critical pairs every graph can
be seen as a tree in which nodes are equivalence classes of the original graph,
and the number of critical pairs will be the sum of pairwise product of sizes of
equivalence classes. We can go through every way of splitting the nodes into
equivalence classes and find one (or none) which has K critical pairs. This can
be done through Dynamic Programming or backtracking with pruning (which is what
this code does). An equivalence class can be output as a "circle" of nodes.
*/
#include <iostream>
#include <vector>

using namespace std;

bool rec(int k, vector<int>& a, int pairsum, int sum, int ind) {
  int n = int(a.size());
  if (pairsum > k) return false;
  if (sum == n) {
    return pairsum == k;
  }
  int pairsum_upperbound = pairsum;
  for (int i = sum; i < n; ++i) {
    pairsum_upperbound += i;
  }
  if (pairsum_upperbound < k) return false;
  for (a[ind] = (ind == 0? 1 : a[ind-1]); a[ind] <= n-sum; ++a[ind]) if (a[ind] !=
     2) {
    if (rec(k, a, pairsum+sum*a[ind], sum+a[ind], ind+1)) return true;
  }
  a[ind] = 0;
  return false;
}

int main() {
  int n, k;
  cin >> n >> k;
  vector<int> a(n, 0);
  if (!rec(k, a, 0, 0, 0)) cout << -1 << endl;
  else {
    int sum = 0;
    for (int i = 0; i < n && a[i] > 0; ++i) {
      if (i > 0) cout << sum << " " << sum+1 << endl;
      if (a[i] > 1) for (int j = 0; j < a[i]; ++j) {
        cout << sum+j+1 << " " << sum+((j+1)%a[i])+1 << endl;
      }
      sum += a[i];
    }
  }
}
```

1171. Lost in Space

Time Limit: 1.0 second
Memory Limit: 4 MB

An astronaut of the RSA (Romanian Space Agency) was "forgotten" on the **N-th** level of a Romanian Space Station. He wants to go down to the 1st level, where the communication devices are located, so he can call a space ship to take him home. Unfortunately, the astronaut doesn't know how long the ship will take until it gets to the space station. That's why he wants to gather as much space food as possible before he sends out the message to the ship.

Every level of the space station contains **16** rooms, arranged in 4 rows and 4 columns (numbered accordingly). Every room contains an amount of space food (a number between 1 and 255).

The astronaut may move freely inside the space station, but he is not allowed to visit the same room twice. Moreover, if he went to a lower level, he is not allowed to move back to a higher level. From every room, he could move either north, south, east or west (on the same level) or down (on the same row and column, but at the level below), if there is a door to the level below in that room. For each level, a map is given which tells which rooms have doors to the level below them.

Whenever he enters a room, the astronaut gets the amount of space food found in that room. The food ratio of the astronaut is defined as the total quantity of food gathered during his trip inside the space station divided by the number of days he spends inside the space station before he sends out the signal to the space ship. It is considered that the astronaut spends the 1st day in the room he begins his trip, on the N-th level, and that he gets the amount of space food found in this room during this day. He is only allowed to move once per day.

You have to find a path from the N-th level to the 1st level, which has the maximum food ratio. Note that the astronaut does not have to call the ship as soon as he gets to a room on the 1st level. He may move around the level first, gather the necessary food and only then call the ship.

Input

The first line of input contains a single number: N ($1 \le N \le 16$), the number of levels of the space station. For each level, there will be 8 lines of input containing its description. The first four lines will contain four integers, representing the amount of space food found in the corresponding room on that level (the number found on the j-th position on the i-th line represents the amount of food in the room found on row i and column j on the level). The next four lines will contain four integers, in the range 0..1. 1 means that there is a door from that room to the level below, 0 means that there isn't one. Level 1 will have only 0s on these four lines (there is no level below level 1).

The order of the levels in the input will be from top to bottom (from level N to level 1). The last line of input will contain 2 numbers **r** and **c**, representing the row and column of the room the astronaut is initially in, on level N.

Output

On the 1st line, you should output the maximum possible food ratio for the astronaut, with 4 decimal digits. On the 2nd line you should output the length of his path (print 0 if the astronaut never gets out of the room he is initially found in). If the length of his path is **L**, L > 0, then on the 3rd line of output you should output L characters: 'N', 'E', 'S', 'W' or 'D', each character corresponding to one direction of movement (north, east, south, west and down). If there are more solutions with the same maximum food ratio, then you may output any of them.

Note that, if L is the length of the astronaut's path, then he spends L+1 days before he calls the space ship.

Every test case is guaranteed to have at least 1 solution.

Sample

input	output
2 1 20 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 20 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1	8.6000 4 EDSW

```
/*
Timus 1171. Lost in Space
For each level, compute the following information with backtracking:
dp_pis[level][begin][dist][end] = maximum amount of food that can be
obtained by following a path of length 'dist' in level 'level' begining
at cell 'begin' and ending at cell 'end'.
After that, a DP algorithm can be run to calculate the following:
dp[level][dist][cell] = maximum amount of food that can be obtained
by following a path of length 'dist' starting at 'cell' of 'level',
ending somewhere in the 1st level.
Now, for each possible distance from the Nth to the 1st level,
we know the maximum amount of food we can get following a path of that
length. The answer is the path with optimal ratio. Additional information
should be stored when calculating dp_pis[] and dp[] in order to
recover the path efficiently.
*/
#include <iostream>
using namespace std;

typedef unsigned int uint;

const int INF = 1000000000;
const int diri[4] = { -1, 0, 1, 0 };
const int dirj[4] = { 0, 1, 0, -1 };
const char dirnom[4] = { 'N', 'E', 'S', 'W' };

int dp_pis[16][16][16][16];       // pis, inici, dist, desti
uint dp_pis_cami[16][16][16][16]; // cami

int dp[16][256][16];      // pis, dist, casella
int dp_quin[16][256][16];  // casella on vaig abans de baixar
int dp_pass[16][256][16];  // passos que faig al pis per anar a la casella

int N;
int food[16][4][4];
int gate[16][4][4];

inline int conv(int x, int y) {
  return 4*x + y;
}

int pis, ini;
void back(int vist, int d, int s, int x, int y, uint cami) {
  if (x < 0 or 4 <= x or y < 0 or 4 <= y) return;
  int c = conv(x, y);
  if ((vist>>c)&1) return;
  vist |= 1<<c;
  s += food[pis][x][y];

  if (s > dp_pis[pis][ini][d][c]) {
    dp_pis[pis][ini][d][c] = s;
    dp_pis_cami[pis][ini][d][c] = cami;
  }

  for (int k = 0; k < 4; ++k) {
    uint t = cami;
    t |= uint(k)<<(2*d);
    int i = x + diri[k];
    int j = y + dirj[k];
    back(vist, d + 1, s, i, j, t);
  }
```

```
}

int main() {
  cout.setf(ios::fixed);
  cout.precision(4);

  cin >> N;
  for (int k = N - 1; k >= 0; --k) {
    for (int i = 0; i < 4; ++i)
      for (int j = 0; j < 4; ++j)
        cin >> food[k][i][j];
    for (int i = 0; i < 4; ++i)
      for (int j = 0; j < 4; ++j)
        cin >> gate[k][i][j];
  }
  int x, y;
  cin >> x >> y;
  --x; --y;

  for (pis = 0; pis < N; ++pis)
    for (int i = 0; i < 4; ++i)
      for (int j = 0; j < 4; ++j) {
        ini = conv(i, j);
        for (int a = 0; a < 16; ++a)
          for (int b = 0; b < 16; ++b)
            dp_pis[pis][ini][a][b] = -INF;
        back(0, 0, 0, i, j, 0);
      }

  for (int n = 0; n < N; ++n) {
    for (int d = 0; d < 256; ++d)
      for (int i = 0; i < 4; ++i)
        for (int j = 0; j < 4; ++j) {
          int c = conv(i, j);
          dp[n][d][c] = -INF;
          for (int k = 0; k < 16 and k <= d; ++k)
            for (int ii = 0; ii < 4; ++ii)
              for (int jj = 0; jj < 4; ++jj) {
                int p = conv(ii, jj);
                int t = dp_pis[n][c][k][p];
                if (n == 0) {
                  if (k != d) continue;
                }
                else {
                  if (k == d) continue;
                  if (gate[n][ii][jj] == 0) continue;
                  t += dp[n - 1][d - k - 1][p];
                }
                if (t > dp[n][d][c]) {
                  dp[n][d][c] = t;
                  dp_quin[n][d][c] = p;
                  dp_pass[n][d][c] = k;
                }
              }
        }
  }

  double maxratio = 0;
  int quin_d = 0;
  for (int d = 0; d < 256; ++d) {
    double r = dp[N - 1][d][conv(x, y)]/double(d + 1);
```

```
    if (r > maxratio) {
      maxratio = r;
      quin_d = d;
    }
  }

  cout << maxratio << endl;
  cout << quin_d << endl;

  int c = conv(x, y), d = quin_d;
  for (int n = N - 1; n >= 0; --n) {
    int p = dp_quin[n][d][c];
    int k = dp_pass[n][d][c];
    uint cami = dp_pis_cami[n][c][k][p];
    for (int i = 0; i < k; ++i)
      cout << dirnom[(cami>>(2*i))&3];
    if (n) cout << 'D';
    c = p;
    d = d - k - 1;
  }
  cout << endl;
}
```

1172. Ship Routes

Time Limit: 1.0 second
Memory Limit: 16 MB

A Romanian tourist went on a trip to the Mediterranean Sea. He arrived to one of the cities of the 3 islands he is going to visit. Every island has exactly **N** cities and they are all ports. The tourist plans to begin his journey from the city he is in, visit all the other **3*N-1** cities exactly once and then return to the starting city so he may go back home after that.

Unfortunately, there are cannibals along the roads on all the 3 islands. That's why travelling on the road between 2 cities on the same island is very dangerous and, consequently, prohibited. Hopefully, there are always ship routes. Every pair of cities which are not on the same island is connected by such a ship route. There are no routes between cities which are on the same island.

The tourist wants to know in how many ways he can plan his journey through the 3 islands.

Input

The input contains a single number: N (1 ≤ N ≤ 30), the number of cities on each of the 3 islands.

Output

You should output a single number: the number of ways the tourist can plan his trip. Note that 2 trips are identical if the successions of the 3*N cities are identical or if the succession of the 3*N cities of the first trip is the same as the succession of the 3*N cities of the 2nd trip, read backwards (for instance, if every island had 1 city, numbered according to the island's number, the trips 1-2-3-1 and 1-3-2-1 would be identical).

Sample

input	output
2	16

Problem Author: Mugurel Ionut Andreica
Problem Source: Romanian Open Contest, December 2001

```
/*
Timus 1172. Ship Routes
This problem can be solved with Dynamic Programming. Be aware that the answer
can be a very large integer.
fun(a, b, c, p) = #{ ways to complete a trip starting at the island 'p', having
'a' cities to explore in island 0, 'b' in island 1, and 'c' in island '2' }
The correct answer to the problem is fun(n - 1, n, n, 0)/2, dividing by 2 we avoid
counting one sequence and its reverse as different sequences.
*/
#include <iostream>
#include <cstring>
#include <vector>
#include <iomanip>
using namespace std;

typedef vector<int> VE;
typedef long long ll;
const ll base = 1000000000;

class BI { public:
  VE v;
  int n;
  int sig;

  void zero() {
    v = VE(1, 0);
    n = sig = 1;
  }

  void remida(int i, int que=0) {
    v.resize(n = i, que);
  }

  void treuzeros() {
    while (--n and !v[n]);
    remida(n + 1);
    if (n == 1 and !v[0]) sig = 1;
  }

  inline int compabs(const BI &b) const {
    if (n != b.n) return n - b.n;
    for (int i = n - 1; i >= 0; --i)
      if (v[i] != b.v[i]) return v[i] - b.v[i];
    return 0;
  }

  inline int compara(const BI &b) const {
    if (sig != b.sig) return sig - b.sig;
    return sig*compabs(b);
  }

  inline int dig(int i) {
    return (i < n ? v[i] : 0);
  }

  void suma(BI &b) {
    if (n < b.n) remida(b.n, 0);
    int ca = 0;
    for (int i = 0; i < n; ++i) {
      v[i] += b.dig(i) + ca;
      ca = v[i]/base;
```

```
    v[i] %= base;
  }
  if (ca) remida(n + 1, ca);
}

void resta(BI &b) {
  int ca = 0;
  for (int i = 0; i < n; ++i) {
    v[i] += base - b.dig(i) + ca;
    ca = v[i]/base - 1;
    v[i] %= base;
  }
  treuzeros();
}

BI() {
  zero();
}

BI(const BI &b) {
  *this = b;
}

BI(ll x) {
  sig = (x < 0 ? -1 : 1);
  x *= sig;
  if (x < base) {
    v = VE(1, x);
    n = 1;
  }
  else {
    v = VE(2);
    v[0] = x%base;
    v[1] = x/base;
    n = 2;
  }
}

void operator=(const BI &b) {
  v = b.v;
  n = b.n;
  sig = b.sig;
}

void operator+=(BI &b) {
  if (sig == b.sig) return suma(b);
  if (compabs(b) >= 0) return resta(b);
  BI aux(b);
  aux.resta(*this);
  *this = aux;
}

void operator-=(BI &b) {
  if (&b == this) return zero();
  b.sig *= -1;
  operator+=(b);
  b.sig *= -1;
}

void operator*=(int x) {
  if (x < 0) {
```

```
      sig *= -1;
      x *= -1;
    }
    remida(n + 1, 0);
    ll ca = 0;
    for (int i = 0; i < n; ++i) {
      ca += ll(v[i])*x;
      v[i] = ca%base;
      ca /= base;
    }
    treuzeros();
  }

  void operator/=(int x) {
    if (x < 0) {
      sig *= -1;
      x *= -1;
    }
    ll ca = 0;
    for (int i = n - 1; i >= 0; --i) {
      ca += v[i];
      v[i] = ca/x;
      ca %= x;
      ca *= base;
    }
    treuzeros();
  }
};

ostream &operator<<(ostream &out, const BI &b) {
  if (b.sig < 0) out << '-';
  int i = b.v.size() - 1;
  out << b.v[i];
  for (--i; i >= 0; --i) out << setw(9) << setfill('0') << b.v[i];
  return out;
}

int tdp[31][31][31][3];
BI dp[31][31][31][3];

BI fun(int a, int b, int c, int p) {
  if (a == 0 and b == 0 and c == 0) {
    if (p == 0) return 0;
    return 1;
  }

  if (tdp[a][b][c][p]) return dp[a][b][c][p];

  BI res = 0;
  if (a > 0 and p != 0) {
    BI tmp = fun(a - 1, b, c, 0);
    tmp *= a;
    res += tmp;
  }
  if (b > 0 and p != 1) {
    BI tmp = fun(a, b - 1, c, 1);
    tmp *= b;
    res += tmp;
  }
  if (c > 0 and p != 2) {
    BI tmp = fun(a, b, c - 1, 2);
```

```
    tmp *= c;
    res += tmp;
  }

  tdp[a][b][c][p] = 1;
  dp[a][b][c][p] = res;
  return res;
}

int main() {
  int n;
  cin >> n;
  BI tmp = fun(n - 1, n, n, 0);
  tmp /= 2;
  cout << tmp << endl;
}
```

Timus 1173. Lazy Snail

1173. Lazy Snail

Time Limit: 1.0 second
Memory Limit: 16 MB

Here, in Romania, all snails are lazy. Take Wally the Snail, for example. He has to visit **N** friends which are located at distinct coordinates in the plane. But since he is so lazy, he doesn't want to leave his house. He said that he will go visit his friends if someone can show him the right path to follow.

He wants to leave his house, visit all of his friends exactly once and then return to his house. Between 2 friends' houses or between his house and a friend's house, he walks on the straight line which connects them. 'Is that all ?', someone asked. Wally realized that this would be too easy, so he added that, during his trip, no two line segments along which he travels should cross (except for every 2 consecutive segments, which cross at one end). You must find a path for Wally, so he can go visit all of his friends (although he doesn't want to).

Input

On the 1st line of input, there will be 2 real numbers: X and Y, separated by a blank, representing the coordinates of Wally's house. On the 2nd line, there will be an integer number: N ($2 \le N \le 1000$), the number of friends Wally has to visit. On the next N lines, there will be 3 numbers, separated by blanks: X , Y and ID. ID will be an integer number, representing the ID of one of Wally's friends. X and Y will be 2 real numbers, representing the coordinates of Wally's friend's house (they will be given with at most 3 decimal digits and will be in the range -100000 .. 100000).

All IDs are unique, between 1 and N. No 3 friends (including Wally) have their houses on the same straight line.

Output

You should output N+2 lines: the IDs of the friends whose houses Wally is about to visit, in the order he visits them. Start with Wally's ID, continue with the ID of the friend he visits first and so on. Finish with Wally's ID. Wally has ID 0.

If there is no solution, then print a single line, containing the number -1.

Sample

input	output
0 0 3 3 3 1 6 0 2 6 2 3	0 1 3 2 0

Problem Author: Mugurel Ionut Andreica
Problem Source: Romanian Open Contest, December 2001

```
/*
Timus 1173. Lazy Snail
The easiest way to solve this problem is probably sorting all the points around
the snail's house by the angle between the X-axis and the segment snail-point,
and visiting the points in that order.
The proposed algorithm does the following:
1. Find the upper/lower convex hull of all the points (including snail's house).
2. Order the rest of the points lexicographically.
One possible solution is to follow the points of (2) in that order, and complete
the cycle with the points in (1).
*/
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
using namespace std;

#define X first
#define Y second

typedef pair<double, double> P;
typedef pair<P, int> PP;
typedef vector<PP> Vpp;
typedef vector<int> Vi;

double cross(P a, P b, P c) {
  return (c.X - b.X)*(a.Y - b.Y) - (c.Y - b.Y)*(a.X - b.X);
}

void chull(Vpp v, Vpp& u) {
  int n = v.size();
  u = Vpp(n);
  int k1 = 0;
  for (int i = 0; i < n; ++i) {
    while (k1 > 1 and cross(u[k1 - 2].X, u[k1 - 1].X, v[i].X) <= 0) --k1;
    u[k1++] = v[i];
  }
  u.resize(k1);
}

int main() {
  double x0, y0;
  cin >> x0 >> y0;

  int n;
  cin >> n;
  ++n;

  Vpp v(n);
  v[0] = PP(P(x0, y0), 0);
  for (int i = 1; i < n; ++i)
    cin >> v[i].X.X >> v[i].X.Y >> v[i].Y;
  sort(v.begin(), v.end());

  Vpp u;
  chull(v, u);
  int m = u.size();

  Vi res(n);
  int k = 0;
  for (int i = 0; i < m; ++i) res[k++] = u[i].Y;
```

```
  Vi vist(n, 0);
  for (int i = 0; i < m; ++i) vist[u[i].Y] = 1;

  for (int i = n - 1; i >= 0; --i)
    if (vist[v[i].Y] == 0) res[k++] = v[i].Y;

  int p = -1;
  for (int i = 0; p == -1 and i < n; ++i)
    if (res[i] == 0) p = i;

  for (int i = 0; i < n; ++i)
    cout << res[(p + i)%n] << endl;
  cout << 0 << endl;
}
```

Timus 1174. Weird Permutations

1174. Weird Permutations

Time Limit: 0.5 second
Memory Limit: 16 MB

Three Romanian programmers developed this new algorithm which generates all the N! permutations with N elements in a specific order, they called the **transposition order**. The algorithm starts with the permutation 1 2 3 .. N. Then it chooses a pair of two adjacent elements (that is, two elements which are located on consecutive positions) and switches them. This way, they get a new permutation. They do the same for this new permutation and they obtain a new one and so on, until all the N! permutations are generated. You realize that the algorithm must be pretty smart in order to generate all the N! permutations exactly once (without repetitions).

Hopefully, your task will not be to write such an algorithm. In fact, you are given the files perm.pas and perm.cpp, which are two implementations of this algorithm (in Pascal and C++). They read the integer **N** ($1 \le N \le 100$) from the keyboard and print to the file **perm.txt** all the N! permutations, one per line, in the order in which the algorithm generates them.

What you have to do is, given a permutation, to find out its index in the list of permutations generated by the algorithm.

Perm.pas

```
const
  fileout = 'perm.txt';
  MAXN = 100;

var
  fout :text;
  n, i :integer;
  permut :array [1..MAXN] of integer;
  position :array [1..MAXN] of integer;
  dir :array [1..MAXN] of integer;

procedure PrintPermutation;
begin
  for i := 1 to n do
    write(fout, ' ', permut[i]);
  writeln(fout);
end;

procedure Switch(p1, p2 :integer);
var
  xch :integer;
begin
  xch := permut[p1];
  permut[p1] := permut[p2];
  permut[p2] := xch;
  position[permut[p1]] := p1;
  position[permut[p2]] := p2;
end;

procedure GeneratePermutation(nn :integer);
var ii :integer;
begin
  if (nn = n + 1) then
    PrintPermutation
  else
  begin
    GeneratePermutation(nn + 1);
    for ii := 1 to nn - 1 do
    begin
      Switch(position[nn],
             position[nn] + dir[nn]);
      GeneratePermutation(nn + 1);
    end;
    dir[nn] := -dir[nn];
  end;
end;

begin
  readln(n);
  for i := 1 to n do
  begin
    permut[i] := i;
    position[i] := i;
    dir[i] := -1;
  end;

  assign(fout, fileout);
  rewrite(fout);

  GeneratePermutation(1);

  close(fout);
end.
```

Perm.cpp

```
#include <stdio.h>

const char *fileout = "perm.txt";
const int MAXN = 100;

FILE *fout;
int n, i;
int permut[MAXN + 1];
int position[MAXN + 1];
int dir[MAXN + 1];

void PrintPermutation()
{
  for (i = 1; i <= n; i++)
    fprintf(fout, " %d", permut[i]);
  fprintf(fout, "\n");
}

void Switch(int p1, int p2)
{
  int xch = permut[p1];
  permut[p1] = permut[p2];
  permut[p2] = xch;
  position[permut[p1]] = p1;
  position[permut[p2]] = p2;
}

void GeneratePermutation(int nn)
{
  int ii;

  if (nn == n + 1)
    PrintPermutation();
  else
  {
    GeneratePermutation(nn + 1);
    for (ii = 1; ii <= nn - 1; ii++)
    {
      Switch(position[nn],
             position[nn] + dir[nn]);
      GeneratePermutation(nn + 1);
    }
    dir[nn] = -dir[nn];
  }
}

int main()
{
  scanf("%d", &n);
  for (i = 1; i <= n; i++)
  {
    permut[i] = i;
    position[i] = i;
    dir[i] = -1;
  }

  fout = fopen(fileout, "wt");

  GeneratePermutation(1);

  fclose(fout);
  return 0;
}
```

Input

The first line contains a single integer: N (1 ≤ N ≤ 100). The 2nd line contains N integers separated by blanks. They are the given permutation with N elements.

Output

Print one single integer, which will be the index of the permutation in the list of N! permutations generated by the algorithm described above.

Sample

input	output
4 2 3 1 4	17

Hint

Run the 2 given programs for N=4 and you will notice that the permutation 2 3 1 4 will be on the 17th line of the file perm.txt.

Problem Author: Mugurel Ionut Andreica. The Pascal and C++ programs of the three programmers are improved (and, obviously, modified) versions of two programs written by Frank Ruskey (I found them on the Web). So, thanks Frank!
Problem Source: Romanian Open Contest, December 2001

```
/*
Timus 1174. Weird Permutations
For k = 2, 3, 4, ..., n compute the position of k among the integers
{1, 2, 3, ..., k}. Using this we can know in which of the k blocks of size n!/k!
in the currently considered interval is our permutation. The block can be either
pos or (k+1-pos) depending on the parity of the position of the currently
considered interval. Since the output may not fit in a 64-bit integer, Big
Integer arithmetic is needed.
*/
#include <iostream>
#include <cstring>
#include <vector>
#include <iomanip>

using namespace std;

typedef vector<int> VE;
typedef long long ll;
const ll base = 1000000000;

class BI { public:
  VE v;
  int n;
  int sig;

  void zero() {
    v = VE(1, 0);
    n = sig = 1;
  }

  void remida(int i, int que=0) {
    v.resize(n = i, que);
  }

  void treuzeros() {
    while (--n and !v[n]);
    remida(n + 1);
    if (n == 1 and !v[0]) sig = 1;
  }

  inline int compabs(const BI &b) const {
    if (n != b.n) return n - b.n;
    for (int i = n - 1; i >= 0; --i)
      if (v[i] != b.v[i]) return v[i] - b.v[i];
    return 0;
  }

  inline int compara(const BI &b) const {
    if (sig != b.sig) return sig - b.sig;
    return sig*compabs(b);
  }

  inline bool operator<=(const BI &b) const { return compara(b) <= 0; }
  inline bool operator>=(const BI &b) const { return compara(b) >= 0; }

  inline int dig(int i) {
    return (i < n ? v[i] : 0);
  }

  void suma(BI &b) {
    if (n < b.n) remida(b.n, 0);
```

```
  int ca = 0;
  for (int i = 0; i < n; ++i) {
    v[i] += b.dig(i) + ca;
    ca = v[i]/base;
    v[i] %= base;
  }
  if (ca) remida(n + 1, ca);
}

void resta(BI &b) {
  int ca = 0;
  for (int i = 0; i < n; ++i) {
    v[i] += base - b.dig(i) + ca;
    ca = v[i]/base - 1;
    v[i] %= base;
  }
  treuzeros();
}

BI() {
  zero();
}

BI(const BI &b) {
  *this = b;
}

BI(ll x) {
  sig = (x < 0 ? -1 : 1);
  x *= sig;
  if (x < base) {
    v = VE(1, x);
    n = 1;
  }
  else {
    v = VE(2);
    v[0] = x%base;
    v[1] = x/base;
    n = 2;
  }
}

void operator=(const BI &b) {
  v = b.v;
  n = b.n;
  sig = b.sig;
}

void operator+=(BI &b) {
  if (sig == b.sig) return suma(b);
  if (compabs(b) >= 0) return resta(b);
  BI aux(b);
  aux.resta(*this);
  *this = aux;
}

void operator-=(BI &b) {
  if (&b == this) return zero();
  b.sig *= -1;
  operator+=(b);
  b.sig *= -1;
```

```
  }

  void operator*=(int x) {
    if (x < 0) {
      sig *= -1;
      x *= -1;
    }
    remida(n + 1, 0);
    ll ca = 0;
    for (int i = 0; i < n; ++i) {
      ca += ll(v[i])*x;
      v[i] = ca%base;
      ca /= base;
    }
    treuzeros();
  }

  void operator/=(int x) {
    if (x < 0) {
      sig *= -1;
      x *= -1;
    }
    ll ca = 0;
    for (int i = n - 1; i >= 0; --i) {
      ca += v[i];
      v[i] = ca/x;
      ca %= x;
      ca *= base;
    }
    treuzeros();
  }

  void operator/=(BI &b) {
    if (compabs(b)<0) return zero();
    if (b.n==1) *this/=b.sig*b.v[0];
    else {
      int st = sig*b.sig, sb = b.sig; sig = b.sig = 1;
      vector<BI> VB, pot2;
      VB.push_back(b); pot2.push_back(1);
      BI curr=0;
      while (VB[VB.size()-1]<=*this) {
        BI ultimo=VB[VB.size()-1]; ultimo+=ultimo; VB.push_back(ultimo);
        ultimo=pot2[pot2.size()-1]; ultimo+=ultimo; pot2.push_back(ultimo);
      }
      curr+=pot2[pot2.size()-2]; (*this)-=VB[VB.size()-2];
      while((*this)>=b) {
        int i=0;
        while (VB[i]<=(*this)) i++;
        curr+=pot2[i-1]; (*this)-=VB[i-1];
      }
      (*this)=curr; sig = st; b.sig = sb;
    }
  }

  ll mod(int x) {
    if (x < 0) x *= -1;
    ll ca = 0;
    for (int i = n-1; i >= 0; --i) { ca *= base; ca += v[i]; ca %= x; }
    return ca;
  }
};
```

```
ostream &operator<<(ostream &out, const BI &b) {
  if (b.sig < 0) out << '-';
  int i = b.v.size() - 1;
  out << b.v[i];
  for (--i; i >= 0; --i) out << setw(9) << setfill('0') << b.v[i];
  return out;
}

int main() {
  int n;
  cin >> n;
  vector<BI> qfac(n+1);
  qfac[0] = 1;
  for (int i = 1; i <= n; ++i) {
    qfac[0] *= i;
  }
  for (int i = 1; i <= n; ++i) {
    qfac[i] = qfac[i-1];
    qfac[i] /= i;
  }
  vector<int> p(n);
  for (int i = 0; i < n; ++i) {
    cin >> p[i];
  }

  BI ans = 0;
  for (int i = 2; i <= n; ++i) {
    int pos = 0;
    for (int j = 0; j < n && p[j] != i; ++j) {
      if (p[j] < i) ++pos;
    }
    BI step = qfac[i];
    BI parity = ans;
    parity /= qfac[i-1];
    if (parity.mod(2) == 0) step *= (i-1-pos);
    else step *= pos;
    ans += step;
  }
  BI one = 1;
  ans += one;
  cout << ans << endl;
}
```

1175. Strange Sequence

Time Limit: 1.0 second

Memory Limit: 2 MB

You have been asked to discover some important properties of one strange sequences set. Each sequence of the parameterized set is given by a recurrent formula:

$$X_{n+1} = F(X_{n-1}, X_n),$$

where $n > 1$, and the value of $F(X,Y)$ is evaluated by the following algorithm:

1. find $H = (A_1*X*Y + A_2*X + A_3*Y + A_4)$;
2. if $H > B_1$ then H is decreased by C until $H \le B_2$;
3. the resulting value of H is the value of function F.

The sequence is completely defined by nonnegative constants A_1, A_2, A_3, A_4, B_1, B_2 and C.

One may easily verify that such sequence possess a property that $X_{p+n} = X_{p+q+n}$ for appropriate large enough positive integers p and q and for all $n \ge 0$. You task is to find the minimal p and q for the property above to hold. Pay attention that numbers p and q are well defined and do not depend on way minimization is done.

Input

The first line contains seven integers: A_1, A_2, A_3, A_4, B_1, B_2 and C. The first two members of sequence (X_1 and X_2) are placed at the second line. You may assume that all intermediate values of H and all values of F fit in range [0..100000].

Output

An output should consist of two integers (p and q) separated by a space.

Sample

input	output
0 0 2 3 20 5 7 0 1	2 3

Problem Author: Alexander Klepinin

Problem Source: Third USU personal programming contest, Ekaterinburg, Russia, February 16, 2002

```cpp
/*
Timus 1175. Strange Sequence
In order to find the smallest p and q such that X_p = X_{p+q} we can compare
X_{2^k} to X_{2^k + i) for all i < 2^k, incrementing k until we find a match.
This algorithm will stop in less than 2*max(p, q) steps because when 2^k > p
and i = q the condition X_{2^k} = X_{2^k + i} will be satisfied. Once the value
of q is known, we can iterate through the pairs (X_j, X_{j+q}) incrementing j
until X_j = X_{j+q}. The condition will be satisfied as soon as j = p.
*/
#include <iostream>

#define fr first
#define sc second

using namespace std;

typedef pair<int, int> PII;

int A1, A2, A3, A4, B1, B2, C;

inline int F(int x, int y) {
  int h = A1*x*y + A2*x + A3*y + A4;
  if (h > B1) h -= ((h-B2+C-1)/C)*C;
  return h;
}

inline PII nxt_pair(PII p) {
  return PII(p.sc, F(p.fr, p.sc));
}

int main() {
  cin >> A1 >> A2 >> A3 >> A4 >> B1 >> B2 >> C;
  PII p1;
  cin >> p1.fr >> p1.sc;
  int i = 1, j = 2, pow2 = 2;
  for (PII pi = p1, pj = nxt_pair(p1); pj != pi; ++j) {
    if (j == pow2) {
      pi = pj;
      i = j;
      pow2 *= 2;
    }
    pj = nxt_pair(pj);
  }
  int q = j-i;
  PII piq = p1;
  for (int k = 0; k < q; ++k) {
    piq = nxt_pair(piq);
  }
  i = 1;
  for (PII pi = p1; pi != piq; ++i) {
    pi = nxt_pair(pi);
    piq = nxt_pair(piq);
  }
  int p = i;
  cout << p << " " << q << endl;
}
```

1176. Hyperchannels

Time Limit: 1.0 second

Memory Limit: 16 MB

The Galaxy Empire consists of N planets. Hyperchannels exist between most of the planets. New Emperor urged to extend hyperchannels network in such a way, that he can move from any planet to any other using no more than one channel. One can pass through the channel only in one direction.

The last channel-establishing ship is located on the base near planet A. This ship can't pass through the existing channel, it always establishes a new one. But presence of two channels connecting the same planets in one direction makes navigation too difficult, almost impossible. The problem is to find a route for this ship to establish all necessary channels with no excessive ones. In the end of this route ship should return to the base.

Input

First line contains integer $N \le 1000$ and number of the planet A ($A \le N$) where the base is situated. Each of the following N lines contain N numbers, the j-th number of the i-th line equals to 1 if there exists channel from planet i to planet j, and equals to 0 otherwise. It is known, that Empire can fulfill its need of hyperchannels by establishing no more than 32000 new ones.

Output

Output should contain the sequence in which channels should be established. Each line should contain two integers — numbers of source and destination planet of channel. You may assume, that solution always exists.

Sample

input	output
4 2 0 0 1 0 0 0 1 0 1 1 0 1 0 0 1 0	2 4 4 1 1 2 2 1 1 4 4 2

Problem Author: Pavel Atnashev

Problem Source: Third USU personal programming contest, Ekaterinburg, Russia, February 16, 2002

```cpp
/*
Timus 1176. Hyperchannels
First, construct the complement graph, which is a graph with the same vertices
as the original, but here two vercies are adjacent if and only if they are not
adjacent in the original graph.
Now find an eulerian cycle on the complement graph and output it in the proper
format.
*/
#include <iostream>
#include <cstdio>
using namespace std;

int N, M;
int mat[1010][1010];
int pos[1010];
int tour[32010];

void eulerian_tour(int n) {
  while (pos[n] < N) {
    while (pos[n] < N and mat[n][pos[n]] == 0) ++pos[n];
    if (pos[n] < N) {
      int t = pos[n]++;
      eulerian_tour(t);
    }
  }
  tour[M++] = n;
}

int main() {
  int s;
  scanf("%d%d", &N, &s);
  --s;

  for (int i = 0; i < N; ++i)
    for (int j = 0; j < N; ++j) {
      int t;
      scanf("%d", &t);
      if (i != j and t == 0) mat[i][j] = 1;
    }

  eulerian_tour(s);

  for (int i = M - 2; i >= 0; --i)
    printf("%d %d\n", tour[i + 1] + 1, tour[i] + 1);
}
```

1178. Akbardin's Roads

Time Limit: 1.0 second
Memory Limit: 16 MB

Great Akbardin decided to build new roads in his caliphate. He wants to build minimal number of roads so that one can travel from any town to any other using only these roads. But this problem is too difficult for him and his mathematicians. So, at first, they decided to build straight roads between towns in such a way, that every town becomes connected with only one other. Because crossroads make movement dangerous, no two roads should intersect.

You task is to make plan of the roads being given coordinates of towns.

Input

First line contains an even integer N ($N \le 10000$) — the number of towns. Each of the next N lines contains pair of integers — coordinates of i-th town x_i, y_i ($-10^9 < x_i, y_i < 10^9$). No three towns lay on one line.

Output

Output $N/2$ lines with description of one road on each. Road is identified by pair of towns it connects.

Sample

input	output
4 0 2 1 1 3 4 4 4	1 3 2 4

Problem Author: Pavel Atnashev
Problem Source: Third USU personal programming contest, Ekaterinburg, Russia, February 16, 2002

```
/*
Timus 1178. Akbardin's Roads
Sort the points lexicographically and make a road between
1st and 2nd points, between 3rd and 4th, and so on. This
way the roads don't intersect.
*/
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>
using namespace std;

#define X first
#define Y second

typedef pair<int, int> P;
typedef pair<P, int> PP;
typedef vector<PP> Vpp;

int main() {
  int n;
  cin >> n;

  Vpp v(n);
  for (int i = 0; i < n; ++i) {
    cin >> v[i].X.X >> v[i].X.Y;
    v[i].Y = i;
  }
  sort(v.begin(), v.end());

  for (int i = 0; i < n; i += 2)
    cout << v[i].Y + 1 << " " << v[i + 1].Y + 1 << endl;
}
```

1179. Numbers in Text

Time Limit: 1.0 second
Memory Limit: 16 MB

During building of roads, Akbardin read many statistical reports. Each report contained a lot of numbers. But different reports contained numbers in different numeric systems. And Akbardin asks his mathematicians a question – in what numeric system text contains maximal amount of numbers. Number is a sequence of digits, with non-digits to the left and right. Capital Latin letters are used in k-based system with k > 10 ('A' = 10, 'B' = 11, …, 'Z' = 35).

You task is to help mathematicians to solve this problem and save their heads.

Input

Text consists of digits, capital Latin letters, spaces and line breaks. Size of input doesn't exceed 1 Mb.

Output

Output should contain two integers: base of numeric system K ($2 \le K \le 36$) and amount of numbers. If more than one answer is possible, output the one with a less K.

Sample

input	output
01234B56789 AZA	11 4

Problem Author: Pavel Atnashev
Problem Source: Third USU personal programming contest, Ekaterinburg, Russia, February 16, 2002

```
/*
Timus 1179. Numbers in Text
Just count, for each base K (2<=K<=36), the number of "connected components"
formed by adjacent digits with a value less than K.
*/
#include <iostream>
#include <string>
using namespace std;

inline int tonum(char c) {
  if ('0' <= c and c <= '9') return c - '0';
  return 10 + c - 'A';
}

int main() {
  string s, line;
  while (getline(cin, line)) s += line + "*";

  int n = s.size();
  for (int i = 0; i < n; ++i) {
    if ('0' <= s[i] and s[i] <= '9') continue;
    if ('A' <= s[i] and s[i] <= 'Z') continue;
    s[i] = 'Z' + 1;
  }

  int maxim = 0, base = 2;
  for (int k = 2; k <= 36; ++k) {
    int t = 0;
    for (int i = 0; i < n; ++i)
      if ((i == 0 or k <= tonum(s[i - 1])) and tonum(s[i]) < k) ++t;
    if (t > maxim) {
      maxim = t;
      base = k;
    }
  }

  cout << base << " " << maxim << endl;
}
```

1180. Stone Game

Time Limit: 1.0 second
Memory Limit: 16 MB

Two Nikifors play a funny game. There is a heap of N stones in front of them. Both Nikifors in turns take some stones from the heap. One may take any number of stones with the only condition that this number is a nonnegative integer power of 2 (e.g. 1, 2, 4, 8 etc.). Nikifor who takes the last stone wins. You are to write a program that determines winner assuming each Nikifor does its best.

Input

An input contains the only positive integer number N (condition $N \le 10^{250}$ holds).

Output

The first line should contain 1 in the case the first Nikifor wins and 2 in case the second one does. If the first Nikifor wins the second line should contain the minimal number of stones he should take at the first move in order to guarantee his victory.

Sample

input	output
8	1 2

Problem Author: Dmitry Filimonenkov
Problem Source: Third USU personal programming contest, Ekaterinburg, Russia, February 16, 2002

```cpp
/*
Timus 1180. Stone Game
The answer depends only on the number stones modulo 3.
It can be easily proved that if the modulo is 0, then
the game is in a loosing state and the first player loses,
otherwise the first player wins and he must take from the
heap a number of stones that makes the size be 0 modulo 3.
*/
#include <iostream>
#include <string>
using namespace std;

int main() {
  string s;
  cin >> s;

  int n = s.size(), m = 0;
  for (int i = 0; i < n; ++i)
    m = (10*m + s[i] - '0')%3;

  if (m == 0) cout << 2 << endl;
  else if (m == 1) cout << 1 << endl << 1 << endl;
  else cout << 1 << endl << 2 << endl;
}
```

1182. Team Them Up!

Time Limit: 1.0 second
Memory Limit: 16 MB

Your task is to divide a number of persons into two teams, in such a way, that:

- everyone belongs to one of the teams;
- every team has at least one member;
- every person in the team knows every other person in his team;
- teams are as close in their sizes as possible.

This task may have many solutions. You are to find and output any solution, or to report that the solution does not exist.

Input

For simplicity, all persons are assigned a unique integer identifier from 1 to N.

The first line contains a single integer number N ($2 \le N \le 100$) - the total number of persons to divide into teams, followed by N lines - one line per person in ascending order of their identifiers. Each line contains the list of distinct numbers A_{ij} ($1 \le A_{ij} \le N$, $A_{ij} \ne i$) separated by spaces. The list represents identifiers of persons that i^{th} person knows. The list is terminated by 0.

Output

If the solution to the problem does not exist, then write a single message “No solution” (without quotes). Otherwise write a solution on two lines. On the first line write the number of persons in the first team, followed by the identifiers of persons in the first team, placing one space before each identifier. On the second line describe the second team in the same way. You may write teams and identifiers of persons in a team in any order.

Sample

input	output
5 2 3 5 0 1 4 5 3 0 1 2 5 0 1 2 3 0 4 3 2 1 0	3 1 3 5 2 2 4

Problem Author: Vladimir Kotov, Roman Elizarov
Problem Source: 2001-2002 ACM Northeastern European Regional Programming Contest

```
/*
Timus 1182. Team Them Up!
Consider an undirected graph with N nodes, one for each person, with
one edge between each pair of persons who cannot be in the same group
(this happens when they are not friends mutually).
If at least one of the connected components of the graph is not
bipartite, then it's obviously impossible to find a solution.
Find a bicoloring of the graphs (paint each node in white or black
color, with the restriction that two adjacent nodes must be of different
color). Now, try to make the number of white/black nodes as close to
N/2 as possible. This can be done efficiently with Dynamic Programming.
*/
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

#define PB push_back

typedef vector<int> Vi;
typedef vector<Vi> Mi;

const int INF = 1000000000;

int N, C;
Mi net;
Vi comp, color;

int dp[110][110];
int que[110][110];

bool pinta(int n, int col, int com) {
  if (color[n] != -1) return color[n] == col;
  color[n] = col;
  comp[n] = com;
  for (int i = 0; i < int(net[n].size()); ++i)
    if (not pinta(net[n][i], 1 - col, com))
      return false;
  return true;
}

int main() {
  cin >> N;

  Mi tmp(N, Vi(N, 0));
  for (int i = 0; i < N; ++i) {
    int t;
    while (cin >> t and t > 0) {
      --t;
      tmp[i][t] = 1;
    }
  }

  net = Mi(N);
  for (int i = 0; i < N; ++i)
    for (int j = i + 1; j < N; ++j)
      if (tmp[i][j] == 0 or tmp[j][i] == 0) {
        net[i].PB(j);
        net[j].PB(i);
      }
```

```
comp = color = Vi(N, -1);
C = 0;
for (int i = 0; i < N; ++i) {
  if (comp[i] != -1) continue;
  if (not pinta(i, 0, C++)) {
    cout << "No solution" << endl;
    return 0;
  }
}

Vi zeros(C, 0), uns(C, 0);
for (int i = 0; i < N; ++i) {
  if (color[i] == 0) ++zeros[comp[i]];
  else ++uns[comp[i]];
}

dp[C][0] = 1;
for (int k = 1; k <= N; ++k) dp[C][k] = 0;

for (int c = C - 1; c >= 0; --c)
  for (int k = 0; k <= N; ++k) {
    if (zeros[c] <= k and dp[c + 1][k - zeros[c]]) {
      dp[c][k] = 1;
      que[c][k] = 0;
    }
    else if (uns[c] <= k and dp[c + 1][k - uns[c]]) {
      dp[c][k] = 1;
      que[c][k] = 1;
    }
  }

int mini = INF, quin = 0;
for (int i = 0; i <= N; ++i) {
  int t = abs(N - i - i);
  if (dp[0][i] and t < mini) {
    mini = t;
    quin = i;
  }
}

Vi res(C);
int k = quin;
for (int c = 0; c < C; ++c) {
  if (que[c][k] == 0) {
    res[c] = 0;
    k -= zeros[c];
  }
  else {
    res[c] = 1;
    k -= uns[c];
  }
}

Vi va, vb;
for (int i = 0; i < N; ++i) {
  if ((color[i] + res[comp[i]])%2 == 0) va.PB(i);
  else vb.PB(i);
}

int na = va.size(), nb = vb.size();
cout << na;
```

```
  for (int i = 0; i < na; ++i) cout << " " << va[i] + 1;
  cout << endl;
  cout << nb;
  for (int i = 0; i < nb; ++i) cout << " " << vb[i] + 1;
  cout << endl;
}
```

1183. Brackets Sequence

Time Limit: 1.0 second
Memory Limit: 16 MB

Let us define a regular brackets sequence in the following way:

1. Empty sequence is a regular sequence.
2. If S is a regular sequence, then (S) and [S] are both regular sequences.
3. If A and B are regular sequences, then AB is a regular sequence.

For example, all of the following sequences of characters are regular brackets sequences:

`()`, `[]`, `(())`, `([])`, `()[]`, `()[()]`

And all of the following character sequences are not:

`(`, `[`, `)`, `)(`, `([)]`, `([(]`

Some sequence of characters '(', ')', '[', and ']' is given. You are to find the shortest possible regular brackets sequence, that contains the given character sequence as a subsequence. Here, a string $a_1 a_2 ... a_n$ is called a subsequence of the string $b_1 b_2 ... b_m$, if there exist such indices $1 \le i_1 < i_2 < ... < i_n \le m$, that $a_j = b_{i_j}$ for all $1 \le j \le n$.

Input

The input contains at most 100 brackets (characters '(', ')', '[' and ']') that are situated on a single line without any other characters among them.

Output

Write a single line that contains some regular brackets sequence that has the minimal possible length and contains the given sequence as a subsequence.

Sample

input	output
`([(]`	`()[()]`

Problem Author: Andrew Stankevich
Problem Source: 2001-2002 ACM Northeastern European Regional Programming Contest

```
/*
Timus 1183. Brackets Sequence
This problem can be solved efficiently with Dynamic Programming.
In the code below, dp[i][j] (i<=j) is the minimum length a regular sequence
must have to contain S[i..j] as a subsequence.
With the table dp completely filled and some additional information
it's easy to recreate the required regular sequence.
*/
#include <iostream>
#include <string>
using namespace std;

const int INF = 1000000000;

string S;
int dp[110][110];
int que[110][110];  // -1: (, -2: [, un altre valor vol dir tallar per alla

void result(int e, int d) {
  if (e > d) return;
  if (que[e][d] == -1) {
    int a = (S[e] == '(' ? e + 1 : e);
    int b = (S[d] == ')' ? d - 1 : d);
    cout << "(";
    result(a, b);
    cout << ")";
    return;
  }
  if (que[e][d] == -2) {
    int a = (S[e] == '[' ? e + 1 : e);
    int b = (S[d] == ']' ? d - 1 : d);
    cout << "[";
    result(a, b);
    cout << "]";
    return;
  }
  result(e, que[e][d]);
  result(que[e][d] + 1, d);
}

int main() {
  cin >> S;

  int n = S.size();

  for (int i = 0; i < n; ++i) {
    dp[i][i] = 2;
    if (S[i] == '(' or S[i] == ')') que[i][i] = -1;
    else que[i][i] = -2;
  }

  for (int sz = 2; sz <= n; ++sz)
    for (int i = 0; i + sz <= n; ++i) {
      int j = i + sz - 1;

      int t1 = INF, t2 = INF;
      if (S[i] == '(' or S[j] == ')') {
        int a = (S[i] == '(' ? i + 1 : i);
        int b = (S[j] == ')' ? j - 1 : j);
        t1 = 2;
        if (a <= b) t1 += dp[a][b];
```

```
      }
      if (S[i] == '[' or S[j] == ']') {
        int a = (S[i] == '[' ? i + 1 : i);
        int b = (S[j] == ']' ? j - 1 : j);
        t2 = 2;
        if (a <= b) t2 += dp[a][b];
      }

      if (t1 <= t2) {
        dp[i][j] = t1;
        que[i][j] = -1;
      }
      else {
        dp[i][j] = t2;
        que[i][j] = -2;
      }

      for (int k = i; k + 1 <= j; ++k) {
        int t = dp[i][k] + dp[k + 1][j];
        if (t < dp[i][j]) {
          dp[i][j] = t;
          que[i][j] = k;
        }
      }
    }

  result(0, n - 1);
  cout << endl;
}
```

Timus 1184. Cable Master

1184. Cable Master

Time Limit: 1.0 second
Memory Limit: 16 MB

Inhabitants of the Wonderland have decided to hold a regional programming contest. The Judging Committee has volunteered and has promised to organize the most honest contest ever. It was decided to connect computers for the contestants using a “star” topology - i.e. connect them all to a single central hub. To organize a truly honest contest, the Head of the Judging Committee has decreed to place all contestants evenly around the hub on an equal distance from it.

To buy network cables, the Judging Committee has contacted a local network solutions provider with a request to sell for them a specified number of cables with equal lengths. The Judging Committee wants the cables to be as long as possible to sit contestants as far from each other as possible.

The Cable Master of the company was assigned to the task. He knows the length of each cable in the stock up to a centimeter, and he can cut them with a centimeter precision being told the length of the pieces he must cut. However, this time, the length is not known and the Cable Master is completely puzzled.

You are to help the Cable Master, by writing a program that will determine the maximal possible length of a cable piece that can be cut from the cables in the stock, to get the specified number of pieces.

Input

The first line contains two integers N and K, separated by a space. N ($1 \le N \le 10000$) is the number of cables in the stock, and K ($1 \le K \le 10000$) is the number of requested pieces. The first line is followed by N lines with one number per line, that specify the length of each cable in the stock in meters. All cables are at least 1 meter and at most 100 kilometers in length. All lengths are written with a centimeter precision, with exactly two digits after a decimal point.

Output

Write the maximal length (in meters) of the pieces that Cable Master may cut from the cables in the stock to get the requested number of pieces. The number must be written with a centimeter precision, with exactly two digits after a decimal point.

If it is not possible to cut the requested number of pieces each one being at least one centimeter long, then the output must contain the single number “0.00” (without quotes).

Sample

input	output
4 11 8.02 7.43 4.57 5.39	2.00

Problem Author: Vladimir Pinaev, Roman Elizarov
Problem Source: 2001-2002 ACM Northeastern European Regional Programming Contest

```
/*
Timus 1184. Cable Master
Fixed a length for the cables, we can easily calculate the maximum number
of cables we can get of that length by cutting the given cables.
Find the maximum length for the cables that produces as many
cables as the number of contestants. This can be done efficiently
with a binary search.
One tip: use only integers instead of floating-point variables.
*/
#include <iostream>
#include <vector>
#include <iomanip>
using namespace std;

typedef vector<int> Vi;

int main() {
  int n, k;
  cin >> n >> k;

  Vi v(n);
  for (int i = 0; i < n; ++i) {
    int a, b;
    char c;
    cin >> a >> c >> b;
    v[i] = 100*a + b;
  }

  int e = 1, d = 10000000, r = 0;
  while (e <= d) {
    int m = (e + d)/2;
    int t = 0;
    for (int i = 0; i < n; ++i) t += v[i]/m;
    if (t < k) d = m - 1;
    else {
      e = m + 1;
      r = m;
    }
  }

  cout << r/100 << "." << setw(2) << setfill('0') << r%100 << endl;
}
```

Timus 1186. Chemical Reactions

1186. Chemical Reactions

Time Limit: 1.0 second

Memory Limit: 16 MB

Bill teaches chemistry in the school and has prepared a number of tests for his students. Each test has a chemical formula and a number of possible reaction outcomes that his students are to choose one correct from. However, Bill wants to make sure that he has not made any typos while entering his tests into a computer and that his students won't easily throw away wrong answers simply by counting a number of chemical elements on the left and on the right side of the equation, which should be always equal in a valid reaction.

You are to write a program that will help Bill. The program shall read the description of the test for the students that consists of the given left side of the equation and a number of possible right sides, and determines if the number of chemical elements on each right side of the equation is equal to the number of chemical elements on the given left side of the equation.

To help you, poor computer folks, that are unaware of the complex world of chemistry, Bill has formalized your task. Each side of the equation is represented by a string of characters without spaces, and consists of one or more chemical sequences separated by a '+' (plus) characters. Each sequence has an optional preceding integer multiplier that applies to the whole sequence and a number of elements. Each element is optionally followed by an integer multiplier that applies to it. An element in this equation can be either distinct chemical element or a whole sequence that is placed in round parenthesis. Every distinct chemical element is represented by either one capital letter or a capital letter that is followed by a small letter.

Even more formally, using notation that is similar to BNF, we can write:

- <formula> ::= [<number>] <sequence> { '+' [<number>] <sequence> }
- <sequence> ::= <element> [<number>] { <element> [<number>] }
- <element> ::= <chem> | '(' <sequence> ')'
- <chem> ::= <uppercase_letter> [<lowercase_letter>]
- <uppercase_letter> ::= 'A'..'Z'
- <lowercase_letter> ::= 'a'..'z'
- <number> ::= '1'..'9' { '0'..'9' }

Every distinct chemical element is said to occur in the given formula for some total number X, if X is the sum of all separate occurrences of this chemical element multiplied to all numbers that apply to it. For example, in the following chemical formula:

```
C2H5OH+3O2+3(SiO2)
```

- `C` occurs for a total of 2 times.
- `H` occurs for a total of 6 times (5 + 1).
- `O` occurs for a total of 13 times (1 + 3*2 + 3*2).
- `Si` occurs for a total of 3 times.

All multipliers in the formula are integer numbers that are at least 2 if explicitly specified and are 1 by default. Each chemical formula is at most 100 characters long, and every distinct chemical element is guaranteed to occur for a total of no more than 10000 times in each formula.

Input

The first line represents a chemical formula that is to be tested as the left side of the equation. The second line contains a single integer number N ($1 \le N \le 10$), which is the number of right sides of the equation that are to be tested. Each one of the following N lines represents one such formula.

Output

You are to write N lines — one line per each possible answer of the chemical test for Bill's students that is given in the input. For each right-hand side formula that is encountered in the input, you should output:

```
<left_formula>==<right_formula>
```

if the total number of occurrences of each distinct chemical element on the left-hand side equals to the total number of occurrences of this chemical element on the right-hand side. Otherwise write:

```
<left_formula>!=<right_formula>
```

Here `<left_formula>` must be replaced exactly (character by character) with the original left-hand side formula as it is given in the first line of the input, and `<right_formula>` must be replaced exactly with each right-hand side formula as they are given in the input. Do not place any spaces in the lines you write.

Sample

input	output

input	output
C2H5OH+3O2+3(SiO2) 7 2CO2+3H2O+3SiO2 2C+6H+13O+3Si 99C2H5OH+3SiO2 3SiO4+C2H5OH C2H5OH+3O2+3(SiO2)+Ge 3(Si(O)2)+2CO+3H2O+O2 2CO+3H2O+3O2+3Si	C2H5OH+3O2+3(SiO2)==2CO2+3H2O+3SiO2 C2H5OH+3O2+3(SiO2)==2C+6H+13O+3Si C2H5OH+3O2+3(SiO2)!=99C2H5OH+3SiO2 C2H5OH+3O2+3(SiO2)==3SiO4+C2H5OH C2H5OH+3O2+3(SiO2)!=C2H5OH+3O2+3(SiO2)+Ge C2H5OH+3O2+3(SiO2)==3(Si(O)2)+2CO+3H2O+O2 C2H5OH+3O2+3(SiO2)!=2CO+3H2O+3O2+3Si

Problem Author: Joseph Romanosky, Roman Elizarov

Problem Source: 2001-2002 ACM Northeastern European Regional Programming Contest

```
/*
Timus 1186. Chemical Reactions
A simple way to solve this problem is to count the number of occurrences of
each distinct chemical element in each of the two given formules. The way to
count element occurrences is by parsing strings using the grammar described in
the problem statement: a formula contains sequences, a sequence contains
elements, elements contain either a chem or a sequence between parenthesis, etc.
*/
#include <iostream>
#include <map>
#include <string>

using namespace std;

bool is_digit(char c) {
  return c >= '0' && c <= '9';
}

bool is_alpha(char c) {
  return (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z');
}

bool is_upper(char c) {
  return c >= 'A' && c <= 'Z';
}

int read_int(string& s, int& a) {
  int res = 0;
  for (; a < int(s.size()) && is_digit(s[a]); ++a) {
    res = 10*res + int(s[a]-'0');
  }
  return res;
}

string read_chem(string& s, int& a) {
  string res = "";
  if (a < int(s.size()) && is_upper(s[a])) {
    res += s[a];
    ++a;
    if (a < int(s.size()) && is_alpha(s[a]) && !is_upper(s[a])) {
      res += s[a];
      ++a;
    }
  }
  return res;
}

map<string, int> process_sequence(string& s, int a, int b) {
  map<string, int> cnt_seq;
  if (b-a < 1) return cnt_seq;
  int end;
  if (s[a] == '(') {
    int depth = 1;
    end = a+1;
    for (; end < b && depth > 0; ++end) {
      if (s[end] == '(') ++depth;
      else if (s[end] == ')') --depth;
    }
    map<string, int> cnt_seq_rec = process_sequence(s, a+1, end-1);
    int mult = 1;
    if (is_digit(s[end])) {
```

```
      mult = read_int(s, end);
    }
    for (map<string, int>::iterator it = cnt_seq_rec.begin(); it != cnt_seq_rec.end
        (); ++it) {
      cnt_seq[it->first] = mult*(it->second);
    }
  }
  else {
    end = a;
    string chem = read_chem(s, end);
    int mult = 1;
    if (end < b && is_digit(s[end])) {
      mult = read_int(s, end);
    }
    cnt_seq[chem] += mult;
  }
  map<string, int> cnt_seq_rec = process_sequence(s, end, b);
  for (map<string, int>::iterator it = cnt_seq_rec.begin(); it != cnt_seq_rec.end();
       ++it) {
    cnt_seq[it->first] += it->second;
  }
  return cnt_seq;
}

map<string, int> process_formula(string& s, int a, int b) {
  map<string, int> cnt_for;
  if (b-a < 1) return cnt_for;
  int ini = a;
  for (int fin = a; fin <= b; ++fin) if (fin == b || s[fin] == '+') {
    int mult = 1;
    if (is_digit(s[ini])) {
      mult = read_int(s, ini);
    }
    map<string, int> cnt_seq = process_sequence(s, ini, fin);
    for (map<string, int>::iterator it = cnt_seq.begin(); it != cnt_seq.end(); ++it)
         {
      cnt_for[it->first] += mult*(it->second);
    }
    ini = fin+1;
  }
  return cnt_for;
}

int main() {
  string left;
  cin >> left;
  map<string, int> cnt_left = process_formula(left, 0, int(left.size()));
  int n;
  cin >> n;
  for (int i = 0; i < n; ++i) {
    string right;
    cin >> right;
    map<string, int> cnt_right = process_formula(right, 0, int(right.size()));
    if (cnt_left == cnt_right) {
      cout << left << "==" << right << endl;
    }
    else cout << left << "!=" << right << endl;
  }
}
```

Timus 1249. Ancient Necropolis

1249. Ancient Necropolis

Time Limit: 5.0 second

Memory Limit: 4 MB

Aerophotography data provide a bitmap picture of a hard-to-reach region. According to the suggestions of scientists, this region is a cemetery of an extinct civilization. Indeed, the picture, having been converted to a binary form, shows distinctly visible areas, dark (marked with symbols 1) and light (marked with 0). It seems that the dark areas are tombstones. It's easy to either confirm or reject the hypothesis since the race that lived in the region knew astronomy, so tombstones were always oriented along the Earth's parallels and meridians. That is why the dark areas in the picture should have the form of rectangles with the sides parallel to the axes. If it is so, then we indeed have a picture of a cemetery of an extinct race. Otherwise, new hypotheses should be suggested.

Input

The first input line contains two integers N and M, which are the dimensions of the picture provided by the aerophotography. Each of the next N lines contains M zeros or ones separated with a space. The numbers N and M do not exceed 3000.

Output

Output "Yes" if all connected dark areas in the picture are rectangles and "No" otherwise.

Samples

input	output
2 2 0 1 1 1	No
3 3 0 0 1 1 1 0 1 1 0	Yes

Problem Author: Nikita Shamgunov and Leonid Volkov

Problem Source: Open collegiate programming contest for student teams, Ural State University, March 15, 2003

```
/*
Timus 1249. Ancient Necropolis
Check for each connected component of 0's or 1's whether it is a rectangle.
To check that, find the minimum/maximum X/Y reachable and look if it's
is consistent with the number of cells in that component.
The provided implementation is based on a different idea, it is a kind of
Sweep Line algorithm.
*/
#include <iostream>
#include <string>
#include <utility>
#include <vector>
using namespace std;

#define PB push_back
#define X first
#define Y second

typedef vector<string> Vs;
typedef vector<int> Vi;
typedef pair<int, int> P;
typedef vector<P> Vp;

const int diri[4] = { -1, 0, 1, 0 };
const int dirj[4] = { 0, 1, 0, -1 };

int R, C;

int main() {
  cin >> R >> C;
  Vp last;
  for (int k = 0; k < R; ++k) {
    string fila(C, '.');
    for (int i = 0; i < C; ++i) cin >> fila[i];

    Vp current;
    int i = 0;
    while (i < C) {
      int j = 1;
      while (i + j < C and fila[i] == fila[i + j]) ++j;
      if (fila[i] == '1') current.PB(P(i, i + j - 1));
      i += j;
    }

    int na = last.size(), nb = current.size();
    int b = 0;
    for (int a = 0; a < na; ++a) {
      while (b < nb and current[b].Y < last[a].X) ++b;
      if (b < nb and current[b].X <= last[a].Y and current[b] != last[a]) {
        cout << "No" << endl;
        return 0;
      }
    }

    last = current;
  }
  cout << "Yes" << endl;
}
```

Timus 1250. Sea Burial

1250. Sea Burial

Time Limit: 1.0 second
Memory Limit: 16 MB

There is Archipelago in the middle of a shoreless ocean. An ancient tribe of cannibals lives there. Shamans of this race have been communicating with gods and admonishing people for ages. They could generate a rain during a drought and clear the sky in a raining season. A long time ago the first shaman of the tribe jumped into one of the seas and drowned while being in a sacred trance. Since then all the land inside this sea is regarded as sacred. According to an ancient law, all shamans must be buried on a sacred land. However, souls of dead shamans cannot get on with each other, so each shaman must be buried on a separate island. An old prophecy says that if two shamans are buried on the same land, then a dreadful time will come and the tribe will perish.

How many shamans will the tribe outlive? This problem bothered all the chiefs of the tribe who were coming into power. So one of the chiefs ordered to compile a map of Archipelago. The cannibals toiled for a whole year and coped with the task. But the map was too large and complicated to count all the sacred islands. So the tribe's shaman appealed to gods and asked them to help with counting the islands. And the tribe was blessed with a Programmer and a Computer, which came down to earth in a cloud of light and fire. Yes, you are this Programmer, and you are destined to live with these cannibals until you have counted the islands; then you'll be transferred back home. Remember that there may be seas inside islands, and islands inside those seas.

Input

The first input line contains four integers W, H, X and Y, separated with one or several spaces. $1 \le W, H \le 500$ are respectively the width and the height of the map. $1 \le X \le W$ and $1 \le Y \le H$ are the coordinates of the point where the first shaman drowned. The next H lines contain description of the map. Each line contains W symbols describing correspondent fragments of the map. Symbol "." stands for a sea fragment and symbol "#" stands for a land fragment. Two fragments belong to the same sea if they are adjacent horizontally, vertically, or diagonally. Two fragments belong to the same island if they are adjacent horizontally or vertically (but not diagonally). Land fragments that are adjacent to the map's border are not considered as islands. Coordinates are counted from the left upper corner.

Output

The output should contain a single integer, which is the number of the islands inside the sacred sea.

Samples

<table>
<tr><th>input</th><th>output</th></tr>
<tr><td><pre>9 7 1 1
........#
.#######.
.#.....#.
.#.#.#.#.
.#.....#.
.#######.
#........</pre></td><td><pre>3</pre></td></tr>
<tr><td><pre>9 7 3 3
........#
.#######.
.#.....#.
.#.#.#.#.
.#.....#.
.#######.
#........</pre></td><td><pre>2</pre></td></tr>
</table>

Problem Author: Stanislav Skorb (prepared by Ivan Dashkevich)
Problem Source: Open collegiate programming contest for student teams, Ural State University, March 15, 2003

```
/*
Timus 1250. Sea Burial
First, mark all the cells of the sacred sea with a BFS from the cell where the
shaman was drowned. Second, mark all the cells reachable from the outside of
the map, without trespassing any previously marked cell (sacred sea). Finally,
count the number of connected components of non-marked cells.
*/
#include <iostream>
#include <queue>
#include <string>
#include <vector>
using namespace std;

#define X first
#define Y second

typedef vector<string> Vs;
typedef pair<int, int> P;
typedef queue<P> Q;
typedef vector<bool> Vb;
typedef vector<Vb> Mb;

const int diri[8] = { -1, 0, 1, 0, -1, 1, 1, -1 };
const int dirj[8] = { 0, 1, 0, -1, 1, 1, -1, -1 };

int R, C;
Vs mapa;
Mb vist;

int main() {
  int sr, sc;
  cin >> C >> R >> sc >> sr;
  --sr; --sc;

  mapa = Vs(R);
  for (int i = 0; i < R; ++i) cin >> mapa[i];

  vist = Mb(R, Vb(C, false));
  vist[sr][sc] = true;

  Q q;
  q.push(P(sr, sc));

  while (not q.empty()) {
    int i = q.front().X, j = q.front().Y;
    q.pop();
    for (int k = 0; k < 8; ++k) {
      int x = i + diri[k], y = j + dirj[k];
      if (0 <= x and x < R and 0 <= y and y < C and not vist[x][y] and mapa[x][y] ==
           '.') {
        vist[x][y] = true;
        q.push(P(x, y));
      }
    }
  }

  for (int i = 0; i < R; ++i)
    if (not vist[i][0]) {
      vist[i][0] = true;
      q.push(P(i, 0));
    }
```

```
  for (int i = 0; i < R; ++i)
    if (not vist[i][C - 1]) {
      vist[i][C - 1] = true;
      q.push(P(i, C - 1));
    }
  for (int i = 0; i < C; ++i)
    if (not vist[0][i]) {
      vist[0][i] = true;
      q.push(P(0, i));
    }
  for (int i = 0; i < C; ++i)
    if (not vist[R - 1][i]) {
      vist[R - 1][i] = true;
      q.push(P(R - 1, i));
    }

  while (not q.empty()) {
    int i = q.front().X, j = q.front().Y;
    q.pop();
    for (int k = 0; k < 4; ++k) {
      int x = i + diri[k], y = j + dirj[k];
      if (0 <= x and x < R and 0 <= y and y < C and not vist[x][y]) {
        vist[x][y] = true;
        q.push(P(x, y));
      }
    }
  }

  int res = 0;

  for (int tx = 0; tx < R; ++tx)
    for (int ty = 0; ty < C; ++ty) {
      if (not vist[tx][ty] and mapa[tx][ty] == '#') {
        vist[tx][ty] = true;
        q.push(P(tx, ty));

        while (not q.empty()) {
          int i = q.front().X, j = q.front().Y;
          q.pop();
          for (int k = 0; k < 4; ++k) {
            int x = i + diri[k], y = j + dirj[k];
            if (0 <= x and x < R and 0 <= y and y < C and not vist[x][y] and mapa[x
                ][y] == '#') {
              vist[x][y] = true;
              q.push(P(x, y));
            }
          }
        }

        ++res;
      }
    }

  cout << res << endl;
}
```

1252. Sorting the Tombstones

Time Limit: 1.0 second
Memory Limit: 16 MB

There is time to throw stones and there is time to sort stones…

An old desolate cemetery is a long dismal row of nameless tombstones There are N tombstones of various shapes. The weights of all the stones are different. People have decided to make the cemetery look more presentable, sorting the tombstone according to their weight. The local custom allows to transpose stones if there are exactly K other stones between them.

Input

The first input line contains an integer N ($1 \le N \le 130000$). Each of the next N lines contains an integer X, the weight of a stone in grams ($1 \le X \le 130000$).

Output

The output should contain the single integer — the maximal value of K ($0 \le K < N$), that makes possible the sorting of the stones according to their weights.

Sample

input	output
5 30 21 56 40 17	1

Problem Author: Alexey Lakhtin
Problem Source: Open collegiate programming contest for student teams, Ural State University, March 15, 2003

```
/*
Timus 1252. Sorting the Tombstones
The number we are looking for is the greatest common divisor of all the distances
between the position of an integer in the original array and the position of the
same integer in the sorted array, minus one.
Tricky thing of the statement: When the statement says "sorting the tombstone
according to their weight", it means sorting either increasingly or decreasingly.
*/
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

#define X first
#define Y second

typedef pair<int, int> P;
typedef vector<P> Vp;

int gcd(int a, int b) {
  if (b == 0) return a;
  return gcd(b, a%b);
}

bool sortf(P a, P b) {
  return a.X > b.X;
}

int main() {
  int n;
  cin >> n;

  Vp v(n);
  for (int i = 0; i < n; ++i) {
    cin >> v[i].X;
    v[i].Y = i;
  }

  sort(v.begin(), v.end());
  int g = 0;
  for (int i = 0; i < n; ++i) g = gcd(g, abs(i - v[i].Y));
  int r1 = n - 1;
  if (g != 0) r1 = g - 1;

  sort(v.begin(), v.end(), sortf);
  g = 0;
  for (int i = 0; i < n; ++i) g = gcd(g, abs(i - v[i].Y));
  int r2 = n - 1;
  if (g != 0) r2 = g - 1;

  cout << max(r1, r2) << endl;
}
```

Timus 1253. Necrologues

1253. Necrologues

Time Limit: 1.0 second
Memory Limit: 16 MB

You know that many necrologues even the most heartfelt are very similar. Our partners from the ACM Company (Advanced Cemetery Management), which is a sponsor of today's thematic problem set, decided to disclose some secrets of skill. As soon as we got into the workshop of the word-painters we found out that:

1. there are N ($1 \le N \le 9$) sample necrologues;
2. each sample necrologue has not more than 1000 symbols (capital and small Latin letters, digits, punctuation marks, spaces and carriage return marks);
3. each sample may contain not more than 10 references to other samples (the reference is marked by *, which is followed by a number of a sample that is referred to; the *-mark is used in the samples in no other way but as a reference mark).

A reference may be *activated* at a client's pleasure: the sequence *M is substituted with the sample necrologue number M (with all its references).

This system worked properly up to the moment that a very rich client entered the workshop. He wanted to add a stone to his mother's-in-law cairn and wished a necrologue according to the sample number 1 with *all* the references activated (including the ones that can appear after activating the references in the first sample, and so on).

It's necessary to write a program in order to understand if it's possible to fulfill the wishes of the client and what will appear in this necrologue.

Input

The first input line contains a number N, an amount of the necrologues samples. Than follow the samples texts. Each samples starts from a new line and ends with the # symbol.

Output

Consider the necrologue forming procedure that starts from the first sample, activates all the references than activates all the references in the result of the previous step and so on ad infinitum. If such a procedure leads to the text not longer than 10^6 symbols (spaces and line feeds are considered as symbols) you should output the necrologue text. You should output # otherwise.

Sample

input	output
7 She w*7s *7 *2 wom*7n. *3# wonderful# Everyone loved her *5. We will miss her *5.# Some text *6# very much# Another text *4# a#	She was a wonderful woman. Everyone loved her very much. We will miss her very much.

Problem Author: Leonid Volkov
Problem Source: Open collegiate programming contest for student teams, Ural State University, March 15, 2003

```
/*
Timus 1253. Necrologues
The input is a grammar which describes a text. The output must be
this text. The grammar may have cycles, in this case you must
detect it and write # to the output. If the grammar doesn't lead
to any cycle, but the resulting text is more than 10^6 characters
long, you must print # too.
The algorithm below performs a DFS to detect cycles and recover
the text if possible.
*/
#include <iostream>
#include <string>
#include <vector>
using namespace std;

typedef vector<string> Vs;
typedef vector<int> Vi;

Vi dins;
Vs cosa;
string res;

bool fun(int n) {
  if (dins[n]) return false;
  dins[n] = 1;

  int s = cosa[n].size(), p = 0;
  while (p < s) {
    int q = 0;
    while (p + q < s and cosa[n][p + q] != '*') ++q;
    res += cosa[n].substr(p, q);
    if (res.size() > 1000000) return false;
    p += q;

    if (p < s) {
      ++p;  // *
      int t = cosa[n][p++] - '1';
      if (not fun(t)) return false;
    }
  }

  dins[n] = 0;
  return true;
}

int main() {
  int n;
  cin >> n;

  cosa = Vs(n);

  string line;
  getline(cin, line);

  int m = 0;
  while (m < n) {
    while (true) {
      getline(cin, line);
      if (line.size() and line[line.size() - 1] == '#') {
        cosa[m] += line.substr(0, line.size() - 1);
        ++m;
```

```
        break;
      }
      else {
        cosa[m] += line + "\n";
      }
    }
  }

  dins = Vi(n, 0);
  if (fun(0)) cout << res << endl;
  else cout << "#" << endl;
}
```

1255. Graveyard of the Cosa Nostra

Time Limit: 1.0 second
Memory Limit: 16 MB

There is a custom among the Ural Mafiosi — a big Mafioso's coffin is to be carried by all his subordinates. The length of the coffin (in meters) equals to the number of the Mafioso's subordinates in order not to let the carriers to encumber each other. As it happens, according to the ancient custom the width of a coffin is equal to 1 meter. So, the length of a coffin shows a dead man's authority. By the way, the Ural Mafiosi are very scrupulous in matters of authority and will not bear neighborhood with less authoritative Mafioso. So, at one cemetery it's possible to bury Mafiosi with equal authority. According to the Mafiosi's custom a cemetery must be square. A cemetery length must be an integer number of meters.

You are to count how many Mafiosi can be buried on the cemetery of the given size. Coffins must be parallel to cemetery borders, coffins mustn't overlap each other and get off the cemetery.

Input

Contains two numbers — a length of the cemetery N ($1 < N < 10000$) and a length of a coffin K ($1 < K < 10000$).

Output

The single integer number — the most amount of the coffins of the size $1 \times K$ that may be buried at the cemetery of the size $N \times N$.

Sample

input	output
5 4	6

Problem Author: Stanislav Vasilyev, Alexey Lakhtin
Problem Source: Open collegiate programming contest for student teams, Ural State University, March 15, 2003

```
/*
Timus 1255. Graveyard of the Cosa Nostra
This problem can be solved with Dynamic Programming based on an greedy
method to distribute the coffins optimally.
V[n] is the number of coffins of length k that can fit in a square of side n.
One optimal way to distribute the coffins in the square is always one of the
following:
(1) Fill each row with as many adjacent horizontal coffins (maybe leaving
empty cells after the last coffin of the row), and then fill the rightmost
empty columns with vertical coffins.
(2) If k > n/2, put n-k horizontal coffins in the first n-k rows, occupying
the first k columns, put also n-k horizontal coffins in the last n-k rows,
occupying the last k columns, and fill as much as possible empty cells
with vertical coffins.
(3) Put n vertical coffins occupying the first k rows, put n-k horizontal
coffins in the rest of the rows occupying the first k rows, and finally
fill the empty space (a square of side n-k) optimally (recursively).
*/
#include <iostream>
using namespace std;

int V[11000];

int main() {
  int m, k;
  cin >> m >> k;

  for (int n = 0; n <= m; ++n) {
    if (k <= n) {
      int r = n*(n/k) + (n/k)*(n%k);
      if (2*k > n) r = max(r, 4*(n - k));
      V[n] = r;

      V[n] = max(V[n], n + n - k + V[n - k]);
    }
    else V[n] = 0;
  }

  cout << V[m] << endl;
}
```

1275. Knights of the Round Table

Time Limit: 1.0 second
Memory Limit: 0.5 MB

N knights gathered at the King Arthur's round table. Each of them has several goblets near him. It is possible that knights have different number of goblets. The goblets are brought (and also carried away) by a servant who can carry only two goblets at a time (one for each hand). When the servant comes he can either bring two goblets, or carry them away. Note that he can serve exactly two knights that sit at a fixed distance K from each other.

For example, if K=1 then the knights who sit side by side are served.

By the end of the feast each of the knights should have an equal predefined number of goblets near him. The number of the times the servant has to come must be minimized.

Your task is to write a program, which plans the servant's work during the feast.

Limitations
2 <= N <= 1000
1 <= K <= N-1
Initial and final number of goblets near each knight is not greater than 1000 and it is always non-negative.
The total number of servant's visits is not greater than 30000.

Input

The first number contains three numbers separated with white-space.
N – the number of knights,
K – "arm-span" of the servant,
F – the final number of goblets each of the knights must have by the end of the feast.
The following N numbers separated with spaces or EOL characters describe the initial number of goblets near each knight. It is assumed that the knights are numbered in a cyclic manner, i.e. the first knight sits after the N-th one.

Output

If it is impossible to reach the goal, write "–1" (without quotes) to the output. If the solution exists then the first line must contain a single integer M – the number of the servant's visits. The following M lines must carry triples: two numbers (the numbers of knights being served) and "+" (plus) character if the goblets are brought or "–" (minus) if they are carried away. The data on each of these lines must be separated with a white-space character.

Sample

input	output
3 1 4 1 2 3	3 1 2 + 1 2 + 3 1 +

Problem Author: © Sergey G. Volchenkov, 2003(volchenkov@yandex.ru); Vladimir N. Pinaev, 2003(vpinaev@mail.ru; http://www.pic200x.chat.ru); Michael Y. Kopachev, 2003 (mkopachev@krista.ru).
Problem Source: 2003-2004 ACM Central Region of Russia Quarterfinal Programming Contest, Rybinsk, October 15-16, 2003

```
/*
Timus 1275. Knights of the Round Table
A key observation here is that the problem can be reduced to the case K=1.
Note that there exists a partition of the set of knights into independent
subsets such that the servant can never carry goblets to knights of different
subsets, and whenever he carries goblets, those are carried to knights of the
same subset. This partition can be found in linear time.
Now, run the algorithm for the case K=1 with each subset as input. The
details of this algorithm can be seen in the function minim() below.
*/
#include <algorithm>
#include <iostream>
#include <utility>
#include <vector>
using namespace std;

#define PB push_back
#define X first
#define Y second

typedef long long ll;
typedef pair<int, int> P;
typedef pair<int, P> PP;
typedef vector<P> Vp;
typedef vector<Vp> Mp;
typedef vector<PP> Vpp;
typedef vector<int> Vi;

const int INF = 1000000000;
const int LIM = 30010;

int minim(Vp v, int f, Vpp& res) {
  int n = v.size();

  if (n == 0) return 0;
  if (n == 1) {
    if (v[0].X == f) return 0;
    return INF;
  }

  int mini = INF, quin = -1;
  for (int d = -LIM; d <= LIM; ++d) {
    int tmp = abs(d);
    int u = d;
    for (int i = 1; i < n; ++i) {
      u = f - (v[i].X + u);
      tmp += abs(u);
    }

    if (v[0].X + u + d == f) {
      if (tmp < mini) {
        mini = tmp;
        quin = d;
      }
    }
  }
  if (mini == INF) return INF;

  res.clear();
  int u = quin;
  res.PB(PP(u, P(v[0].Y, v[1].Y)));
```

```
  for (int i = 1; i < n; ++i) {
    u = f - (v[i].X + u);
    res.PB(PP(u, P(v[i].Y, v[(i + 1)%n].Y)));
  }

  return mini;
}

int main() {
  int n, k, f;
  cin >> n >> k >> f;

  Vi init(n);
  for (int i = 0; i < n; ++i) cin >> init[i];

  Mp cicle;
  Vi vist(n, 0);
  for (int i = 0; i < n; ++i) {
    if (vist[i]) continue;
    Vp tmp;
    int j = i;
    while (vist[j] == 0) {
      vist[j] = 1;
      tmp.PB(P(init[j], j));
      j = (j + k)%n;
    }
    cicle.PB(tmp);
  }

  int cost = 0;
  Vpp res;
  for (int i = 0; i < int(cicle.size()); ++i) {
    Vpp tmp;
    cost += minim(cicle[i], f, tmp);
    if (cost > LIM) break;
    for (int j = 0; j < int(tmp.size()); ++j) res.PB(tmp[j]);
  }

  if (cost > 30000) {
    cout << -1 << endl;
    return 0;
  }

  cout << cost << endl;
  sort(res.begin(), res.end());

  int sz = res.size();
  for (int i = 0; i < sz; ++i) {
    if (res[sz - i - 1].X == 0) continue;
    int num = res[sz - i - 1].X;
    char sig = (num < 0 ? '-' : '+');
    int a = res[sz - i - 1].Y.X + 1;
    int b = res[sz - i - 1].Y.Y + 1;

    for (int j = 0; j < abs(num); ++j) {
      cout << a << " " << b << " " << sig << endl;
    }
  }
}
```

Timus 1276. Train

1276. Train

Time Limit: 1.0 second
Memory Limit: 4 MB

Train Ltd., a company that is in for railroad transportation received an order to form a train having a certain number of carriages. The problem is that Train Ltd. has carriages built in different years, therefore each of the carriages may have one of the two kinds of coupling at each side. The company also has one locomotive at its disposal.

The coupling systems for both the locomotive and the carriages are labeled as “A” or “B”. It is impossible to turn either the locomotive or a carriage in the opposite direction.

You are given information about the carriages and the locomotive. Your task is to determine the number of ways to form different trains using the given carriages. The additional requirement is that the coupling systems at each of the ends of the train must correspond to those of the locomotive.

The trains are considered different if there is at least one mismatch when they are compared from one end to another.

Example 1. Let the company possess the following carriages: “AA”, “AA”, “AB”, “BA”, “BA” and the locomotive “AB”. The train must have four carriages. Then it is possible to form only two different trains having these carriages: “BAAAABBA” and “BAABBAAA”. It is possible to connect the locomotive at the left end of this train (using coupling “B”) or at the right one (using coupling “A”).

Example 2. Let the company now have only one carriage of each type: “AA”, “AB”, “BA”, “BB”, and the locomotive is “AA”. The train must have three carriages now. There are three ways to form a train: “AAABBA”, “ABBAAA” and “ABBBBA”.

Input

The first line of the input contains two integers separated with white-space character: N $(0 < N \le 40)$ — the number of carriages the company has at its disposal, and K $(0 < K \le N)$ — the required length of the train (measured in carriages). The following $N + 1$ lines describe the coupling systems for the locomotive (line 2) and the carriages. These descriptions are given as “AB”, “AA”, “BB” or “BA” (without quotes).

Output

Write the word “YES” if it is possible to form one (or more) trains according to the given parameters, or “NO” otherwise. If it is possible to form a train then write the number of different ways of doing so to the second line.

Samples

input	output
4 4 AB AA AB BA BA	YES 2
4 4 BA AA AB BA BA	NO

Problem Author: © Sergey G. Volchenkov, 2003(volchenkov@yandex.ru); Vladimir N. Pinaev, 2003(vpinaev@mail.ru; http://www.pic200x.chat.ru); Michael Y. Kopachev, 2003 (mkopachev@krista.ru).
Problem Source: 2003-2004 ACM Central Region of Russia Quarterfinal Programming Contest, Rybinsk, October 15-16, 2003

```
/*
Timus 1276. Train
This problem can be solved using Dynamic Programming. We define a state
(aa, ab, ba, bb, left, right, k) as the number of trains of length k having a
coupling of type "left" at its left end and a coupling of type "right" at its
right end using at most aa carriages of type "AA", ab carriages of type "AB",
ba carriages of type "BA" and bb carriages of type "BB". The value of a state
can be computed from the value of states with smaller k by splitting the trains
into groups depending on the type of the first carriage.
*/
#include <iostream>
#include <string>
#include <vector>

using namespace std;

typedef long long ll;
typedef vector<ll> VL;
typedef vector<VL> V2L;
typedef vector<V2L> V3L;
typedef vector<V3L> V4L;
typedef vector<V4L> V5L;

V5L mem_trains;

ll trains(int aa, int ab, int ba, int bb, int left, int right, int rem) {
  if (rem == 0) {
    if (left == right) return 1;
    return 0;
  }
  ll& ret = mem_trains[aa][ab][ba][bb][left];
  if (ret != -1) return ret;
  ret = 0;
  if (aa > 0) {
    if (left == 0) ret += trains(aa-1, ab, ba, bb, 0, right, rem-1);
  }
  if (ab > 0) {
    if (left == 0) ret += trains(aa, ab-1, ba, bb, 1, right, rem-1);
  }
  if (ba > 0) {
    if (left == 1) ret += trains(aa, ab, ba-1, bb, 0, right, rem-1);
  }
  if (bb > 0) {
    if (left == 1) ret += trains(aa, ab, ba, bb-1, 1, right, rem-1);
  }
  return ret;
}

int main() {
  int n, k;
  cin >> n >> k;
  string locomotive;
  cin >> locomotive;
  int left, right;
  if (locomotive == "AA") {
    left = 0;
    right = 0;
  }
  else if (locomotive == "AB") {
    left = 1;
    right = 0;
```

```
  }
  else if (locomotive == "BA") {
    left = 0;
    right = 1;
  }
  else {
    left = 1;
    right = 1;
  }
  int aa = 0, ab = 0, ba = 0, bb = 0;
  for (int i = 0; i < n; ++i) {
    string carriage;
    cin >> carriage;
    if (carriage == "AA") ++aa;
    else if (carriage == "AB") ++ab;
    else if (carriage == "BA") ++ba;
    else ++bb;
  }
  mem_trains = V5L(aa+1, V4L(ab+1, V3L(ba+1, V2L(bb+1, VL(2, -1)))));
  ll ans = trains(aa, ab, ba, bb, left, right, k);
  if (ans > 0) cout << "YES" << endl << ans << endl;
  else cout << "NO" << endl;
}
```

1278. "… Connecting People"

Time Limit: 1.0 second
Memory Limit: 16 MB

A new model of a mobile phone is powered by a special sound generation processor, developed expressly for this purpose.

The processor has a sound generator, an instruction pointer (IP), command memory (100 cells) and a stack (100 cells).

The instruction set of this processor consists of the two commands only:

CALL X – a subroutine call command – pushes the incremented current value of the IP (the return address) on the top of the stack and sets the IP to X;

BELL&RET – the combined command "bell-return" – emits a sound of a fixed (unit) duration and performs control return (pops a value from the top of the stack and assigns it to the IP) or stops the execution of the program (if the stack is empty).

Any command (together with its operand) occupies exactly one memory cell.

The return address occupies only one cell on the stack, too.

The work cycle of the processor starts when it is necessary to emit a sound. At this moment the IP points to the zero memory cell. The processor stops its work if the stack is empty after emitting a sound (when BELL&RET command has been executed).

The manufacturers of the processor affirm that by using of this processor may be emitted sound of rather long duration.

Your task is to write a program, which accepts K as input and produces a program for this processor to emit a sound of duration K.

Limitations
The resulting program may have no more than 100 commands including exactly one BELL&RET command, being always the last one. Unused memory cells following the BELL&RET command are considered free.

Input

There will be one integer number K in the input ($0 < K \le 10^9$) – sound duration.

Output

Output a program for the processor described above. The execution of this program leads to emission of the sound of duration K. The first line should correspond to the zero cell of the memory, the second line – to the first cell, the third – to the second one etc. All lines, with exception for the last one, may contain only CALL commands. The operand of the CALL command is an integer (from 1 to 99) and must be separated from the command by a white-space character.

The last line must contain the BELL&RET command.

Samples

input	output
1	BELL&RET
4	CALL 3 CALL 3 CALL 3 BELL&RET

Problem Author: © Sergey G. Volchenkov, 2003(volchenkov@yandex.ru); Vladimir N. Pinaev, 2003(vpinaev@mail.ru; http://www.pic200x.chat.ru); Michael Y. Kopachev, 2003 (mkopachev@krista.ru).
Problem Source: 2003-2004 ACM Central Region of Russia Quarterfinal Programming Contest, Rybinsk, October 15-16, 2003

```
/*
Timus 1278. "... Connecting People"
Let 'm' be the largest number such that 2^m<=n. It is easy to construct a sequence
of 'm' call instructions with a final bell&ret instruction that makes exactly 2^m
rings. Put this sequence at the end of the program, and before that sequence, put
as many call instructions as necessary, pointing to the proper instruction of the
final sequence to make exactly 2^x rings, for the minimum amount of powers of 2
that add exactly to n-2^m.
*/
#include <iostream>
using namespace std;

typedef long long ll;

int main() {
  ll n;
  cin >> n;

  int m = 0;
  while ((1LL<<(m + 1)) <= n) ++m;

  int b = 0;
  for (int i = 0; i < m; ++i)
    if ((n>>i)&1) ++b;

  for (int i = 0; i < m; ++i)
    if ((n>>i)&1) cout << "CALL " << b + m - i << endl;

  for (int i = 0; i < m; ++i)
    cout << "CALL " << b + i + 1 << endl;

  cout << "BELL&RET" << endl;
}
```

Timus 1279. Warehouse

1279. Warehouse

Time Limit: 1.0 second
Memory Limit: 16 MB

A warehouse is an N×M meter sized rectangle, which is divided into sections of 1×1 meter. The warehouse is served by a roof-mounted crane. 1×1 meter sized containers may be stacked one atop another in each section.

A new lot of K containers of the same kind arrived to the warehouse. It was decided to place the new containers so that the sections having less containers would be filled first.

For example, let N=3, M=3, K=10 and the number of containers in each section is represented in the table below.

	1	2	3
X	1	2	3
Y	4	5	6
Z	7	8	9

In this case the new containers will be sequentially placed in sections: x1, x1, x2, x1, x2, x3, x1, x2, x3, y1. After that the heights of the sections will be as follows:

	1	2	3
x	5	5	5
y	5	5	6
z	7	8	9

Your task is to write a program, which determines the minimum height of the sections after placing new containers.

Input

The first line contains three integer numbers N, M, K ($0 < N, M \le 100$, $0 < K \le 10^7$), where N and M are dimensions of the warehouse, and K is the number of new containers.

Each of the following N lines contains M space-separated integer numbers ranging from 1 to 1000. These numbers are the heights of the corresponding sections.

Output

Output the minimum height of the warehouse sections after the placement of new containers.

Samples

input	output
2 2 3 1 3 2 4	3
3 3 10 1 2 3 4 5 6 7 8 9	5

Problem Author: © Sergey G. Volchenkov, 2003(volchenkov@yandex.ru); Vladimir N. Pinaev, 2003(vpinaev@mail.ru; http://www.pic200x.chat.ru); Michael Y. Kopachev, 2003 (mkopachev@krista.ru).
Problem Source: 2003-2004 ACM Central Region of Russia Quarterfinal Programming Contest, Rybinsk, October 15-16, 2003

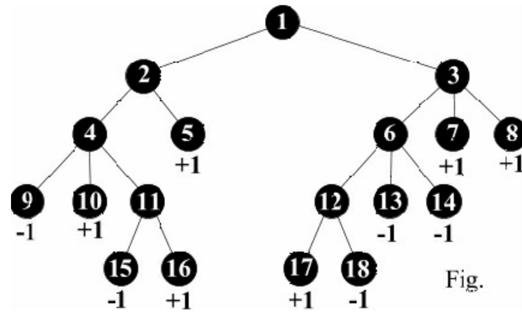
```
/*
Timus 1279. Warehouse
Count how many boxes can fit in each level of the warehouse. Let's name F[h]
the number of new boxes that fit in level h.
Let V[h] be the number of new boxes we can fit allowing to put boxes only at
levels not above h. V[h] can be computed as V[h]=F[1]+F[2]+...+F[h].
The answer is the minimum h such that K<=V[h].
*/
#include <iostream>
#include <vector>
using namespace std;

typedef long long ll;
typedef vector<int> Vi;

const int HMAX = 1000;

int main() {
  int R, C, K;
  cin >> R >> C >> K;

  int N = R*C;
  Vi V(HMAX+1, 0);

  for (int i = 0; i < N; ++i) {
    int t;
    cin >> t;
    ++V[t];
  }

  int mini = HMAX;
  for (int i = 0; i <= HMAX; ++i)
    if (V[i] != 0) {
      mini = i;
      break;
    }

  int e = mini, d = 100000000;
  int r = d;
  while (e <= d) {
    int m = (e + d)/2;
    ll k = 0;
    for (int i = 0; i <= HMAX and i < m; ++i) k += ll(m - i)*V[i];
    if (K <= k) {
      if (K == k) r = m;
      else r = m - 1;
      d = m - 1;
    }
    else e = m + 1;
  }

  cout << r << endl;
}
```

Timus 1282. Game Tree

1282. Game Tree

Time Limit: 1.0 second
Memory Limit: 16 MB

A game for two players is determined by its tree. The competitors make moves in turn. The first competitor starts the game. The game ends up with either a draw, or a victory of one of the players. The leaf nodes of the tree of this game may have values equal to one of three numbers: "+1" – victory of the first competitor, "–1" – victory of the second competitor, "0" – draw.



You have to find out who will win if both competitors follow the right strategy.

Input

The nodes of the tree are numbered with successive integer numbers. The root of the tree always has number 1.

The first line contains an integer N ($1 < N \leq 1000$) – the number of the nodes in the game tree. Next N-1 lines describe the nodes – one line for each node (with exception for the first node). The second line will contain the description of the second node of the tree, the third line – the description of the third node, and so on.

If the node is a leaf of the tree, the first symbol of the line is "L", followed by a space, then the number of the ancestor of this node goes, another space, and the result of the game (+1: victory of the first player, –1: victory of the second one, 0: draw).

If the node is an inner one then the line contains the first symbol "N", a space character and the number of the ancestor of this node.

Output

contains "–1" if the second competitor wins, "+1" if so does the first and "0" if the result is a draw.

Samples

input	output
7 N 1 N 1 L 2 -1 L 2 +1 L 3 +1 L 3 +1	+1
7 N 1 N 1 L 2 -1 L 2 +1 L 3 +1 L 3 0	0
18 N 1 N 1 N 2 L 2 +1 N 3 L 3 +1 L 3 +1 L 4 -1 L 4 +1 N 4 N 6 L 6 -1 L 6 -1 L 11 -1 L 11 +1	+1

L 12 +1 L 12 -1	

Problem Author: © Sergey G. Volchenkov, 2003(volchenkov@yandex.ru); Vladimir N. Pinaev, 2003(vpinaev@mail.ru; http://www.pic200x.chat.ru); Michael Y. Kopachev, 2003 (mkopachev@krista.ru).
Problem Source: 2003-2004 ACM Central Region of Russia Quarterfinal Programming Contest, Rybinsk, October 15-16, 2003

```
/*
Timus 1282. Game Tree
This problem can be solved using the standard minimax algorithm: recursively
compute the best move assuming that the opponent will play optimally.
*/
#include <iostream>
#include <vector>

using namespace std;

int minimax(vector<vector<int> >& children, vector<int>& value, int nd, int player)
    {
  if (children[nd].size() == 0) {
    return value[nd];
  }
  int ret;
  if (player == 0) ret = -42;
  else ret = 42;
  for (int i = 0; i < int(children[nd].size()); ++i) {
    if (player == 0) ret = max(ret, minimax(children, value, children[nd][i], 1));
    else ret = min(ret, minimax(children, value, children[nd][i], 0));
  }
  return ret;
}

int main() {
  int n;
  cin >> n;
  vector<vector<int> > children(n);
  vector<int> value(n);
  for (int i = 1; i < n; ++i) {
    char type;
    int ancestor;
    cin >> type >> ancestor;
    --ancestor;
    children[ancestor].push_back(i);
    if (type == 'L') {
      cin >> value[i];
    }
  }
  int ans = minimax(children, value, 0, 0);
  if (ans == 1) cout << "+1" << endl;
  else cout << ans << endl;
}
```

Timus 1453. Queen

1453. Queen

Time Limit: 1.0 second
Memory Limit: 16 MB

Background

Years and years have passed since famous grand master Paul V. Pawnstein invented N-dimensional chess and formulated the classical problem "Queen II". Since then hundreds of researchers tried their best to cognize its inconceivable essence and get to the solution, but only few of them have finally succeeded. The others, as usual, began to whimper and complain of this problem's baffling complexity. "Give us something easier! Let it be not two moves, but only one, please?", - demanded those thoughtless comrades.

But the grand master foresaw it. He knew exactly, that a scalability of the limitations is a great thing. And as if he tried to scoff at the simplicity lovers, Mr. Pawnstein posed a problem which was known as "Queen I".

Problem

A board in N-dimensional chess is N-dimensional cube S*S*...*S cells in size. A cell in one of its corners (this corner is chosen at will) has coordinates (1, 1, ..., 1), and a cell in the opposite corner has coordinates (S, S, ..., S).

A rook in N-dimensional chess makes its move shifting by any non-zero number of cells along one of its coordinates. A bishop in N-dimensional chess makes its move shifting by any non-zero number of cells along all its coordinates at once, and these shifts must be equal to each other by their absolute values. A queen in N-dimensional chess can make its move both as a bishop and as a rook.

A queen is situated on empty chess-board in a cell with coordinates (C[1], C[2], ..., C[N]). You should calculate a number of different cells the queen can make its move to.

Input

The first line contains the integer numbers N (1 ≤ N ≤ 10000) and S (2 ≤ S ≤ 100000). The second line contains N integer numbers C[i] (1 ≤ C[i] ≤ S).

Output

You should output the solution of "Queen I" problem.

Sample

input	output
3 3 1 2 3	8

Hint

Let us consider three-dimensional chess-board 3*3*3 cells in size. If a queen is initially located in the cell with coordinates (1, 2, 3) it can make its move to the cells with coordinates (2, 2, 3), (3, 2, 3), (1, 1, 3), (1, 3, 3), (1, 2, 1) and (1, 2, 2) moving as a rook, and to the cells with coordinates (2, 3, 2) and (2, 1, 2) moving as a bishop.

Problem Author: Dmitry Kovalioff, Ilya Grebnov, Nikita Rybak
Problem Source: Timus Top Coders: Second Challenge

```java
/*
Timus 1453. Queen
If n = 1, the number of moves is S. If N > 1, we can split the moves into 2
groups: rook-like moves and bishop-like moves. The number of rook-like moves
is (S-1)*N. The number bishop-like moves can be counted by splitting them as
follows: consider the dimension in which the queen is closest to an edge of
the board, and let d[0] be the distance to the edge. The number of bishop-like
moves in which the queen moves towards the closest edge in this dimension is
d[0] * 2^(N-1), because it can move up to d[0] cells regardless of the
direction taken in all other N-1 dimensions. The remaining bishop-like moves
are the ones in which the queen moves away from the closest edge in this
dimension, and we can count them using the same splitting technique because
the first dimension considered will never limit the distance that the queen
can move.

Therefore, the number of bishop-like moves is
(sum for i = 0..N-1 of d[i]*2^(N-1-i)) + (S-1-d[N-1])
where d[i] are the distances to the edge of the board in each dimension,
sorted increasingly, and (S-1-d[N-1]) is the number of moves possible when
the queen moves away from the closest edge of the board in every dimension.
*/
import java.math.*;
import java.util.*;
import java.io.*;

public class Main1453 {
  public static void main(String[] args) {
    InputReader cin =  new InputReader(System.in);
    PrintWriter cout =  new PrintWriter(System.out);

    int n, s;
    n = cin.nextInt();
    s = cin.nextInt();

    ArrayList<Integer> c = new ArrayList<Integer>(n);
    for (int i = 0; i < n; ++i) {
      int ci;
      ci = cin.nextInt();
      ci = Math.min(ci-1, s-ci);
      c.add(ci);
    }
    Collections.sort(c);
    BigInteger ans = BigInteger.valueOf(0);
    for (int i = 0; i < n; ++i) {
      ans = ans.add(BigInteger.valueOf(c.get(i)).shiftLeft(n-1-i));
    }
    ans = ans.add(BigInteger.valueOf(s-1-c.get(n-1)));
    if (n > 1) {
      ans = ans.add(BigInteger.valueOf((s-1)*n));
    }
    cout.println(ans);
    cout.flush();
  }
}

class InputReader {
  private BufferedReader reader;
  private StringTokenizer tokenizer;
  public InputReader(InputStream stream) {
    reader = new BufferedReader(new InputStreamReader(stream));
    tokenizer = null;
```

```
  }
  public String next() {
    while (tokenizer==null || !tokenizer.hasMoreTokens()) {
      try {
        tokenizer = new StringTokenizer(reader.readLine());
      } catch(IOException e) {
        throw new RuntimeException(e);
      }
    }
    return tokenizer.nextToken();
  }
  public int nextInt() {
    return Integer.parseInt(next());
  }
}
```

1456. Jedi Riddle 2

Time Limit: 0.25 second
Memory Limit: 16 MB

Background

Everyone wants to be respected and famous. At that many of us forget that most people became respected and famous after death only. For instance, let us take system administrator Vasily "Jedi Master" Slipman. For the first time his name was mentioned in connection with the sensational case concerning password decoding. That time nearly a half of the humanity strived for getting access to an archive, which contained some information of great pith and moment (this story is fully described in the problem "Jedi riddle").

After that case, the greatest cryptography scientists became interested in uncommon personality of Mr. Slipman and his research activities. In-depth study of Vasily's scientific heritage revealed, that during the last years of his life he was trying to understand the nature of the Force itself. Mr. Slipman tried to find the legendary Number of Force. In the volume XII of "The Book of the Light and the Darkness" an amazing experiment is described in detail:

Problem

"...And I took the Number of Light A and the Number of Darkness N. However the Darkness and the Light cannot be disjointed, so I took the One since Its essence is unknowable and sacred. And then I multiplied the One by the Number of Light, divided the result by the Number of Darkness and took the Remainder Z[1] = (1*A) modulo N. Then I multiplied the Remainder by the Number of Light, divided the result by the Number of Darkness and took the Remainder Z[2] = (Z[1]*A) modulo N once more. Being impatient, I was multiplying, dividing and taking the new Remainders Z[i] again and again... Until the day came when I understood I had been blind. The One is a key to the Force, the Alpha and the Omega, the Beginning and the Ending. I returned to my work with the eagerness I had never felt before. Because I knew - the Number of Force X will be found as soon as some Remainder Z[X] is equal to the One. And may the Force be with me..."

Input

The only line contains the integer Numbers A and N ($2 \le A < N \le 10^9$).

Output

You should output the minimal positive Number X, if it exists. Otherwise you should output zero.

Sample

input	output
7 20	4

Hint

In the sample, the Remainders Z[1] = (1*7) modulo 20 = 7, Z[2] = (7*7) modulo 20 = 9, Z[3] = (9*7) modulo 20 = 3 and Z[4] = (3*7) modulo 20 = 1.

Problem Author: Ilya Grebnov, Dmitry Kovalioff, Nikita Rybak
Problem Source: Timus Top Coders: Second Challenge

```
/*
Timus 1456. Jedi Riddle 2
The fact that A^X = 1 mod N implies that
A^X_1 = 1 mod p_1^e_1
A^X_2 = 1 mod p_2^e_2
...
A^X_k = 1 mod p_k^e_k
where p_1^e_1 * p_2^e_2 * ... * p_k^e_k = N is the prime factorization of N,
and X_i <= X for all i. In fact, if X is minimal and X_i are minimal for all i,
X = LCM(X_1, X_2, ..., X_k). The values X_i can be computed using fermat's
little theorem.
*/
#include <algorithm>
#include <iostream>
#include <map>
#include <vector>

using namespace std;

typedef long long ll;

vector<ll> fac, primes;

ll pow_mod(ll a, int b, ll m) {
  if (b == 0) return 1;
  ll t = pow_mod(a, b/2, m);
  t = (t*t)%m;
  if (b%2 == 1) t = (t*a)%m;
  return t;
}

ll pow(ll a, int b) {
  if (b == 0) return 1;
  ll t = pow(a, b/2);
  t = t*t;
  if (b%2 == 1) t = t*a;
  return t;
}

ll gcd(ll a, ll b) {
  if (b == 0) return a;
  return gcd(b, a%b);
}

ll lcm(ll a, ll b) {
  return (a/gcd(a, b))*b;
}

vector<ll> factorization(ll n) {
  vector<ll> ret;
  for (; n > 1;) {
    if (n < int(fac.size())) {
      if (fac[n] == -1) {
        ret.push_back(n);
        n = 1;
      }
      else {
        ret.push_back(fac[n]);
        n /= fac[n];
      }
    }
```

```
    else {
      int ind = -1;
      for (int i = 0; i < int(primes.size()) && primes[i]*primes[i] <= n && ind ==
          -1; ++i) {
        if (n%primes[i] == 0) {
          ind = i;
        }
      }
      if (ind == -1) {
        ret.push_back(n);
        n = 1;
      }
      else {
        ret.push_back(primes[ind]);
        n /= primes[ind];
      }
    }
  }
  return ret;
}

ll number_of_force(ll a, ll p, int alpha) {
  ll ret = -1;
  ll o = pow(p, alpha-1)*(p-1);
  ll p_to_the_alpha = pow(p, alpha);
  for (ll d = 1; d*d <= o; ++d) if (o%d == 0) {
    if (pow_mod(a, d, p_to_the_alpha) == 1) return d;
    if (pow_mod(a, o/d, p_to_the_alpha) == 1) ret = o/d;
  }
  return ret;
}

int main() {
  fac = vector<ll>(1000000, -1);
  fac[0] = fac[1] = 0;
  for (int i = 2; i*i <= int(fac.size()); ++i) if (fac[i] == -1) {
    for (int j = 2*i; j < int(fac.size()); j += i) {
      fac[j] = i;
    }
  }
  primes = vector<ll>(0);
  for (int i = 0; i < int(fac.size()); ++i) {
    if (fac[i] == -1) {
      primes.push_back(i);
    }
  }
  ll a, n;
  cin >> a >> n;
  vector<ll> fn = factorization(n);
  sort(fn.begin(), fn.end());
  map<ll, int> fnm;
  for (int i = 0; i < int(fn.size()); ++i) {
    ++fnm[fn[i]];
  }
  ll ans = 1;
  for (map<ll, int>::iterator it = fnm.begin(); it != fnm.end() && ans != 0; ++it) {
    ll numf = number_of_force(a, it->first, it->second);
    if (numf == -1) {
      ans = 0;
    }
    else {
```

```
      ans = lcm(ans, numf);
    }
  }
  cout << ans << endl;
}
```

1457. Heating Main

Time Limit: 1.0 second
Memory Limit: 16 MB

Background

I like my hometown very much. Those dilapidated buildings rising proudly above the city and streets dug up as far back as the last century inspire me greatly. Crowds of everlastingly offended working class representatives, stupid students escaping the army, retirees hunting for empty bottles, extremely nice vagrants and amiable young people wearing black caps, leather jackets and baseball bats - all of them are so close to me.

Furthermore, an old man lives in the city. To be more precise, he had lived in the city until his house was demolished and a new casino was built on its place. No wonder, because the casino is much more useful for the city than some old man. The foundations of market economy are impossible to resist.

So the old man had to resettle into a heating main, which lies straight under the city. Despite all its disadvantages, inhabitation in a heating main implies free water supply, heating and no rent at all. In short, the old man is going to live a worth old age. Thank the government and the President for such a great concern.

No matter how gorgeous a life in the heating main is, it is necessary for the old man to get out from the heating main to the city and visit one of some important places. Sometimes he has to make sure that there are no free drugs at the clinic, provide himself with foodstuffs at the market dump, get a pension at the post-office or give this pension to the grandson - it is just enough to buy an ice cream!

Problem

The heating main was build under Stalin, so it is a straight branchless tunnel. Each point of it is defined by its main offset. The main offset of the start point, which is located under the courthouse, is zero. The distance between any two points of the heating main equals to the absolute value of the difference between their main offsets.

It appeared that the heating main lies under all N places visited by the old man. For each gulley, which leads from the heating main straight to one of the places, the main offset P[i] was found. The old man can get out from the heating main through these gulleys only. If he tries to use another gulley, he would be immediately caught by watchful policemen as a dangerous vagrant.

The old man is rather old, and his effort to pass some distance is proportionate to the square of this distance. That is why the old man would like to live in some point of the heating main so that the arithmetic mean of the efforts to reach each of the places is minimal.

Input

The first line contains the integer number N ($1 \le N \le 1000$). The second line contains N integer numbers P[i] ($0 \le P[i] \le 10^6$).

Output

You should output the main offset of the desired point. The offset should be printed with at least six digits after decimal point. If the problem has several solutions, you should output any of them.

Sample

input	output
3 7 4 5	5.333333

Problem Author: Nikita Rybak, Dmitry Kovalioff, Ilya Grebnov
Problem Source: Timus Top Coders: Second Challenge

```cpp
/*
Timus 1457. Heating Main
The function that we want to minimize is
(x-P[0])^2 + (x-P[1])^2 + ... + (x-P[n])^2
and its derivative is
2*(x-P[0]) + 2*(x-P[1]) + ... + 2*(x-P[n])
therefore at x = (P[0] + P[1] + ... + P[n])/n it has a local minimum, which is
in fact a global minimum because the second derivative is always positive.
*/
#include <iostream>

using namespace std;

int main() {
  cout.setf(ios::fixed);
  cout.precision(6);
  int n;
  cin >> n;
  int psum = 0;
  for (int i = 0; i < n; ++i) {
    int p;
    cin >> p;
    psum += p;
  }
  cout << double(psum)/double(n) << endl;
}
```

1486. Equal Squares

Time Limit: 2.0 second
Memory Limit: 16 MB

During a discussion of problems at the Petrozavodsk Training Camp, Vova and Sasha argued about who of them could in 300 minutes find a pair of equal squares of the maximal size in a matrix of size $N \times M$ containing lowercase English letters. Squares could overlap each other but could not coincide. He who had found a pair of greater size won. Petr walked by, looked at the matrix, said that the optimal pair of squares had sides K, and walked on. Vova and Sasha still cannot find this pair. Can you help them?

Input

The first line contains integers N and M separated with a space. $1 \le N, M \le 500$. In the next N lines there is a matrix consisting of lowercase English letters, M symbols per line.

Output

In the first line, output the integer K which Petr said. In the next two lines, give coordinates of upper left corners of maximal equal squares. If there exist more than one pair of equal squares of size K, than you may output any of them. The upper left cell of the matrix has coordinates (1, 1), and the lower right cell has coordinates (N, M). If there are no equal squares in the matrix, then output 0.

Sample

input	output
5 10 ljkfghdfas isdfjksiye pgljkijlgp eyisdafdsi lnpglkfkjl	3 1 1 3 3

Problem Author: Vladimir Yakovlev
Problem Source: The XIth USU Programing Championship, October 7, 2006

```
/*
Timus 1486. Equal Squares
A binary search on K can be performed, because if for some value of K there is
a pair of equal squares there will also be a pair of equal squares for every
smaller value of K, and if for some value of K there is no pair of equal squares
there will not be any pair of equal squares for larger values of K. Once K is
fixed, hashing can be used to find two equal squares: the hashing function
maps a KxK square of symbols to its value when interpreted as a number, where
each symbol is a digit in base 27 ('a' is 1, 'b' is 2, ..., 'z' is 26). The
hash function of a square can be computed very efficiently from the value of a
neighbour square. In order to find a pair of equal squares just iterate over all
KxK-sized rectangles storing their hash values until a hash value is repeated.
*/
#include <iostream>
#include <string>
#include <vector>

#define FR first
#define SC second

using namespace std;

typedef unsigned long long ull;

const ull BASE = 27;
const ull HASH_SIZE = 100003;

ull pow(ull a, int b) {
  if (b == 0) return 1;
  ull x = pow(a, b/2);
  x *= x;
  if (b%2 == 1) x *= a;
  return x;
}

inline ull value(char c) {
  if (c >= 'a' && c <= 'z') return ull(c-'a'+1);
  return 0;
}

pair<int, int> repeated_pair(vector<string>& sym, int k) {
  int n = int(sym.size());
  int m = int(sym[0].size());
  ull pbk = pow(BASE, k);
  vector<vector<ull> > down(n+1, vector<ull>(m, 0));
  for (int j = 0; j < m; ++j) {
    for (int i = n-1; i >= 0; --i) {
      down[i][j] = down[i+1][j]*BASE + value(sym[i][j]);
      if (i+k < n) {
        down[i][j] -= pbk*value(sym[i+k][j]);
      }
    }
  }

  ull pbk2 = pow(BASE, k*k);
  vector<vector<ull> > right(n, vector<ull>(m+1, 0));
  for (int i = 0; i < n; ++i) {
    for (int j = m-1; j >= 0; --j) {
      right[i][j] = right[i][j+1]*pbk + down[i][j];
      if (j+k < m) {
        right[i][j] -= pbk2*down[i][j+k];
```

```
      }
    }
  }

  vector<vector<pair<ull, int> > > t(HASH_SIZE);

  for (int i = 0; i+k <= n; ++i) {
    for (int j = 0; j+k <= m; ++j) {
      int ind = right[i][j]%HASH_SIZE;
      for (int l = 0; l < int(t[ind].size()); ++l) {
        if (t[ind][l].FR == right[i][j]) {
          return pair<int, int>(t[ind][l].SC, i*m+j);
        }
      }
      t[ind].push_back(pair<ull, int>(right[i][j], i*m+j));
    }
  }
  return pair<int, int>(-1, -1);
}

int main() {
  int n, m;
  cin >> n >> m;
  vector<string> sym(n);
  for (int i = 0; i < n; ++i) {
    cin >> sym[i];
  }
  int lo = 0, hi = min(n, m)+1;
  for (; hi-lo > 1;) {
    int k = (lo+hi)/2;
    pair<int, int> cur = repeated_pair(sym, k);
    if (cur.FR >= 0) lo = k;
    else hi = k;
  }
  cout << lo << endl;
  if (lo > 0) {
    pair<int, int> p = repeated_pair(sym, lo);
    cout << (p.FR/m)+1 << " " << (p.FR%m)+1 << endl;
    cout << (p.SC/m)+1 << " " << (p.SC%m)+1 << endl;
  }
}
```

1542. Autocompletion

Time Limit: 2.0 second
Memory Limit: 64 MB

The Japanese are infinitely in love with machinery that surrounds them. They follow closely all technical innovations and try to use the most modern and clever of them. Den and Sergey have an ingenious plan: they want to create a text editor that will win the Japanese over. The most important feature of the editor will be the autocompletion function. If a user has typed first several letters of a word, then the editor will automatically suggest the most probable endings.

Den and Sergey have collected a lot of Japanese texts. For each Japanese word they counted the number of times it was found in the texts. For the first several letters entered by a user, the editor must show no more than ten words starting with these letters that are most commonly used. These words will be arranged in the order of decreasing encounter frequencies.

Help Sergey and Den to turn over the market of text editors.

Input

The first line contains the number of words found in the texts N ($1 \le N \le 10^5$). Each of the following N lines contains a word w_i and an integer n_i separated with a space, where w_i is a nonempty sequence of lowercase Latin letters no longer than 15 symbols, and n_i ($1 \le n_i \le 10^6$) is the number of times this word is encountered in the texts. The (N + 2)th line contains a number M ($1 \le M \le 15000$). In each of the next M lines there is a word u_i (a nonempty sequence of lowercase Latin letters no longer than 15 symbols), which is the beginning of a word entered by a user.

Output

For each of the M lines, output the most commonly used Japanese words starting with u_i in the order of decreasing encounter frequency. If some words have equal frequencies, sort them lexicographically. If there are more than ten different words starting with the given sequence, output the first ten of them. The lists of words for each u_i must be separated by an empty line.

Sample

<table>
<tr><th>input</th><th>output</th></tr>
<tr><td><pre>5
kare 10
kanojo 20
karetachi 1
korosu 7
sakura 3
3
k
ka
kar</pre></td><td><pre>kanojo
kare
korosu
karetachi

kanojo
kare
karetachi

kare
karetachi</pre></td></tr>
</table>

Problem Author: Den Raskovalov
Problem Source: The 11th Urals Collegiate Programing Championship, Ekaterinburg, April 21, 2007

```
/*
Timus 1542. Autocompletion
Build a Trie containing the M words. The list of suggestions of each of the
M words is initially empty. Now, for each of the N words, try to find it in
the tree previously built and, while finding the word, if you reach a node where
one of the M words ends, then the word which ends there is for sure a prefix
of the one you are looking for in the tree. Update the suggestions list of
the word that ends there, adding or replacing (if the list contains 10
suggestions already) the new suggestion if necessary.
*/
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
#include <map>
#include <set>
#include <string>
#include <utility>
using namespace std;

typedef pair<int, int> P;
typedef pair<P, int> TR; // string endpoints, child's node
typedef map<char, TR> MAP;
typedef struct {
  int id; // id of the word that ends here (-1 if no such word exists)
  MAP ch; // children
} ST;

const int S_SIZE = 16*15010;
const int T_SIZE = 30010;

char S[S_SIZE];
int M;          // number of nodes
ST T[T_SIZE];   // trie

inline int size(P p) { return p.second - p.first + 1; }

void trie_init() {
  M = 1;
  for (int i = 0; i < T_SIZE; ++i) T[i].id = -1;
}

// inserts word S[a..b] with the specified id into the trie
// if the word already exists, then returns its id and does nothing
// otherwise, the word is inserted and the specified id is returned
int trie_insert(int a, int b, int id) {
  int t = 0;
  while (a <= b) {
    if (T[t].ch.count(S[a]) == 0) break;
    TR tr = T[t].ch[S[a]];
    int q = 1, m = min(b - a + 1, size(tr.first));
    while (q < m and S[a + q] == S[tr.first.first + q]) ++q;
    if (q == size(tr.first)) { t = tr.second; a += q; }
    else {
      TR nt(P(tr.first.first + q, tr.first.second), tr.second);
      T[M].ch[S[tr.first.first + q]] = nt;
      TR nt2(P(tr.first.first, tr.first.first + q - 1), M);
      T[t].ch[S[a]] = nt2; a += q; t = M++; break;
    }
  }
  if (a > b) {
```

```
    if (T[t].id == -1) { T[t].id = id; return id; }
    return T[t].id;
  }
  T[M].id = id; T[t].ch[S[a]] = TR(P(a, b), M++);
  return id;
}

typedef set<P> SET;
typedef SET::iterator Sit;

char word[100010][16];
int freq[100010];
int original[100010], posicio[100010];
int ident[15010];
SET res[15010];

bool sortf(int a, int b) {
  return strcmp(word[a], word[b]) < 0;
}

int main() {
  trie_init();
  int sp = 0, id = 0;

  int n;
  scanf("%d", &n);
  for (int i = 0; i < n; ++i) scanf("%s%d", word[i], &freq[i]);

  for (int i = 0; i < n; ++i) original[i] = i;
  sort(original, original + n, sortf);
  for (int i = 0; i < n; ++i) posicio[original[i]] = i;

  int m;
  scanf("%d", &m);
  for (int i = 0; i < m; ++i) {
    scanf("%s", &S[sp]);
    int t = strlen(&S[sp]);
    int tmp = trie_insert(sp, sp + t - 1, id);
    if (tmp == id) ++id;
    ident[i] = tmp;

    sp += t;
  }

  for (int i = 0; i < n; ++i) {
    int s = strlen(word[i]), p = 0, t = 0;
    while (p < s) {
      if (T[t].ch.count(word[i][p]) == 0) break;
      TR tr = T[t].ch[word[i][p]];
      int sz = size(tr.first);
      if (s - p < sz) break;
      bool ok = true;
      for (int j = 0; ok and j < sz; ++j)
        if (word[i][p + j] != S[tr.first.first + j])
          ok = false;
      if (not ok) break;

      p += sz;
      t = tr.second;

      if (T[t].id != -1) {
```

```
        res[T[t].id].insert(P(-freq[i], posicio[i]));
        if (res[T[t].id].size() > 10) {
          Sit it = res[T[t].id].end();
          --it;
          res[T[t].id].erase(it);
        }
      }
    }
  }

  for (int i = 0; i < m; ++i) {
    if (i) printf("\n");
    int t = ident[i];
    for (Sit it = res[t].begin(); it != res[t].end(); ++it)
      printf("%s\n", word[original[it->second]]);
  }
}
```

1544. Classmates 3

Time Limit: 1.0 second
Memory Limit: 64 MB

Tanya (see problems Classmates and Classmates 2) has grown up and works as a computer science teacher at school. New Japanese software has been installed in her classroom recently. Now each computer can communicate with other computers in the classroom using a Japanese protocol or a European protocol and can switch between these protocols. When a computer gets a command to change protocol, it sends this command automatically to the computers to which it is connected and then switches itself immediately to the new protocol. Unfortunately, the protocols are incompatible, so a command to change protocol can be sent only to computers that use the same protocol as the computer that sends the command. Note that each of the computers that has received the command will send it back to the computer from which it was received, but that computer will not understand it because it will already use the new protocol.

At the start of a lesson Tanya has discovered that after the installation of the new software each computer was assigned at random one of the two available protocols. In order to conduct the lesson, Tanya has to switch all the computers to the same protocol as soon as possible.

Tanya can ask one of the pupils to change protocol at his or her computer, for example, from Japanese to European. Then this computer and all computers that use the Japanese protocol and are connected to that computer directly or via computers with the Japanese protocol will switch to the European protocol. All other computers will be unaffected. In the case when one of the computers is switched from the European protocol to the Japanese protocol, the result will be similar. Help Tanya to switch all the computers to the same protocol by means of the minimal number of requests to her pupils.

Input

The first line contains the number of computers in the class N ($1 \le N \le 50$) and the number of connections between them M. In the next line there are N letters **E** or **J**. If the i-th computer is using the European protocol, then the i-th letter is **E**, otherwise it is **J**. The letters in the line are separated with a space. Each of the next M lines contains two different integers a_i and b_i ($1 \le a_i, b_i \le N$), which are the numbers of computers that have a direct connection. It is known that all computers in the class are connected to each other directly or via other computers.

Output

In the first line output an integer K, which is the minimal number of requests to switch protocol that Tanya should make to her pupils in order to switch all the computers to the same protocol. Then output K lines describing the requests. A request to switch the i-th computer to the European protocol must be written as "i **E**", and the request to switch it to the Japanese protocol must be written as "i **J**". If there are several solutions, output any of them.

Sample

input	output
5 5 E E E J J 1 2 1 3 1 4 4 2 5 2	1 1 J

Problem Author: Folklore (prepared by Sergey Pupyrev)
Problem Source: The 11th Urals Collegiate Programing Championship, Ekaterinburg, April 21, 2007

```
/*
Timus 1544. Classmates 3
Note that a command that changes the protocol of a computer A will also change
the protocol of all the neighbours of A that were using the same protocol, and
a command that computer B sends to a neighbour computer C will have no effect
if C was not using the same protocol as B before B switched. Thus, we can see
the network of computers as a graph of white and black nodes, where a change
command issued to a node S will propagate through the connex component of nodes
that are reachable from S without changing color.

Now, we can turn this graph into another graph where connected components of the
same color are nodes, and there is an edge between two nodes if and only if
there was an edge between some node of one component and one of the other in the
original graph. Notice how what a change command does is merge the component
where the command is issued to all of its neighbour components.

Our goal is to reduce the number of components to 1 using the minimal number of
change commands. This can be achived by choosing the connected component that
has a smallest distance to the farthest connected component and issuing change
commands from any of the computers in it, thus reducing the "radius" of the
graph. Once the computer has been chosen, it is enough to issue from it as many
alternating protocol change commands as the distance to the farthest connected
component.
*/
#include <iostream>
#include <queue>
#include <vector>

#define FR first
#define SC second

using namespace std;

const int INF = 1000000000;

typedef pair<int, int> PII;

int main() {
  int n, m;
  cin >> n >> m;
  vector<bool> color(n);
  for (int i = 0; i < n; ++i) {
    char c;
    cin >> c;
    color[i] = (c == 'J');
  }
  vector<vector<int> > g(n);
  for (int i = 0; i < m; ++i) {
    int a, b;
    cin >> a >> b;
    --a;
    --b;
    g[a].push_back(b);
    g[b].push_back(a);
  }
  int best = -1, bmxdist = INF;
  for (int src = 0; src < n; ++src) {
    vector<int> dist(n, INF);
    priority_queue<PII, vector<PII>, greater<PII> > pq;
    dist[src] = 0;
    pq.push(PII(0, src));
```

```
    for (; !pq.empty();) {
      int dst = pq.top().FR;
      int cur = pq.top().SC;
      pq.pop();
      if (dst > dist[cur]) continue;
      for (int i = 0; i < int(g[cur].size()); ++i) {
        if (color[g[cur][i]] != color[cur]) {
          if (dst+1 < dist[g[cur][i]]) {
            dist[g[cur][i]] = dst+1;
            pq.push(PII(dst+1, g[cur][i]));
          }
        }
        else {
          if (dst < dist[g[cur][i]]) {
            dist[g[cur][i]] = dst;
            pq.push(PII(dst, g[cur][i]));
          }
        }
      }
    }
    int mxdist = 0;
    for (int i = 0; i < n; ++i) {
      mxdist = max(mxdist, dist[i]);
    }
    if (mxdist < bmxdist) {
      best = src;
      bmxdist = mxdist;
    }
  }
  cout << bmxdist << endl;
  for (int i = 0; i < bmxdist; ++i) {
    bool eng = color[best];
    if (i%2 == 1) eng = !eng;
    if (eng) cout << best+1 << " E" << endl;
    else cout << best+1 << " J" << endl;
  }
}
```

1547. Password Search

Time Limit: 1.0 second
Memory Limit: 64 MB

After his trip to Japan, Vova has forgotten his password to Timus Online Judge. Fortunately, students of the Ural State University have access to a powerful multiprocessor computer MVS-1000, and Vova can be allowed to use *M* processors for solving complex mathematical problems. Vova wants to use the supercomputer for a simple search of passwords. He remembers that his password is no longer than *N* symbols and consists of lowercase Latin letters. First he wants to check all words of length 1 in the lexicographic order (that is, **a**, **b**, …, **z**), then all words of length 2 in the same order (that is, **aa**, **ab**, …, **zz**), and so on.

In order to use the supercomputer with maximal efficiency, the search must be distributed equally between all processors: the first portion of words is checked by the first processor, the second portion is checked by the second processor, and so on. If it is impossible to distribute the work equally, let the first several processors check one word more than the remaining processors. Vova wants to know the range of words for each processor.

Input

The only line of the input contains the integers N and M ($1 \le N, M \le 50$). It is guaranteed that the number of words to be checked is no less than the number of processors.

Output

Output M lines. Each line must contain the range of words that will be checked by the corresponding processor. See the required format in the sample.

Sample

input	output

`5 4`	`a-fssst`
	`fsssu-mmmmn`
	`mmmmo-tgggg`
	`tgggh-zzzzz`

Problem Author: Vladimir Yakovlev
Problem Source: The 11th Urals Collegiate Programing Championship, Ekaterinburg, April 21, 2007

```java
/*
Timus 1547. Password Search
Calculate the number of different passwords and the interval of
them that has to be executed on each processor. Now, the first
and last password of each interval has to be found. In order to
find the m-th password for any m, first determine its length.
Suppose now that we are looking for the m-th password of length
d. First, guess what the first letter is: if m is not greater
than the number of d-1 characters long passwords, the first
letter must be 'a', otherwise, if m is not greater than two
times the number of passwords of length d-1, it must be a 'b',
and so on. Once the first letter is determined, find the next
ones in the same way.
*/
import java.math.*;
import java.util.*;
import java.io.*;

public class Main1547 {
  public static String calc(BigInteger m) {
    int mida = 1;
    while (m.compareTo(BigInteger.valueOf(26).pow(mida)) >= 0) {
      m = m.subtract(BigInteger.valueOf(26).pow(mida));
      ++mida;
    }
    String res = "";
    for (int k = 0; k < mida; ++k) {
      BigInteger tmp = BigInteger.valueOf(26).pow(mida - k - 1);
      for (int i = 0; i < 26; ++i) {
        if (m.compareTo(tmp) < 0) {
          res += (char) ('a' + i);
          break;
        }
        else m = m.subtract(tmp);
      }
    }
    return res;
  }
  public static void main(String[] args) {
    InputStream inputStream = System.in;
    OutputStream outputStream = System.out;
    InputReader cin = new InputReader(inputStream);
    PrintWriter cout = new PrintWriter(outputStream);

    int n = cin.nextInt();
    int m = cin.nextInt();

    BigInteger total = BigInteger.valueOf(0);
    for (int i = 1; i <= n; ++i)
      total = total.add(BigInteger.valueOf(26).pow(i));
    BigInteger grup = total.divide(BigInteger.valueOf(m));

    int more = total.mod(BigInteger.valueOf(m)).intValue();

    BigInteger quin = BigInteger.valueOf(0);
    for (int i = 0; i < m; ++i) {
      BigInteger t = grup;
      if (i < more) t = t.add(BigInteger.valueOf(1));

      String a = calc(quin);
      String b = calc(quin.add(t).subtract(BigInteger.valueOf(1)));
```

```
      cout.println(a + "-" + b);
      quin = quin.add(t);
    }
    cout.flush();
  }
}

class InputReader {
  private BufferedReader reader;
  private StringTokenizer tokenizer;

  public InputReader(InputStream stream) {
    reader = new BufferedReader(new InputStreamReader(stream));
    tokenizer = null;
  }

  public String next() {
    while (tokenizer == null || !tokenizer.hasMoreTokens()) {
      try {
        tokenizer = new StringTokenizer(reader.readLine());
      } catch (IOException e) {
        throw new RuntimeException(e);
      }
    }
    return tokenizer.nextToken();
  }

  public int nextInt() {
    return Integer.parseInt(next());
  }
}
```

1549. Another Japanese Puzzle

Time Limit: 1.0 second
Memory Limit: 64 MB

A lecturer of the Ural State University has bought an amusing toy in Tokyo: a small airplane and a set of plastic plates. The plates can be put together to form a path for the plane the same way as a puzzle can be assembled from pieces. There are many ways to put together the plates, but if one makes it the right way, a map of Japan is assembled, and the plane will go along a closed path visiting all major Japan's sights. Try to guess what the lecturer's wife said when she saw the toy—“Thank you!” or “Was it necessary to spend so much money on such rubbish?” No, she said: “What a pity that you haven't bought several sets! We could assemble a much longer path!”

Imagine that you have many plates with path segments. What is the longest closed path that you can assemble?

The plates can be assumed to be squares of equal size, and the path always connects the centers of two sides of a square. It means that there are two kinds of plates: with straight lines and with turns.

Input

The only line of the input contains two integers: the number of plates with straight segments S and the number of plates with turns T ($0 \le S, T \le 1000$, $S + T > 0$).

Output

In the first line output the maximal number of plates N that can be used to assemble a path for the plane. In the second line output the path in the following format: a line of length N consisting of letters **F**, **L**, and **R**. Here **F** means that the corresponding segment of the path is straight, **L** denotes a left turn, and **R** denotes a right turn. The total number of letters **F** must not exceed S, and the total number of letters **L** and **R** must not exceed T. The path must be closed (the last square must join the first square) and the squares can't overlap. If it is impossible to assemble a closed path from the available plates, then output “Atawazu” (“Impossible”, Jap.).

Samples

input	output
`5 6`	`10` `FLLFRLLFLF`
`49 3`	`Atawazu`

Problem Author: Stanislav Vasilyev (prepared by Vladimir Yakovlev)
Problem Source: The 11th Urals Collegiate Programing Championship, Ekaterinburg, April 21, 2007

```
/*
Timus 1549. Another Japanese Puzzle
Let S be the number of plates with stright segments and let T the number of
plates with turns. In order to build a closed path, both S and T have to be
even. The following ways to use plates lead to a way to build a closed path:

Start with 4 turn plates forming a cycle. For the sake of clarity let's give
the closed path an orientation and say it is counterclockwise.

Straight plates can be added 2 by 2, by inserting one of them in a part that
goes to the left and the other one in a part that goes to the right.

Turn plates can be used 4 by 4, by inserting a RL (turn right, then turn left)
block in a part that goes up, and another one in a part that goes down. This
operation can cause an overlap unless there is some separation between the 2
top initial turn plates. In order to create a separation between the 2 top
initial turn plates one can insert between them either some straight plate or,
if there are no straight plates, a RLLR block simulating a FF block (boolean
t4t4 in the source code).

It is possible to use 2 straight plates + 2 turn plates by inserting an RL
block in a part that goes up, a straight plate in a part that goes left and a
straight plate in a part that goes down (boolean s2t2 in the source code). This
is useful when we have 2 remaining turn plates after using them 4 by 4.

Using these operations one can build a path for every pair (S, T) of the form
(2+2a, 4+2b)U(0, 4+4c)\(0, 8) for all natural (nonzero) a, b, c. All other
pairs (S, T) can't be built into a closed path using all the plates.

Given a pair (S, T), output the solution for the pair (S', T') having S' <= S
and T' <= T which has solution and maximizes S'+T'.
*/
#include <iostream>
#include <string>

using namespace std;

string longest_circuit(int s, int t) {
  s -= s%2;
  t -= t%2;
  bool s2t2 = false;
  if (t%4 == 2) {
    if (s == 0) {
      t -= 2;
    }
    else {
      s2t2 = true;
      s -= 2;
      t -= 2;
    }
  }

  if (t == 0) return "";
  t -= 4;

  bool t4t4 = false;
  if (!s2t2 && s == 0 && t > 0) {
    if (t == 4) {
      t = 0;
    }
    else {
```

```
      t4t4 = true;
      t -= 8;
    }
  }

  string up = "", left = "", down = "", right = "";
  if (s2t2) up += "RL";
  for (int i = 0; i < t/4; ++i) up += "RL";

  if (s2t2) left += "F";
  for (int i = 0; i < s/2; ++i) left += "F";
  if (t4t4) left += "RLLR";

  for (int i = 0; i < t/4; ++i) down += "RL";
  if (s2t2) down += "F";

  for (int i = 0; i < s/2; ++i) right += "F";
  if (t4t4) right += "RLLR";

  return up+"L"+left+"L"+down+"L"+right+"L";
}

int main() {
  int s, t;
  cin >> s >> t;
  string ans = longest_circuit(s, t);
  if (ans == "") cout << "Atawazu" << endl;
  else {
    cout << ans.size() << endl << ans << endl;
  }
}
```

1551. Sumo Tournament

Time Limit: 1.0 second
Memory Limit: 64 MB

A sumo tournament is held in Tokyo, in which 2^N sportsmen take part. In each encounter there is a winner, and the loser drops out of the tournament. Thus, in order to determine the winner of the tournament, it is necessary to conduct N rounds.

The organizers wish that in as many rounds as possible all encounters would be held between sumoists from different prefectures of Japan. For that they can forge the drawing results arbitrarily.

Input

The first line contains the number N ($1 \le N \le 10$). Each of the next 2^N lines contains the name of a sumouist and the prefecture which he presents. The name and prefecture are sequences of Latin letters of length not exceeding 30.

Output

Output the maximal number of rounds in which sumoists from the same prefecture will not fight each other regardless of the outcomes of encounters (that is, find the maximal possible K such that in at least K rounds all encounters will be between sumoists from different prefectures). The organizers can control the initial arrangement of sportsmen but can't control results of encounters.

Sample

input	output
3 Homasho Ishikawa Tamakasuga Tokyo Futeno Tochigi Takekaze Tokyo Kasugao Yamaguchi Kotoshogiku Ishikawa Kotomitsuki Tokyo Miyabiyama Shizuoka	1

Problem Author: Sergey Pupyrev
Problem Source: The 11th Urals Collegiate Programing Championship, Ekaterinburg, April 21, 2007

```
/*
Timus 1551. Sumo Tournament
The answer to this problem can be determined by N and M, where
M is the maximum amount of participants that represent the same
region.
It's easy to see that in case 2^(N-1) < M, the answer is 0.
If 2^(N-2) < M <= 2^(N-1), then the answer is 1.
If 2^(N-3) < M <= 2^(N-2), the answer is 2, and so on.
Knowing this, it should be easy to calculate the proper answer.
*/
#include <iostream>
#include <map>
#include <string>
using namespace std;

typedef map<string, int> MAP;
typedef MAP::iterator Mit;

int main() {
  int n;
  cin >> n;
  MAP mp;
  for (int i = 0; i < (1<<n); ++i) {
    string a, b;
    cin >> a >> b;
    ++mp[b];
  }

  int m = 0;
  for (Mit it = mp.begin(); it != mp.end(); ++it)
    m = max(m, it->second);

  int r = 0;
  while ((1<<r) < m) ++r;

  cout << n - r << endl;
}
```

1552. Brainfuck

Time Limit: 2.0 second
Memory Limit: 64 MB

Chairman of "Horns and hoofs" company, Mr. Phunt, decided to start advertising campaign. First of all, he wants to install an indicator panel on the main square of the city that will show advertisements of the company. So he charged the manager of the company, Mr. Balaganov, to do this job.

After analyzing offers of indicator panels, Balaganov ordered one at a price of only $19999.99. But when it was delivered, a little problem was found. The panel was programmable, but the instruction set of the processor was a subset of brainfuck language commands. The commands that processor was capable to execute were '>', '<', '+', '–' and '.', which are described in the table below. Moreover, this panel had very little memory for the program, so not every program typing a particular string will fit into memory.

Now Balaganov wants to know the minimal program that will output the given string. But because he is not very good at programming, he asks you to solve this problem. The brainfuck program is a sequence of commands executed sequentially (there are some exceptions, but panel processor cannot execute such commands). The brainfuck machine has, besides the program, an array of 30000 byte cells initialized to zeros and a pointer into this array. The pointer is initialized to point to the leftmost byte of the array.

Command	Description
>	Increment the pointer (to point to the next cell to the right). If the pointer before increment points to the rightmost byte of the array, then after increment it points to the leftmost byte.
<	Decrement the pointer (to point to the next cell to the left). If the pointer before decrement points to the leftmost byte of the array, then after increment it points to the rightmost byte.
+	Increment (increase by one) the byte at the pointer. If the value of the cell before increment is 255 then it becomes 0.
–	Decrement (decrease by one) the byte at the pointer. If the value of the cell before decrement is 0 then it becomes 255.
.	Output the value of the byte at the pointer.

Input

Input has one line containing the string brainfuck program must output. Every character of the string is a small English letter ('a'–'z'). The length of the string is not greater than 50. You may assume that optimal program will not have to modify more than four memory cells.

Output

Output one line with minimal brainfuck program. Any characters except '>', '<', '+', '–' and '.' are not allowed (quotes for clarity only). If there are several solutions any will be acceptable.

Sample

input	output
a	+++++++++++++++++++++++++++++++++++ +++++++++++++++++++++++++++++++++++ +++++++++++++++++++++++++++.

Hint

Please note that the sample output is divided into several lines only for convenience. In the real output whole program must be printed on a single line.

Problem Source: Novosibirsk SU Contest. Petrozavodsk training camp, September 2007

```
/*
Timus 1552. Brainfuck
An important observation is that it is possible to make a minimal sequence
of instructions using at most 4 cells of memory. This observation makes
possible to solve this problem with Dynamic Programming.
dp[n][m1][m2][m3][c] = length of the shortest sequence that produces the
text S[n..N-1], assuming the pointer is pointing at the c-th cell, the
content of three of the cells is determined by m1, m2, m3, and the content
of the c-th cell is determined implicitly by the last character printed
(the content of the c-th cell will be 0 when n=0, and S[n-1] otherwise).
In the implementation below, m1, m2, m3 can only be 0 or one
of the letters a-z. This is necessary to reduce the number of different
states and to fit in the allowed memory.
*/
#include <algorithm>
#include <iostream>
#include <string>
using namespace std;

int dp[52][27][27][27][4];
int que[52][27][27][27][4];

inline int fun(int ci, int vi) {
  if (ci == 26) return 'a' + vi;
  return abs(vi - ci);
}

int main() {
  string s;
  cin >> s;
  int n = s.size();

  int v[100];
  for (int i = 0; i < n; ++i) v[i] = s[i] - 'a';

  for (int sp = n - 1; sp >= 0; --sp) {
    int t[3];
    for (t[0] = 0; t[0] <= 26; ++t[0]) {
      for (t[1] = 0; t[1] <= 26; ++t[1]) {
        for (t[2] = 0; t[2] <= 26; ++t[2]) {
          for (int cp = 0; cp < 4; ++cp ) {
            int c[4];
            c[cp] = (sp == 0 ? 26 : v[sp - 1]);
            for (int i = 0, j = 0; i < 4; ++i) if (i != cp) c[i] = t[j++];
            int d0 = abs(0 - cp) + fun(c[0], v[sp]) + 1 + dp[sp + 1][c[1]][c[2]][c
                [3]][0];
            int d1 = abs(1 - cp) + fun(c[1], v[sp]) + 1 + dp[sp + 1][c[0]][c[2]][c
                [3]][1];
            int d2 = abs(2 - cp) + fun(c[2], v[sp]) + 1 + dp[sp + 1][c[0]][c[1]][c
                [3]][2];
            int d3 = abs(3 - cp) + fun(c[3], v[sp]) + 1 + dp[sp + 1][c[0]][c[1]][c
                [2]][3];
            int m = min(min(d0, d1), min(d2, d3));
            dp[sp][t[0]][t[1]][t[2]][cp] = m;
            if (d0 == m) que[sp][t[0]][t[1]][t[2]][cp] = 0;
            else if (d1 == m) que[sp][t[0]][t[1]][t[2]][cp] = 1;
            else if (d2 == m) que[sp][t[0]][t[1]][t[2]][cp] = 2;
            else que[sp][t[0]][t[1]][t[2]][cp] = 3;
          }
        }
      }
```

```
    }
  }

  int cp = 0;
  for (int i = 1; i < 4; ++i)
    if (dp[0][26][26][26][i] < dp[0][26][26][26][cp])
      cp = i;
  int c[4]; c[0] = c[1] = c[2] = c[3] = 26;
  for (int sp = 0; sp < n; ++sp) {
    int t[3];
    for (int i = 0, j = 0; i < 4; ++i) if (i != cp) t[j++] = c[i];
    int q = que[sp][t[0]][t[1]][t[2]][cp];
    int x = q - cp, y = (c[q] == 26 ? 'a' + v[sp] : v[sp] - c[q]);
    if (x < 0) cout << string(-x, '<');
    else cout << string(x, '>');
    if (y < 0) cout << string(-y, '-');
    else cout << string(y, '+');
    cout << ".";
    c[q] = v[sp];
    cp = q;
  }
  cout << endl;
}
```

1553. Caves and Tunnels

Time Limit: 3.0 second
Memory Limit: 64 MB

After landing on Mars surface, scientists found a strange system of caves connected by tunnels. So they began to research it using remote controlled robots. It was found out that there exists exactly one route between every pair of caves. But then scientists faced a particular problem. Sometimes in the caves faint explosions happen. They cause emission of radioactive isotopes and increase radiation level in the cave. Unfortunately robots don't stand radiation well. But for the research purposes they must travel from one cave to another. So scientists placed sensors in every cave to monitor radiation level in the caves. And now every time they move robots they want to know the maximal radiation level the robot will have to face during its relocation. So they asked you to write a program that will solve their problem.

Input

The first line of the input contains one integer N ($1 \le N \le 100000$) — the number of caves. Next $N - 1$ lines describe tunnels. Each of these lines contains a pair of integers a_i, b_i ($1 \le a_i, b_i \le N$) specifying the numbers of the caves connected by corresponding tunnel. The next line has an integer Q ($Q \le 100000$) representing the number of queries. The Q queries follow on a single line each. Every query has a form of "C U V", where C is a single character and can be either 'I' or 'G' representing the type of the query (quotes for clarity only). In the case of an 'I' query radiation level in U-th cave ($1 \le U \le N$) is incremented by V ($0 \le V \le 10000$). In the case of a 'G' query your program must output the maximal level of radiation on the way between caves with numbers U and V ($1 \le U, V \le N$) after all increases of radiation ('I' queries) specified before current query. It is assumed that initially radiation level is 0 in all caves, and it never decreases with time (because isotopes' half-life time is much larger than the time of observations).

Output

For every 'G' query output one line containing the maximal radiation level by itself.

Sample

input	output
4 1 2 2 3 2 4 6 I 1 1 G 1 1 G 3 4 I 2 3 G 1 1 G 3 4	1 0 1 3

Problem Source: Novosibirsk SU Contest. Petrozavodsk training camp, September 2007

```
/*
Timus 1553. Caves and Tunnels
This solution is based on the following idea:
Select an arbitrary node as the root of thre tree. Partition the tree
into disjoint subtrees of height 300 (ideally O(sqrt(N))). Construct an
Intervall Tree which will allow us to consult the maximum value in the
path from a given value to the root of the subtree it belongs to.
Enumerate the nodes in each subtree in preorder. When an increment
is performed on a node, all the nodes affected in its subtree form
an interval (they have consecutive numbers) and the Interval Tree
can be updated efficiently.
When a query of type G has to be performed, find the lowest common ancestor
of the two nodes, and then find the maximum value in each path (from one
node to the LCA, and from the other node to the LCA) as follows:
Traverse each subtree until reaching the subtree where the LCA is located
(using the Interval Tree for traversing each subtree in O(log(N))), and
finally iterate over every necessary node inside the last subtree to reach
the LCA (you will traverse 300 nodes at most).
*/
#include <iostream>
#include <vector>
using namespace std;

#pragma comment(linker, "/STACK:16777216")

#define PB push_back

typedef vector<int> Vi;
typedef vector<Vi> Mi;

typedef long long ll;
typedef vector<ll> Vll;
typedef vector<Vll> Mll;

const int LIM = 300;

int N, M, R;
Mi net;

Vi pare, quin, prof;
Vll value;

Vi root;
Vi mida;
Mll tree;

Vi first, last;

void dfs(int n, int m) {
  if (prof[n]%LIM == 0) {
    m = M++;
    root.PB(n);
    mida.PB(0);
  }
  ++mida[m];
  quin[n] = m;
  for (int i = 0; i < int(net[n].size()); ++i) {
    int f = net[n][i];
    if (f == pare[n]) continue;
    prof[f] = prof[n] + 1;
    pare[f] = n;
```

```
    dfs(f, m);
  }
}

int tmp;
int dfs(int n) {
  first[n] = last[n] = tmp++;
  for (int i = 0; i < int(net[n].size()); ++i) {
    int f = net[n][i];
    if (f == pare[n] or quin[n] != quin[f]) continue;
    last[n] = dfs(f);
  }
  return last[n];
}

void update(int n, int e, int d, int x, int y, ll v, Vll& t) {
  if (d < x or y < e) return;
  if (x <= e and d <= y) {
    t[n] = max(t[n], v);
    return;
  }
  int m = (e + d)/2;
  update(2*n, e, m, x, y, v, t);
  update(2*n + 1, m + 1, d, x, y, v, t);
}

void update(int n, ll v) {
  int t = quin[n];
  update(1, 0, mida[t] - 1, first[n], last[n], v, tree[t]);
}

ll query(int n, int e, int d, int x, Vll& t) {
  if (e == d) return t[n];
  ll res = t[n];
  int m = (e + d)/2;
  if (x <= m) res = max(res, query(2*n, e, m, x, t));
  else res = max(res, query(2*n + 1, m + 1, d, x, t));
  return res;
}

ll query(int n) {
  int t = quin[n];
  return query(1, 0, mida[t] - 1, first[n], tree[t]);
}

int main() {
  cin >> N;
  net = Mi(N);
  for (int i = 0; i < N - 1; ++i) {
    int a, b;
    cin >> a >> b;
    --a; --b;
    net[a].PB(b);
    net[b].PB(a);
  }
  R = 1234567%N;

  pare = quin = prof = Vi(N, 0);
  pare[R] = -1;
  dfs(R, 0);
```

```
  first = last = Vi(N);
  tree = Mll(M);
  for (int i = 0; i < M; ++i) {
    tree[i] = Vll(4*mida[i], 0);
    tmp = 0;
    dfs(root[i]);
  }

  value = Vll(N, 0);

  int queries;
  cin >> queries;
  while (queries--) {
    char op;
    int a, b;
    cin >> op >> a >> b;
    if (op == 'I') {
      --a;
      if (b > 0) {
        value[a] += b;
        update(a, value[a]);
      }
    }
    else {
      --a; --b;
      ll res = max(value[a], value[b]);

      if (prof[a] < prof[b]) swap(a, b);
      while (prof[a] - prof[b] > LIM) {
        res = max(res, query(a));
        if (a == root[quin[a]]) a = pare[a];
        else a = root[quin[a]];
      }
      while (prof[a] > prof[b]) {
        res = max(res, value[a]);
        a = pare[a];
      }

      while (root[quin[a]] != root[quin[b]]) {
        res = max(res, query(a));
        res = max(res, query(b));
        a = pare[root[quin[a]]];
        b = pare[root[quin[b]]];
      }

      while (a != b) {
        res = max(res, max(value[a], value[b]));
        a = pare[a];
        b = pare[b];
      }
      res = max(res, value[a]);

      cout << res << endl;
    }
  }
}
```

1554. Multiplicative Functions

Time Limit: 2.0 second
Memory Limit: 64 MB

In number theory, a multiplicative function is an arithmetic function $F(n)$ of the positive integer n with property that $F(1) = 1$ and whenever a and b are coprime ($\gcd(a, b) = 1$), then $F(ab) = F(a)F(b)$.

The function $E(n)$ defined by $E(n) = 1$ if $n = 1$ and $= 0$ if $n > 1$, is sometimes called *multiplication unit* for Dirichlet convolution or simply the unit function. If F and G are two multiplicative functions, one defines a new multiplicative function $F * G$, the Dirichlet convolution of F and G, by

$$(F * G)(n) = \sum_{d|n} F(d)G\left(\frac{n}{d}\right)$$

where the sum extends over all positive divisors d of n. With this operation, the set of all multiplicative functions turns into an abelian group; the identity element is E.

from *Wikipedia*, the free encyclopedia

In this task you have to find the inverse of a multiplicative function. To cope with overflow problem, we define arithmetic functions as: F: $\mathbf{N} \longrightarrow \mathbf{Z}_{2007}$ where $\mathbf{N}$ is the set of positive integers, and $\mathbf{Z}_{2007}$ is a residue ring (ring of integers 0–2006, where arithmetic operations + and × are performed modulo 2007). Function G is called the inverse of function F if and only if $F * G = G * F = E$.

You are given the first N values of function F, you need to find the first N values of the inverse function G.

Input

In the first line of the input one number N is written ($1 \le N \le 10^4$). In the second line values $F(1)$, $F(2)$, $F(3)$, …, $F(N)$ are listed. Numbers are separated by spaces. (Each value is nonnegative and doesn't exceed 2006.)

Output

In the first line of the output print first N values of inverse function G, separated by spaces: $G(1)$, $G(2)$, …, $G(N)$.

Sample

input	output
16 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1 2006 2006 0 2006 1 2006 0 0 1 2006 0 2006 1 1 0

Problem Source: Novosibirsk SU Contest. Petrozavodsk training camp, September 2007

```
/*
Timus 1554. Multiplicative Functions
Because (F*G)(1) = 1, G(1) has to be 1. For i > 1 (F*G)(i) = 0, and using the
values of G(j) for j < i that we have already computed the value of G(i) is
determined.
*/
#include <iostream>
#include <vector>

using namespace std;

const int MOD = 2007;

int main() {
  int n;
  cin >> n;
  vector<int> f(n+1);
  for (int i = 1; i <= n; ++i) {
    cin >> f[i];
  }
  vector<int> g(n+1);
  g[1] = 1;
  for (int i = 2; i <= n; ++i) {
    int sum = f[i];
    for (int d = 2; d*d <= i; ++d) if (i%d == 0) {
      sum += f[d]*g[i/d];
      if (d*d < i) {
        sum += f[i/d]*g[d];
      }
      sum %= MOD;
    }
    g[i] = (MOD-sum)%MOD;
  }
  for (int i = 1; i <= n; ++i) {
    if (i > 1) cout << " ";
    cout << g[i];
  }
  cout << endl;
}
```

1560. Elementary Symmetric Functions

Time Limit: 4.0 second
Memory Limit: 64 MB

In this task, you are to read an array of integer numbers ($A[1..N]$) and a sequence of M queries of two types:

1. Increase $A[i]$ by D.
2. Calculate first $K + 1$ elementary symmetric polynomials of the numbers of the interval $[L..R]$ ($S(0)$, $S(1)$, $S(2)$, …, $S(K)$).

Elementary symmetric polynomials of the interval $[L..R]$ are given by:

$$S(k) = \sum_{L \le I_1 < I_2 < \ldots < I_k \le R} A[I_1] \cdot A[I_2] \cdot \ldots \cdot A[I_k]$$

You should compute their values modulo prime P.

Input

The first line consists of three integer numbers: N (size of the array), M (number of queries) and P (prime number) ($1 \le N \le 80000$; $1 \le M \le 100000$; $1000 \le P \le 10^9$ (prime)).

The second line contains N integer numbers not exceeding 10^5 by absolute value (the initial values in the array).

The next M lines contain queries. Each query can be either increase or calculation query.

I *index delta* — increase *index*-th value by *delta* ($1 \le index \le N$; $-10^5 \le delta \le 10^5$).

C *left right K* — compute $S(0)$, …, $S(K)$ for the interval [*left*..*right*] ($1 \le left \le right \le N$; $1 \le K \le 4$; $K \le right - left + 1$).

All fields in each line are separated by spaces.

Output

For each calculation query print a line consisting of $K + 1$ numbers — $S(0)$ $S(1)$ … $S(K)$. These numbers must be nonnegative and less than P.

Sample

input	output
7 6 1237 0 0 1 1 1 1 1 C 3 3 1 C 1 6 4 I 1 -1235 C 1 7 3 I 4 1 C 2 5 4	1 1 1 4 6 4 1 1 7 20 30 1 4 5 2 0

Problem Source: Novosibirsk SU Contest. Petrozavodsk training camp, September 2007

```
/*
Timus 1560. Elementary Symmetric Functions
This problem can be solved using a modified interval tree. We build a complete
binary tree on top of the array A[1..N] in which every node stores 5 values:
S(0), S(1), S(2), S(3) and S(4) for the interval that is covered by its node
(i. e. all the array in the root, only 1 position in the leafs). We can compute
S(k) for a given node as sum i=0..k of S_left_child(i)*S_right_child(k-i).

For the type of queries that increase A[i], we compute again the value of all
the nodes from the leaf corresponding to A[i] to the root, which takes O(log n)
operations.

For the type of queries that ask the elementary symmetric polynomial of an
interval [L..R], we split the interval in disjoint intervals which correspond
to nodes in our tree and multiply their values.
*/
#include <iostream>
#include <vector>

using namespace std;

typedef long long ll;

ll p;
vector<vector<ll> > mem_get;
vector<vector<int> > query_get;
int cur_query;

ll get_elem_sym_poly(const vector<vector<ll> >& t, int a, int b, int nd, int i, int
    j, int k) {
  if (i <= a && b <= j) return t[nd][k];
  if (j <= a || b <= i) {
    if (k == 0) return 1;
    return 0;
  }

  ll &ret = mem_get[nd][k];
  if (query_get[nd][k] == cur_query) return ret;
  query_get[nd][k] = cur_query;
  ret = 0;
  for (int left = 0; left <= k; ++left) {
    ret += get_elem_sym_poly(t, a, (a+b)/2, 2*nd, i, j, left) * get_elem_sym_poly(t,
         (a+b)/2, b, 2*nd+1, i, j, k-left);
    ret %= p;
  }
  return ret;
}

void set_elem_sym_poly(vector<vector<ll> >& t, int x, ll v) {
  int nd = int(t.size())/2 + x;
  t[nd][1] = v;
  for (nd /= 2; nd >= 1; nd /= 2) {
    t[nd][0] = 1;
    for (int k = 1; k < int(t[nd].size()); ++k) {
      t[nd][k] = 0;
      for (int left = 0; left <= k; ++left) {
        int right = k-left;
        t[nd][k] += t[2*nd][left]*t[2*nd+1][right];
        t[nd][k] %= p;
      }
    }
```

```
  }
}

int main() {
  int n, m;
  cin >> n >> m >> p;
  vector<ll> a(n);
  for (int i = 0; i < n; ++i) {
    cin >> a[i];
    a[i] = ((a[i]%p)+p)%p;
  }
  int p2 = 2;
  for (; p2 < n;) p2 *= 2;
  p2 *= 2;
  vector<vector<ll> > t(p2, vector<ll>(5, 0));
  mem_get = vector<vector<ll> >(p2/2, vector<ll>(5));
  query_get = vector<vector<int> >(p2/2, vector<int>(5, -1));
  for (int i = int(t.size())/2; i < int(t.size()); ++i) {
    t[i][0] = 1;
  }
  for (int i = 0; i < n; ++i) {
    t[i+int(t.size())/2][1] = a[i];
  }
  for (int i = int(t.size())/2 - 1; i >= 1; --i) {
    t[i][0] = 1;
    for (int k = 1; k < int(t[i].size()); ++k) {
      t[i][k] = 0;
      for (int left = 0; left <= k; ++left) {
        int right = k-left;
        t[i][k] += t[2*i][left]*t[2*i+1][right];
        t[i][k] %= p;
      }
    }
  }
  cur_query = 0;
  for (int q = 0; q < m; ++q) {
    char c;
    cin >> c;
    if (c == 'I') {
      int index;
      ll delta;
      cin >> index >> delta;
      --index;
      a[index] += delta;
      a[index] = ((a[index]%p)+p)%p;
      set_elem_sym_poly(t, index, a[index]);
    }
    else {
      int left, right, mxk;
      cin >> left >> right >> mxk;
      --left;
      for (int k = 0; k <= mxk; ++k) {
        if (k > 0) cout << " ";
        cout << get_elem_sym_poly(t, 0, int(t.size())/2, 1, left, right, k);
      }
      cout << endl;
      ++cur_query;
    }
  }
}
```

1584. Pharaohs' Secrets

Time Limit: 1.0 second
Memory Limit: 64 MB

When programmer Alex was in Egypt, he not only swam in the Red Sea and went sightseeing, but also studied history. When Alex visited the place where an archeological dig of an ancient temple was carried out, an excavation worker complained to him that they had to drag very heavy statues from place to place every day. This was because some Egyptologist had read in an ancient papyrus that if the statues were arranged in a special order, then some ancient hiding-place would open. When the temple had been dug out, these statues had stood as soldiers, forming a rectangle. Some statues were identical, so there were several types of statues. They were to be arranged into a rectangle of the same dimensions on the same place with all rows and columns symmetric with respect to their middles. This meant that the statues standing in the same row or column at equal distances to its ends had to be of the same type.

Alex offered his help. He wants to find the way to transform the rectangle into a symmetric one by means of the minimal number of moves.

Input

The first line contains the dimensions of the rectangle n and m ($2 \le n, m \le 20$). These integers are even. Each of the next n lines contains m lowercase English letters. Each letter denotes the type of the statue that stands in the rectangle at this position.

Output

Output the minimal number of statues that should be moved in order to make a symmetric rectangle. It is guaranteed that this is possible.

Sample

input	output
4 4 abxa xyyb xyyx abba	2

Hint

The arrangement in the example can be transformed to a symmetric one in only two moves: first the statue of the type **x** from the upper row should be moved to the place in the rightmost column where there is the statue of the type **b**, and this statue then should moved to the place where the first statue stood. After all moves each place must be occupied by exactly one statue, but during the moving process there can be several statues at the same place.

Problem Author: Alex Gevak
Problem Source: ACM ICPC 2007–2008. NEERC. Eastern Subregion. Yekaterinburg, October 27, 2007

```
/*
Timus 1584. Pharaohs' Secrets
One cell and its symmetrical cells form a group of 4 cells which must contain the
same character. For each group, we must decide in an optimal way the character it
will contain at the end. Note that in case the frequency of one letter was not
divisible by 4, then no solution would be possible, so it will always be a multible
of 4. This problem can be reduced to a MaxFlow-MinCost problem.
Consider the following graph consisting of N*M/4 (number of groups) + 26 (number
of letters) + 2 (S source, T sink):
(1) There is an edge between S and each of the first N*M/4 nodes (representing the
groups). These edges have capacity 1, cost 0.
(2) There is an edge from every group to every letter. The capacity of each edge
is 1, and its cost is the number of cells in that group which have a different
letter than the one the edge is connected to.
(3) There is an edge from every letter to the sink T. The cost of these edges is
0, while its capacity is to the number of apparitions of that letter in the map,
divided by 4.
*/
#include <cstring>
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

#define PB push_back
#define NN 200
#define pot(u,v) (pi[u]-pi[v])

typedef pair<int, int> P;
typedef vector<int> Vi;
typedef vector<Vi> Mi;

const int INF = 1000000000;

char mapa[500][500];

Mi adj;
int deg[NN];
int padre[NN];

int w[NN][NN], cap[NN][NN], pi[NN], d[NN], f[NN][NN];
int N, m;
int flow, cost;

bool dijkstra(int s, int t) {
  memset(padre, -1, sizeof(padre));
  for (int i = 0; i < N; ++i) d[i] = INF;
  d[s] = 0;
  priority_queue<P> Q;
  Q.push(P(0, s));
  while (not Q.empty()) {
    int u = Q.top().second;
    int dist = -Q.top().first;
    Q.pop();
    if (dist != d[u]) continue;
    for (int i = 0; i < deg[u]; ++i) {
      int v = adj[u][i];
      if (f[u][v] >= 0 and cap[u][v] - f[u][v] > 0 and
          d[v] > d[u] + pot(u, v) + w[u][v]) {
        d[v] = d[u] + pot(u, v) + w[u][v];
        Q.push(P(-d[v], v));
```

```
        padre[v] = u;
      }
      else if (f[u][v] < 0 and d[v] > d[u] + pot(u, v) - w[v][u]) {
        d[v] = d[u] + pot(u, v) - w[v][u];
        Q.push(P(-d[v], v));
        padre[v] = u;
      }
    }
  }
  for (int i = 0; i < N; ++i) if (pi[i] < INF) pi[i] += d[i];
  return padre[t] >= 0;
}

void maxmin(int s, int t) {
  memset(f, 0, sizeof(f));
  memset(pi, 0, sizeof(pi));
  flow = cost = 0;
  while (dijkstra(s, t)) {
    int bot = INF;
    for (int v = t, u = padre[v]; u != -1; v = u, u = padre[u]) {
      if (f[u][v] >= 0) bot = min(cap[u][v] - f[u][v], bot);
      else bot = min(f[v][u], bot);
    }
    for (int v = t, u = padre[v]; u != -1; v = u, u = padre[u]) {
      if (f[u][v] >= 0) {
        f[u][v] += bot;
        f[v][u] -= bot;
        cost += w[u][v]*bot;
      }
      else {
        f[u][v] += bot;
        f[v][u] -= bot;
        cost -= w[v][u]*bot;
      }
    }
    flow += bot;
  }
}

int main() {
  int r, c;
  cin >> r >> c;

  Vi total(26, 0);
  for (int i = 0; i < r; ++i)
    for (int j = 0; j < c; ++j) {
      cin >> mapa[i][j];
      ++total[mapa[i][j] - 'a'];
    }

  int n = r*c/4;
  N = 2 + 26 + n;

  adj = Mi(N);
  for (int i = 0; i < r/2; ++i)
    for (int j = 0; j < c/2; ++j) {
      int t = i*(c/2) + j;
      Vi v(26, 4);
      --v[mapa[i][j] - 'a'];
      --v[mapa[i][c - j - 1] - 'a'];
      --v[mapa[r - i - 1][j] - 'a'];
```

```
      --v[mapa[r - i - 1][c - j - 1] - 'a'];
      for (int k = 0; k < 26; ++k) {
        cap[n + k][t] = 1;
        w[n + k][t] = v[k];
        adj[n + k].PB(t);
        adj[t].PB(n + k);
      }
    }

  for (int i = 0; i < 26; ++i) {
    cap[N - 2][n + i] = total[i]/4;
    adj[N - 2].PB(n + i);
    adj[n + i].PB(N - 2);
  }
  for (int i = 0; i < n; ++i) {
    cap[i][N - 1] = 1;
    adj[i].PB(N - 1);
    adj[N - 1].PB(i);
  }

  for (int i = 0; i < N; ++i) deg[i] = adj[i].size();

  maxmin(N - 2, N - 1);
  cout << cost << endl;
}
```

Timus 1585. Penguins

1585. Penguins

Time Limit: 1.0 second

Memory Limit: 64 MB

Programmer Denis has been dreaming of visiting Antarctica since his childhood. However, there are no regular flights to Antarctica from his city. That is why Denis has been studying the continent for the whole summer using a local cinema. Now he knows that there are several kinds of penguins:

- Emperor Penguins, which are fond of singing;
- Little Penguins, which enjoy dancing;
- Macaroni Penguins, which like to go surfing.

Unfortunately, it was not said in the cartoons which kind of penguins was largest in number. Petya decided to clarify this. He watched the cartoons once more and every time he saw a penguin he jotted down its kind in his notebook. Then he gave his notebook to you and asked you to determine the most numerous kind of penguins.

Input

The first line contains the number n of entries in the notebook ($1 \le n \le 1000$). In each of the next n lines, there is the name of a kind of penguins, which is one of the following: "Emperor Penguin," "Little Penguin," and "Macaroni Penguin."

Output

Output the most numerous kind of penguins. It is guaranteed that there is only one such kind.

Sample

input	output
7 Emperor Penguin Macaroni Penguin Little Penguin Emperor Penguin Macaroni Penguin Macaroni Penguin Little Penguin	Macaroni Penguin

Problem Author: Vladimir Yakovlev

Problem Source: ACM ICPC 2007–2008. NEERC. Eastern Subregion. Yekaterinburg, October 27, 2007

```
/*
Timus 1585. Penguins
This problem doesn't need any explanation... go for more interesting problems!
*/
#include <iostream>
#include <string>
using namespace std;

int main() {
  int n;
  cin >>  n;

  int emperor = 0, macaroni = 0, little = 0;

  for (int i = 0; i < n; ++i) {
    string a, b;
    cin >> a >> b;

    if (a[0] == 'E') ++emperor;
    else if (a[0] == 'M') ++macaroni;
    else ++little;
  }

  if (emperor >= macaroni and emperor >= little) {
    cout << "Emperor Penguin" << endl;
  }
  else if (macaroni >= emperor and macaroni >= little) {
    cout << "Macaroni Penguin" << endl;
  }
  else {
    cout << "Little Penguin" << endl;
  }
}
```

1586. Threeprime Numbers

Time Limit: 1.0 second
Memory Limit: 64 MB

Rest at the sea is wonderful! However, programmer Pasha became awfully bored of lying on a beach in Turkey; so bored that he decided to count the quantity of three-digit prime numbers. This turned out to be so interesting that he then started to study threeprime numbers. Pasha calls an integer a threeprime number if any three consecutive digits of this integer form a three-digit prime number. Pasha had already started working on the theory of the divine origin of such numbers when some vandals poured water on Pasha and cried some incomprehensible words like "Sonnenstich!", "Colpo di sole!", and "Coup de soleil!"

You are to continue Pasha's work and find out how often (or rare) threeprime numbers are.

Input

The input contains an integer n ($3 \le n \le 10000$).

Output

Output the quantity of n-digit threeprime numbers calculated modulo $10^9 + 9$.

Sample

input	output
4	204

Problem Author: Denis Musin
Problem Source: ACM ICPC 2007–2008. NEERC. Eastern Subregion. Yekaterinburg, October 27, 2007

```
/*
Timus 1586. Threeprime Numbers
This problem can be solved with Dynamic Programming.
In the following code, d[i][j] (0<=i<100) is the number of
different (j+2)-digit threeprime numbers which its two first
digits are i/10 and i%10.
*/
#include <iostream>
using namespace std;

const int MOD = 1000000009;

int prime[1000];
int dp[100][10010];

bool isprime(int n) {
  if (n < 2) return false;
  for (int i = 2; i*i <= n; ++i)
    if (n%i == 0) return false;
  return true;
}

int main() {
  for (int i = 100; i < 1000; ++i)
    if (isprime(i)) prime[i] = 1;

  int n;
  cin >> n;

  for (int i = 0; i < 100; ++i) dp[i][0] = 1;
  for (int j = 1; j <= n - 2; ++j) {
    for (int i = 0; i < 100; ++i) {
      for (int d = 0; d < 10; ++d) {
        int t = 10*i + d;
        if (prime[t]) dp[i][j] = (dp[i][j] + dp[t%100][j - 1])%MOD;
      }
    }
  }

  int res = 0;
  for (int i = 10; i < 100; ++i)
    res = (res + dp[i][n - 2])%MOD;
  cout << res << endl;
}
```

1588. Jamaica

Time Limit: 1.0 second

Memory Limit: 64 MB

Programmer Andrey is very lucky. The Governor-General of Jamaica invited him to visit this wonderful island. The whole trip, including plenty of entertainment, will be completely free for Andrey. He will only have to help the Jamaican Ministry of Transport and write a small program.

The point is that the Jamaican government decided to construct a new network of expressways, in order to boost the economic growth of the country. Each two cities are to be connected by a road going along a straight line. One road may connect several cities if they lie on the same straight line. Jamaican economists are sure that the new network will minimize transportation expenses. In order to estimate the cost of the project, it is required to determine the total length of the roads to be constructed, and this is the job for Andrey.

Input

The first line contains the number n of cities in Jamaica ($1 \le n \le 300$). The next n lines contain the coordinates of the cities. In each of these lines, there are two integers x_i and y_i separated by a space ($0 \le x_i, y_i \le 10000$). There are no cities with coinciding coordinates.

Output

Output the total length of the roads rounded to the nearest integer.

Sample

input	output
4 0 0 0 100 100 0 50 50	412

Problem Author: Andrey Demidov

Problem Source: ACM ICPC 2007–2008. NEERC. Eastern Subregion. Yekaterinburg, October 27, 2007

```
/*
Timus 1588. Jamaica
Every pair of cities have to be directly connected, but we only count the roads
that are aligned as one. In order to do this we can store the roads grouped by
their slope and only insert road segments that are disjoint with all the
previously inserted road segments.
*/
#include <cmath>
#include <iostream>
#include <map>
#include <vector>

#define X first
#define Y second
#define FR first
#define SC second

using namespace std;

typedef pair<int, int> PII;

int gcd(int a, int b) {
  if (b == 0) return a;
  return gcd(b, a%b);
}

double dist(PII a, PII b) {
  return sqrt(double((a.X-b.X)*(a.X-b.X)+(a.Y-b.Y)*(a.Y-b.Y)));
}

PII get_slope(PII a, PII b) {
  if (a == b) return PII(0, 0);
  if (a > b) swap(a, b);
  PII v(b.X-a.X, b.Y-a.Y);
  int div = gcd(abs(v.X), abs(v.Y));
  v.X /= div;
  v.Y /= div;
  return v;
}

bool same_line(PII slope, PII a, PII b) {
  if (a == b) return true;
  return get_slope(a, b) == slope;
}

int main() {
  cout.setf(ios::fixed);
  cout.precision(0);

  int n;
  cin >> n;
  vector<PII> cities(n);
  for (int i = 0; i < n; ++i) {
    cin >> cities[i].X >> cities[i].Y;
  }

  map<PII, vector<pair<PII, PII> > > lines;
  for (int i = 0; i < n; ++i) {
    for (int j = i+1; j < n; ++j) {
      PII slope = get_slope(cities[i], cities[j]);
      vector<pair<PII, PII> >& v = lines[slope];
```

```
      int ind = -1;
      for (int k = 0; k < int(v.size()) && ind == -1; ++k) {
        if (same_line(slope, cities[i], v[k].FR)) {
          ind = k;
        }
      }
      if (ind == -1) {
        v.push_back(pair<PII, PII>(min(cities[i], cities[j]), max(cities[i], cities[
            j])));
      }
      else {
        v[ind].FR = min(v[ind].FR, min(cities[i], cities[j]));
        v[ind].SC = max(v[ind].SC, max(cities[i], cities[j]));
      }
    }
  }

  double ans = 0.0;
  for (map<PII, vector<pair<PII, PII> > >::iterator it = lines.begin(); it != lines.
      end(); ++it) {
    vector<pair<PII, PII> >& v = it->second;
    for (int i = 0; i < int(v.size()); ++i) {
      ans += dist(v[i].FR, v[i].SC);
    }
  }
  cout << ans << endl;
}
```

1590. Bacon's Cipher

Time Limit: 1.0 second
Memory Limit: 64 MB

Programmer Vasya was down on his luck. Instead of a vacation, he was sent to a scientific conference.

"It is necessary to increase your competence," his boss said, "it's an important conference on cryptography, and it's held in France, where they used encryption in the days of de Richelieu and cracked codes in the days of Viete."

One of the talks at the conference was about the attempts to solve Bacon's ciphers. The speaker proposed a hypothesis that the key to Bacon's secrets could be found if all possible substrings of Bacon's works were analyzed.

"But there are too many of them!" Vasya expressed his astonishment.

"Not as many as you think," the speaker answered, "count them all and you'll see it yourself."

That evening Vasya found on the Web the complete set of Bacon's works. He wrote a program that converted the texts into one long string by removing all linebreaks, spaces, and punctuation marks. And now Vasya is confused because he doesn't know how to calculate the number of different substrings of this string.

Input

You are given a nonempty string consisting of lowercase English letters. The string is no longer than 5000 symbols.

Output

Output the number of different substrings of this string.

Sample

input	output
aaba	8

Problem Author: Alex Gevak
Problem Source: ACM ICPC 2007–2008. NEERC. Eastern Subregion. Yekaterinburg, October 27, 2007

```
/*
Timus 1590. Bacon's Cipher
Although this problem can be solved in linear time using a Suffix Tree
(which is the solution implemented here), a simplier to code algorithm
running in O(N^2) (for example, computing a hash for every substring)
should also solve the problem.
Construct the Suffix Tree of the string given in the input in O(N) and
then count the number of different paths from the root to any other
node (implicit or explicit (the Suffix Tree's linear version has
implicit nodes along the edges, and can count them without visiting
them, just visiting explicit nodes)).
*/
#include <iostream>
#include <map>
#include <string>
#include <utility>
using namespace std;

#define X first
#define Y second

typedef int ttype;
typedef pair<int, int> P;
typedef pair<P, int> TR;
typedef map<ttype, TR> MAP;
typedef MAP::iterator Mit;
typedef pair<int, MAP> ND;

const int MAXN = 5000;

int N, M;
ND tree[2*MAXN + 17];
ttype T[MAXN + 17];

int nextsep[MAXN + 17];

bool test_and_split(int s, int k, int p, ttype t, int& r) {
  r = s;
  if (p < k) return tree[s].Y.find(t) != tree[s].Y.end();
  TR tr = tree[s].Y[T[k]];
  int sp = tr.Y, kp = tr.X.X, pp = tr.X.Y;
  if (t == T[kp + p - k + 1]) return true;
  r = M++;
  tree[s].Y[T[kp]] = TR(P(kp, kp + p - k), r);
  tree[r].Y[T[kp + p - k + 1]] = TR(P(kp + p - k + 1, pp), sp);
  return false;
}

void canonize(int s, int k, int p, int& ss, int& kk) {
  if (p < k) { ss = s; kk = k; return; }
  TR tr = tree[s].Y[T[k]];
  int sp = tr.Y, kp = tr.X.X, pp = tr.X.Y;
  while (pp - kp <= p - k) {
    k += pp - kp + 1; s = sp;
    if (k <= p) {
      tr = tree[s].Y[T[k]];
      sp = tr.Y; kp = tr.X.X; pp = tr.X.Y;
    }
  }
  ss = s; kk = k;
}
```

```
void update(int& s, int& k, int i) {
  int oldr = 0, r;
  while (not test_and_split(s, k, i - 1, T[i], r)) {
    tree[r].Y[T[i]] = TR(P(i, N - 1), M++);
    if (oldr != 0) tree[oldr].X = r;
    oldr = r;
    canonize(tree[s].X, k, i - 1, s, k);
  }
  if (oldr != 0) tree[oldr].X = s;
}

void suffixtree() {
  for (int i = 0; i < M; ++i) { tree[i].X = 0; tree[i].Y.clear(); }
  M = 2;
  for (int i = 0; i < N; ++i) tree[1].Y[T[i]] = TR(P(i, i), 0);
  tree[0].X = 1;
  int s = 0, k = 0;
  for (int i = 0; i < N; ++i) {
    update(s, k, i);
    canonize(s, k, i, s, k);
  }
}

void escriu(int n, int p = 1) {
  for (Mit it = tree[n].Y.begin(); it != tree[n].Y.end(); ++it) {
    cout << string(4*p, '-') << "> ";
    for (int i = it->Y.X.X; i <= it->Y.X.Y; ++i) {
      if (T[i] < 0) cout << "(" << -T[i] - 1 << ")";
      else cout << char(T[i]);
    }
    cout << endl;
    escriu(it->Y.Y, p + 1);
  }
}

int fun(int n) {
  int res = 0;
  for (Mit it = tree[n].Y.begin(); it != tree[n].Y.end(); ++it) {
    res += it->Y.X.Y - it->Y.X.X + 1;
    res += fun(it->Y.Y);
  }
  return res;
}

int main() {
  string s;
  cin >> s;

  N = s.size();
  for (int i = 0; i < N; ++i) T[i] = s[i];

  suffixtree();
  cout << fun(0) << endl;
}
```

1591. Abstract Thinking

Time Limit: 1.0 second
Memory Limit: 64 MB

This summer programmer Dima wanted to take a rest from programming because he felt that the level of abstraction his profession required was driving him mad. He decided to go to Greece. But he forgot that Greece was the homeland of geometry where famous Euclid lived and worked. In geometry, instead of real figures, abstract notions are studied, and the proofs are based not on intuition, but on axioms and formal definitions. Even shepherds in Greece have well-developed abstract thinking skills.

For example, consider the following problem, which will show the level of your abstract thinking abilities. Imagine a circle, then put mentally n points on its periphery at equal distances. After that connect (again mentally!) these points pairwise by straight segments (of course, you remember that such segments are called chords).

Now consider any three different chords from this set that intersect pairwise. If at least one of their intersection points lies inside the circle, we will call the figure formed by these chords an interesting triangle. If you can count the number of interesting triangles correctly, then you can go to Greece and not be ashamed of yourself there.

Input

You are given the number n of points on the periphery of a circle ($3 \le n \le 2000$).

Output

Output the number of interesting triangles.

Samples

input	output
4	4
5	25

Problem Author: Denis Musin
Problem Source: ACM ICPC 2007–2008. NEERC. Eastern Subregion. Yekaterinburg, October 27, 2007

```cpp
/*
Timus 1591. Abstract Thinking
There are three kinds of triangle:
(1) Triangles which have exactly one vertice strictly inside the circle:
  Let's call A and B the vertices lying in the circle's perimeter, and C the
  other one. Choose the location of A (N possibilities), then choose the number
  of points 'a' that are between A and B (0<=a<=N-2). Then, choose two of the
  'a' points, where the two chords whose intersection is C will intersect with
  the circle.
(2) Triangles with 2 vertices inside the circle:
  Let's call A and B the vertices lying strictly inside the circle, and C to
  the other one. Choose the location of C (N possibilities), then choose the
  number of points 'a' between C and the intersection of the circle with the
  chord A-C. Then, choose the intersection point of the chord A-B with the
  circle in one of the two sides in which A-C divides the circle) ('a'
  possibilities). And finally, choose the intersection points of A-B and B-C
  with the circle in the other side (ways of choosing two points among N-a-2).
(3) All the vertices are strictly inside the circle: Ways of choosing 6 points
  out of N. Let 1,2,3,4,5,6 be the 6 points ordered clockwise, join them as
  follows to get the triangle: 1-4, 2-5, 3-6.
*/
#include <iostream>
using namespace std;

typedef long long ll;

ll cn[2010][2010];

int main() {
  for (int n = 1; n <= 2000; ++n) {
    cn[n][0] = cn[n][n] = 1;
    for (int k = 1; k < n; ++k)
      cn[n][k] = cn[n - 1][k - 1] + cn[n - 1][k];
  }

  int n;
  cin >> n;

  ll res = 0;
  for (int a = 0; a <= n - 2; ++a) {
    int b = n - 2 - a;
    res += ll(a)*ll(a - 1)/2;
    res += ll(b)*ll(a)*ll(a - 1)/2;
  }
  res *= n;
  res += cn[n][6];

  cout << res << endl;
}
```

1888. Pilot Work Experience

Time Limit: 1.0 second
Memory Limit: 64 MB

Leonid had n Oceanic Airlines flights during his business trip. He studied the latest issue of the monthly on-board magazine of this company from cover to cover. In particular, he learned about the rules of forming an airplane crew. It turned out that the work experience of the captain (i.e., the number of complete years of work in civil aviation) was always greater by exactly one than the work experience of the second pilot.

The Oceanic Airlines company does not disclose information on the work experience of their pilots. Leonid is interested in the largest possible difference between the work experiences of pilots on the flights he had. He has written the names of the two pilots on each flight but couldn't remember who was the captain and who was the second pilot. Leonid assumes that the work experience of each pilot is in the range from 1 to 50 years and that the work experiences didn't change in the period between his first and last flights with Oceanic Airlines.

Help Leonid use the available information to find out the maximum possible difference in the work experiences of pilots on the flights he had.

Input

The first line contains integers n and p, which are the number of flights Leonid had and the number of pilots flying the planes on these flights ($2 \le n \le 1\ 000$; $2 \le p \le 50$). The pilots are numbered from 1 to p. In the ith of the following n lines you are given two different integers denoting the pilots of the ith flight.

Output

In the first line output the maximum possible difference in the work experiences of the pilots. In the second line output p integers. The ith integer must be the work experience of the ith pilot. If there are several possible answers, output any of them. If Leonid is wrong in his assumptions or his data are incorrect, output “-1”.

Samples

input	output
4 4 1 2 3 1 2 4 3 4	2 1 2 2 3
3 3 1 2 2 3 1 3	-1

Problem Author: Magaz Asanov
Problem Source: NEERC 2011, Eastern subregional contest

```
/*
Timus 1888. Pilot Work Experience
Let's consider the graph where each node corresponds to a pilot, and there is
an edge between two nodes if the corresponding pilots have been in the same
flight. The answer will be -1 if and only if this graph is not bicolorable:
it has to be bicolorable because of the parity of the years of experience, and
if it is bicolorable there is always a solution in which white nodes have a
value of x (where 1 <= x < 50) and black nodes have a value of x+1.

Notice that if the graph has more than one connex component, the maximum
difference between two nodes is the highest possible, 50-1 = 49, because we can
have some node with a value of 1 in one component and some node with a value of
50 in the other component.

If there is one single connex component, we want to split the nodes in the
maximum number of subsets G_1, G_2, ..., G_n such that nodes in a subset G_k
only have edges going to subsets G_{k-1} and G_{k+1}. We can generate maximal
partitions of this kind by choosing a starting node and then placing two nodes
in the same group G_k if and only if they are at the same distance from the
starting node. We can see this way of generating partitions as an "unfolding" of
the bipartite graph that is always possible int more than 2 layers.

The maximum number of subsets G_1, G_2, ..., G_n will be achieved when the
starting node is an endpoint of a diameter of the graph, and therefore the
answer will be min(49, diameter of the graph).

Because the problem asks for a possible set of values of the nodes, in order to
make things simple the following solution solves for each component, starting
with a value of 1 and going up for half of the components and with a value of 50
and going down for the remaining components.
*/
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

const int INF = 1000000000;

bool bicolor(vector<vector<int> >& g, vector<int>& color, int nd, int col) {
  if (color[nd] >= 0) return color[nd] == col;
  color[nd] = col;
  for (int i = 0; i < int(g[nd].size()); ++i) {
    if (!bicolor(g, color, g[nd][i], 1-col)) return false;
  }
  return true;
}

void dfs_comp(vector<vector<int> >& g, vector<int>& comp, int nd, int cmp) {
  if (comp[nd] >= 0) return;
  comp[nd] = cmp;
  for (int i = 0; i < int(g[nd].size()); ++i) {
    dfs_comp(g, comp, g[nd][i], cmp);
  }
}

vector<int> bfs(vector<vector<int> >& g, int src) {
  queue<int> q;
  vector<int> dist(g.size(), INF);
  q.push(src);
  dist[src] = 0;
```

```
  for (; !q.empty(); q.pop()) {
    int cur = q.front();
    for (int i = 0; i < int(g[cur].size()); ++i) {
      if (dist[cur]+1 < dist[g[cur][i]]) {
        dist[g[cur][i]] = dist[cur]+1;
        q.push(g[cur][i]);
      }
    }
  }
  return dist;
}

int main() {
  int n, p;
  cin >> n >> p;
  vector<vector<int> > g(p);
  for (int i = 0; i < n; ++i) {
    int a, b;
    cin >> a >> b;
    --a;
    --b;
    g[a].push_back(b);
    g[b].push_back(a);
  }
  vector<int> color(p, -1);
  bool bicol = true;
  for (int i = 0; i < p && bicol; ++i) if (color[i] < 0) {
    if (!bicolor(g, color, i, 0)) {
      bicol = false;
    }
  }
  if (!bicol) {
    cout << -1 << endl;
    return 0;
  }
  vector<int> comp(p, -1);
  int ncomp = 0;
  for (int i = 0; i < p; ++i) {
    if (comp[i] < 0) {
      dfs_comp(g, comp, i, ncomp);
      ++ncomp;
    }
  }

  vector<int> num(p, -1);
  // solve for each component
  for (int i = 0; i < ncomp; ++i) {
    int mxnode = -1, mxdiam = -1;
    for (int j = 0; j < p; ++j) if (comp[j] == i) {
      vector<int> dist = bfs(g, j);
      int mxdist = -1;
      for (int k = 0; k < p; ++k) if (comp[k] == i) {
        if (dist[k] > mxdist) mxdist = dist[k];
      }
      if (mxdist > mxdiam) {
        mxnode = j;
        mxdiam = mxdist;
      }
    }
    vector<int> dist = bfs(g, mxnode);
    for (int j = 0; j < p; ++j) if (comp[j] == i) {
```

```
      if (dist[j]+1 <= 50) num[j] = dist[j]+1;
      else if ((dist[j]+1)%2 == 0) num[j] = 50;
      else num[j] = 49;
      if (i%2 == 1) {
        num[j] = 51-num[j];
      }
    }
  }
  int mn = 51, mx = -1;
  for (int i = 0; i < p; ++i) {
    if (num[i] < mn) mn = num[i];
    if (num[i] > mx) mx = num[i];
  }
  cout << mx-mn << endl;
  for (int i = 0; i < p; ++i) {
    if (i > 0) cout << " ";
    cout << num[i];
  }
  cout << endl;
}
```

Timus 1891. Language Ocean

1891. Language Ocean

Time Limit: 1.0 second
Memory Limit: 64 MB

Engineers from the Oceanic Airlines company have proposed to write software for airplane equipment using the new, very simple, and efficient programming language Ocean.

This language has only three data types: int, real, and string. Each program starts with a list of function headers in which all the functions used in the program must be mentioned. A function header has the form

- <function name>(<type of arg 1>, ..., <type of arg k>) : <type of the result>

A function may have no arguments. The names of different functions may coincide.

The text of the main program is a list of variable declarations together with their initializations. Each line of the program has the form

- <variable type> <variable name> = <function name>(<arg 1>, ..., <arg k>)

For a programmer's convenience, the variable type can be replaced with the keyword auto, which means that the variable type is defined by the type of the value returned by the function. When a function is called, the names of previously declared variables are specified as its arguments. If there are several functions with the specified name, then a function will be called in which the number of arguments and their types correspond to the function call. The name of a variable may coincide with the name of some function.

Employees of Oceanic Airlines ask you to write an interpreter of the Ocean language. The interpreter must check each line of the main program searching for the following errors (in the following order):

1. “Double declaration” — a variable with this name has already been declared;
2. “Unknown variable” — the name of one of the arguments hasn't been declared in previous lines;
3. “No such function” — there is no function with this name and a suitable list of arguments;
4. “Invalid initialization” — the type of the returned value does not correspond to the type of the variable.

Input

The first line contains the number f of function headers ($1 \le f \le 100$). The headers are given in the following f lines. Any two functions differ either by their names or by the lists of their arguments. In the following line you are given the number n of lines in the main program ($1 \le n \le 100$). Each of the following n lines contains a variable declaration and its initialization. Each function header and each function call contains at most 10 arguments. The names of variables and functions have lengths from 1 to 20 symbols. Names of the variables consist of lowercase English letters, whereas names of the functions may also contain uppercase English letters. The names of functions differing by letter cases only are considered to be different. The strings “int”, “real”, “string”, and “auto” cannot be names of variables or functions. Spacing is as in the input data samples.

Output

If there are no errors in the program, output the list of all variables declared as “auto” and their types. The variables should be given in the order in which they are declared in the program. If there are errors in the program, output the number of the line containing the first error and the type of the error. The format of the output should correspond to the samples.

Samples

input	output
2 ReadInt() : int IntToReal(int) : real 3 auto a = ReadInt() auto b = IntToReal(a) int c = ReadInt()	a : int b : real
1 ReadInt() : int 2 int x = ReadInt() int x = ReadInt()	Error on line 2: Double declaration
3 ReadInt() : int ReadReal() : real SumRealInt(real, int) : real 4 auto x = ReadInt() auto y = ReadReal() real z = SumRealInt(y, x) real q = SumRealInt(x, y)	Error on line 4: No such function

Problem Author: Alex Samsonov

```cpp
/*
Timus 1891. Language Ocean
Functions can be stored in a map where the key is
(<function name>, <type of arg 1>, ..., <type of arg k>)
and the value is <type of the result>.
The type of each variable can be kept track of using another map where the key
is <variable name> and the value is <type of variable>.

Using and updating these structures properly it is easy to find errors in input
and keep track of the type of every "auto" variable.
*/
#include <iostream>
#include <map>
#include <sstream>
#include <string>
#include <vector>

using namespace std;

vector<string> parse_function(string fun) {
  for (int i = 0; i < int(fun.size()); ++i) {
    if (fun[i] == '(' || fun[i] == ')' || fun[i] == ',') {
      fun[i] = ' ';
    }
  }
  vector<string> pfun;
  istringstream iss(fun);
  for (string s; iss >> s;) {
    pfun.push_back(s);
  }
  return pfun;
}

int main() {
  int f;
  cin >> f;
  map<vector<string>, string> func;
  for (int i = 0; i < f; ++i) {
    string fun = "", ret;
    for (string s; cin >> s && s != ":";) {
      fun += s;
    }
    cin >> ret;
    vector<string> pfun = parse_function(fun);
    func[pfun] = ret;
  }
  map<string, string> type;
  vector<string> ans;
  int n;
  cin >> n;
  for (int i = 0; i < n; ++i) {
    string typ, var, fun;
    cin >> typ >> var >> fun;
    getline(cin, fun);

    // double declaration
    if (type.count(var) > 0) {
      cout << "Error on line " << i+1 << ": Double declaration" << endl;
      return 0;
    }
```

```
    vector<string> pfun = parse_function(fun);
    // unkown variable
    for (int j = 1; j < int(pfun.size()); ++j) {
      if (type.count(pfun[j]) == 0) {
        cout << "Error on line " << i+1 << ": Unknown variable" << endl;
        return 0;
      }
    }

    // no such function
    for (int j = 1; j < int(pfun.size()); ++j) {
      pfun[j] = type[pfun[j]];
    }

    if (func.count(pfun) == 0) {
      cout << "Error on line " << i+1 << ": No such function" << endl;
      return 0;
    }

    // invalid initialization
    if (typ != "auto" && func[pfun] != typ) {
      cout << "Error on line " << i+1 << ": Invalid initialization" << endl;
      return 0;
    }
    if (typ == "auto") {
      ans.push_back(var);
      type[var] = func[pfun];
    }
    else {
      type[var] = typ;
    }
  }
  for (int i = 0; i < int(ans.size()); ++i) {
    cout << ans[i] << " : " << type[ans[i]] << endl;
  }
}
```

1893. A380

Time Limit: 1.0 second
Memory Limit: 64 MB

There was no limit to Jack's joy! He managed to buy on the Internet tickets to the ICPC semifinal contest, which was to be held very soon in the mysterious overseas city of Saint Petersburg. Now he was going to have a transoceanic flight in the world largest passenger aircraft Airbus A380.

Jack started studying the aircraft seating chart in Wikipedia, so that he would be able to ask for a nice window seat at check-in. Or maybe he would ask for an aisle seat—he hadn't decided yet.

Airbus A380 has two decks with passenger seats. The upper deck is for premium and business class passengers. The premium class section embraces the first and second rows. Each row contains four seats identified by letters from A to D. The aisles in this section are between the first and second seats and between the third and fourth seats of each row. The rows from the third to the twentieth are for business class passengers. There are six seats in each row, and they are identified by letters from A to F. The aisles are between the second and third and between the fourth and fifth seats of each row.

The lower deck is reserved for economy class passengers. The rows are numbered from 21 to 65. Each row contains ten seats identified by letters from A to K (the letter I is omitted). The aisles are between the third and fourth seats and between the seventh and eighth seats of each row.

Help Jack determine if a seat is next to the window or next to the aisle given the seat designation.

Input

The only line contains a seat designation: the row number and the letter identifying the position of the seat in the row.

Output

If the seat is next to the window, output “window”. Otherwise, if the seat is next to the aisle, output “aisle”. If neither is true, output “neither”.

Samples

input	output
3C	aisle
64A	window
21F	neither

Problem Author: Denis Dublennykh
Problem Source: NEERC 2011, Eastern subregional contest

```
/*
Timus 1893. A380
Compute the column number of the seat. Then consider the 3 kinds of row and
check what kind of seat corresponds to the seat column in its row.
*/
#include <iostream>
#include <string>

using namespace std;

int main() {
  int row;
  char seat;
  cin >> row >> seat;
  int col = 1+int(seat-'A');
  if (col > 9) --col; // letter I is omitted
  string ans = "";
  if (row <= 2) {
    if (col == 1 || col == 4) ans = "window";
    else ans = "aisle";
  }
  else if (row <= 20) {
    if (col == 1 || col == 6) ans = "window";
    else ans = "aisle";
  }
  else {
    if (col == 1 || col == 10) ans = "window";
    else if (col == 3 || col == 4 || col == 7 || col == 8) ans = "aisle";
    else ans = "neither";
  }
  cout << ans << endl;
}
```

5 Anàlisi Econòmica

Aquest projecte ha estat desenvolupat per 2 estudiants treballant a temps complet i 1 professor treballant 15 hores per setmana, al llarg de 18 setmanes, entre Febrer i Juny de 2012.

Salari	Hores	€/Hora	Total
Professor	270	34	9180
Estudiant	1440	5	7200
Total	1710	-	16380

Taula 1: Costs dels salaris

Al fer servir software lliure per desenvolupar els programes, no hi ha cap cost adicional degut al software utilitzat. L'ús dels PCs, connexió a internet, accés a llibres i material docent es considera un cost fix de la Universitat Politècnica de Catalunya, i per tant tampoc afecta al preu d'aquest projecte. El cost total és, per tant, 16380€.

6 Conclusions

6.1 Resultat a la final de l'ICPC

En la final mundial de l'ICPC 2012, celebrada a Varsòvia, l'equip de la Universitat Politècnica de Catalunya ha quedat classificat en la posició número 36 d'un total de 112 equips finalistes. Tenint en compte que en total a l'ICPC 2012 han participat uns 25000 estudiants de més de 2200 universitats diferents, es pot qualificar aquest resultat d'excel·lent. La resolució dels problemes presentats en aquest document ens ha fet desenvolupar tècniques noves i guanyar agilitat de pensament i programació, sent per tant un dels factors que ha fet possible aquest bon resultat.

6.2 Visió de futur

Globalment pensem que resoldre problemes d'aquest tipus ha estat molt productiu de cara a les competicions de programació. Tot i així, com a possible millora del mètode d'aprenentatge per a futurs participants de l'ICPC o per a nosaltres mateixos en un futur, proposem donar més importància a simular les condicions d'una competició real. Una possible manera d'aconseguir això és participant freqüentment en concursos online com ara TopCoder o Codeforces, que tot i ser diferents a l'ICPC perquè són competicions individuals, proporcionen una sèrie de condicions que són factors determinants en una competició presencial:

- Els problemes tenen una dificultat prou alta com perquè només els millors del món siguin capaços de resoldre'ls tots durant la duració del concurs.
- Hi ha classificació en directe, de manera que el participant té informació sobre quins problemes ha resolt cadascú.
- El participant ha de resoldre els problemes en el mínim temps possible, i per tant està treballant sota una certa pressió.

Més enllà de les competicions de programació, l'habilitat per resoldre problemes de tipus algorísmic o matemàtic mitjançant la programació pot ser útil en diversos camps científics i també en el desenvolupament de noves tecnologies. Un clar exemple d'això és que les grans empreses d'avui dia es troben amb problemes similars, com és el cas de Google, que en els darrers anys ha contractat a un gran nombre de programadors dels que han destacat en competicions de l'estil de l'ICPC.

7 Bibliografia

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

2. Terence Tao. *Solving Mathematical Problems*. Oxford University Press, 2006.

3. Dan Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

4. Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. *Algorithms*. McGrawHill, 2006.

5. Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., 1998.

6. *TopCoder Algorithm Tutorials*. URL http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=alg_index