OREGON STATE UNIVERSITY COMPUTER SCIENCE DEPARTMENT

A Brief Introduction to Analysis of Algorithms

Paul Cull
Department of Computer Science
Oregon State University
Corvallis, Oregon 97331

85-20-3

A BRIEF INTRODUCTION TO ANALYSIS OF ALGORITHMS

Paul Cull
Department of Computer Science
Oregon State University
Corvallis, OR 97331

1. INTRODUCTION

1.1 What Is An Algorithm?

An algorithm is a procedure which solves a problem and is suitable for implementation as a program for a digital computer. This informal definition makes two important points. First, an algorithm solves a problem. A computer program may not solve a problem. For example, there are computer programs which never terminate. It would be very difficult to say that such a program does anything, let alone solve a problem. Second, each step of the algorithm should be well defined and be representable, at least in principle, by a program. For example, s:= P/q would not be well defined if q could be 0; "find the smallest x such that P(x) is true" would not be well defined if no such x existed, or if x were allowed to range over the positive and negative integers and P(x) were true for all negative x. "Add a dash of salt" would not be an acceptable instruction in a computer program unless one had created a model world and a mapping which would translate the instruction into the modification of some location in a computer's memory.

For our purposes, and for many purposes, the above informal definition of algorithm is sufficient, but we should point out that a formal definition of algorithm can be created. This has been done by Turing, Markoff, and others [Rogers, 1967]. The satisfying thing about all these formal definitions is that they define the same class of algorithms. If we have an algorithm in one of these senses, then there is an equivalent algorithm in any one of the other senses. Further, the existence of a formal definition means that if we run into difficulties with the informal definition, we could use the formal definition to unravel the difficulty.

1.2 What Is Analysis Of Algorithms?

The task of analysis of algorithms is three-fold:

(a) To produce provably correct algorithms, that is, algorithms which not only solve the problem they are designed to solve, but which also can be demonstrated to solve the problem;

(b) To compare algorithms for a problem with respect to various measures of resources (e.g., time and space), so that one can say when one algorithm is better than another;

(c) To find, if possible, the best algorithm for a problem with respect to a particular measure of resource usage.

Each of these tasks will be examined in more detail later, but first I want to say a few words about where analysis of algorithms fits in the worlds of mathematics and computer science. Mathematics has two great branches, pure mathematics and applied mathematics. Mathematics also has two major techniques, proof and computation. It would be tempting to suggest that pure mathematics uses proof and applied mathematics uses computation, but both pure and applied mathematics use both proof and computation. The difference between the two branches is the world they work in. Pure mathematics works in an abstract world. It needs no reference to a real world. Applied mathematics, on the other hand, assumes there is a real world and then constructs an abstract world. The mapping or correspondence between entities of the real and abstract worlds form an essential part of applied mathematics.

Where does computer science fit in? Computer science is the science which deals with computation and computing devices. In its theoretical branches, computer science uses the mathematical techniques of proof and computation, while in its more practical branches, computer science uses engineering techniques and experimental techniques.

Analysis of algorithms, like applied mathematics, assumes a real world. This

real world contains actual computers and actual computer programs. From this real world, an abstract world of abstract computers and abstract programs is constructed. This construction is usually informal and exact definitions of abstract entities are often not stated, but a number of real world limitations disappear in the abstract world. For example, real computers have a fixed finite memory size and they have an upper bound on the size of numbers which can be represented. In the abstract world, these limitations do not exist. Finite but unbounded memories are assumed to exist. No bound on the size of numbers exists.

Again like applied mathematics, analysis of algorithms uses the techniques of proof and computation to deal with the entities in its abstract world, and like applied mathematics, one must be cautious in applying results from the abstract world to the real world. There are examples of algorithms which would work very quickly if arbitrarily large numbers could be used, but implementing these algorithms on real computers results in algorithms which are much slower. As another example, there are algorithms which can be shown to be quicker than other algorithms, but only if the input is astronomically large. These examples make perfect sense in the abstract world, but have little or no relevance for the real world.

Analysis of algorithms is the applied mathematics of computer science. Whether it should be called mathematics or computer science usually depends on who is doing it. If a mathematician does analysis of algorithms, it may be called mathematics, but if a computer scientist does analysis of algorithms, it is usually called computer science.

1.2.1 Proofs of correctness

The first of the three tasks of analysis of algorithms is to produce provably correct algorithms. This task suggests that we need a methodology to both produce

algorithms and to produce proofs of the correctness of these algorithms. Later we will consider two design strategies: (a) to solve a problem, break it into smaller problems of the same kind; (b) to solve a problem, search an answer space until you find the answer to the problem you have.

Proving correctness of algorithms based on a search strategy is usually relatively easy: Show that the answer to your problem is in the space; show that your algorithm searches the whole space, or that if your algorithm decides not to search a portion of the space, then your algorithm has established that your answer cannot lie in the portion not to be searched.

Proving correctness of algorithms based on breaking a problem into smaller problems can most readily be done by using mathematical induction. You establish that your algorithm correctly solves all small enough cases of the problem. Your algorithm will usually have a section which deals with these small cases, but particularly when the small cases may be of size 0 and no action by the algorithm is required, the section for small cases may not exist. The other section of your algorithm will deal with larger cases by breaking them into smaller cases and combining the solutions to the smaller cases to give the solution to the larger problem. You then have to show that if the smaller problems are solved correctly, then your algorithm combines these solutions correctly to yield the solution to the larger problem.

The proof procedure just outlined assumes that your algorithm is recursive, that is, to solve large problems the algorithm calls itself to solve smaller problems, but proofs by mathematical induction are not limited to recursive algorithms. Inductive proofs are also natural when dealing with iterative algorithms which are based on loops, for example, WHILE loops, FOR loops, DO loops, or REPEAT...UNTIL loops. Inductive proofs of correctness of loop programs involve the creation of a "loop invariant," a truth-valued function, which is true on

each iteration of the loop and which states that the loop has completed the desired computation when the loop is exited. Such a proof also requires one to show that the loop does in fact terminate. When there are several loops, the correctness of each loop must be established, and then it must be shown that the correctness of each loop implies the correctness of the whole algorithm. The most difficult part of such proofs is usually the identification of the loop invariants. Without a knowledge of how and why the algorithm was designed, such proofs are nearly impossible. Some programming languages, like EUCLID, now allow the specification of the loop invariants and other propositions so that at least in principle it would be possible to mechanically establish the correctness of a fully annotated program.

While it is often easier to establish the correctness of a recursive algorithm, the overwhelming majority of programs are iterative rather than recursive. Why is this true? One reason is that a number of common programming languages like FORTRAN, BASIC, and COBOL do not permit recursion. Another reason is that many programmers believe that recursive programs are always slower than iterative programs. While this may have been true in languages like ALGOL and PL/1, and when recursive programs in ALGOL were compared to iterative programs in FORTRAN, the speed advantage of iterative over recursive programs has disappeared in languages like PASCAL and C.

1.2.2 Comparing Algorithms

For any (solvable) problem there will be an infinity of algorithms which solve the problem. How do we decide which is the "best" algorithm? There are a number of possible ways to compare algorithms. We will concentrate on two measures: time and space. We would like to say that one algorithm is faster, uses less time, than another algorithm if when we run the two algorithms on a

computer the faster one will finish first. Unfortunately, to make this a fair test we would have to keep a number of conditions constant. For example, we would have to code the two algorithms in the same programming language, compile the two programs using the same compiler, and run the two programs under the same operating system on the same computer, and have no interference with either program while it is running. Even if we could practically satisfy all these conditions, we might be chagrined to find that algorithm A is faster under conditions C, but that algorithm B is faster under conditions D.

To avoid this unhappy situation we will only calculate time to order. We let n be some measure of the size of the problem, and give the running time as a function of n. For example: we could use the number of digits as the measure of problem size if the problem is the addition or multiplication of two integers; we could use the number of elements if the problem is to sort several elements; we could use the number of edges or the number of vertices if the problem is to determine if a graph has a certain property. We do not distinguish running times of the same order. For our purposes two functions of n, f(n) and g(n), have the same order if for some N there are two positive constants C_1 and C_2 so that $C_1|g(n)| \leq |f(n)| \leq C_2|g(n)|$ for all $n \geq N$. We symbolize this relation by $f(n) = \theta(g(n))$, read f(n) is order g(n). Thus we will consider two algorithms to take the same time if their running times have the same order. In particular, we do not distinguish between algorithms whose running times are constant multiples of one another.

If we find that algorithm A has a time order which is strictly less than algorithm B, then we can be confident that for any large enough problem algorithm A will run faster than algorithm B, regardless of the actual conditions. On the other hand, if algorithms A and B have the same time order, then we will not predict which one will be faster under a given set of actual conditions.

The space used by an algorithm is the number of bits the algorithm uses to store and manipulate data. We expect the space to be an increasing function of n, the size of the problem. This space measurement ignores the number of bits used to specify the algorithm, which has a fixed constant size independent of the size of the problem. Since we have chosen bits as our unit, we can be more exact about space than we can be about time. We can distinguish an algorithm which uses 3n bits from an algorithm which uses 2n bits. But we will not distinguish an algorithm which uses 3n + 7 bits from an algorithm which uses 3n + 1 bits, because we can hide a constant number of bits within the algorithm itself.

1.2.3 Best algorithms

The third task of analysis of algorithms is to find the best algorithm with respect to a particular measure of resource usage. This involves proving "lower bounds," that is, showing that every algorithm which solves the problem must use at least so much of the particular resource. To establish a best algorithm, one must have both a proof of a lower bound and an algorithm which uses no more than this lower bound. Here a distinction should be made between bounds for an algorithm and bounds for a problem. If one establishes an upper bound on a particular resource used by an algorithm for a problem, then one has an upper bound both for the algorithm and for the problem. If one establishes a lower bound for a problem, then one has a lower bound for all algorithms which solve the problem. But demonstrating a lower bound for one algorithm for a problem does not establish a lower bound for the problem.

2. TOWERS OF HANOI: AN ANALYSIS OF A PROBLEM

2.1 The Towers of Hanoi Problem

In this section, we will demonstrate the three-fold task of analysis of algorithms using the Towers of Hanoi problem. We have chosen the Towers of Hanoi problem because each of the tasks can be demonstrated rather easily, and a best algorithm can be discovered. Material in this section is based on Cull and Ecklund [1985].

In the Towers of Hanoi problem, one is given three towers, usually called A, B, and C, and n disks of different sizes. Initially the disks are stacked on tower A in order of size with disk n, the largest, on the bottom, and disk 1, the smallest, on the top. The problem is to move the stack of disks to tower C, moving the disks one at a time in such a way that a disk is never stacked on top of a smaller disk. An extra constraint is that the sequence of moves should be as short as possible. An algorithm solves the Towers of Hanoi problem if, when the algorithm is given as input n the number of disks, and the names of the towers, then the algorithm produces the shortest sequence of moves which conforms to the above rules.

2.2 A Recursive Algorithm

The road to a best algorithm starts with some algorithm which one then attempts to improve. One often uses some sort of strategy to create an algorithm. A very useful strategy is to look at the problem and see if the solution can be expressed in terms of the solutions of several problems of the same kind, but of smaller size. This strategy is usually called divide-and-conquer. If the problem yields to the divide-and-conquer approach, one can construct a recursive algorithm which solves the problem. This construction also gives almost immediately an inductive proof that the algorithm is correct. Time and space analyses of a divide-and-conquer algorithm are often straightforward, since the algorithm directly gives difference equations for time and space usage.

While these divide-and-conquer algorithms have many nice properties, they may not use minimal time and space. They may, however, serve as a starting point for constructing more efficient algorithms.

Consideration of the Towers of Hanoi problem leads to the key observation that moving the largest disk requires that all of the other disks are out of the way. Hence the n-1 smaller disks should be moved to tower B, but this is just another Towers of Hanoi problem with fewer disks. After the largest disk has been moved the n-1 smaller disks can be moved from B to C; again this is a smaller Towers of Hanoi problem. These observations lead to the following recursive algorithm:

```
PROCEDURE HANOI(A,B,C,n)
   IF n=1 THEN move the top disk from tower A to tower C
          ELSE HANOI(A,C,B,n-1)
               move the top disk from tower A to tower C
               HANOI(B,A,C,n-1).
```

Is this the best algorithm for the problem? We will show that this algorithm has minimum time complexity, but does not have minimum space complexity. First, though, we prove that the algorithm correctly solves the problem, the first task of analysis of algorithms as outlined in the Introduction.

Proposition 1: The recursive algorithm HANOI correctly solves the Towers of Hanoi problem.

Proof: Clearly the algorithm gives the correct minimal sequence of moves for 1 disk. If there is more than 1 disk the algorithm moves n-1 disks to tower B, then moves the largest disk to tower C, and then moves the n-1 disks from tower B to tower C. This is precisely what is required in a minimum move algorithm because,according to the rules,the largest disk can only be moved when all the other n-1 disks are on a single tower. So the n-1 disks must be moved from tower A to some other tower. Clearly at least one move is required to move the largest disk from tower A to tower C. When the largest disk is moved to tower C, the other n-1 disks are on a single tower and still have to be moved to tower C. By inductively assuming n-1 disks are moved in the minimum number of moves, we see that the algorithm for n disks makes no more than the minimal number of moves and finishes with all the n disks moved from tower A to tower C. □

Here we should remark that we have not only produced a provably correct algorithm for the problem; we have also shown that the minimal sequence of moves is unique. This uniqueness makes the proof of correctness easy. The proof would be more complicated if more than one minimum sequence were possible.

We would like to calculate the running time of HANOI, but we don't know how long various operations will take. How long will it take to move a disk? How long will it take to subtract 1 from n? How long will it take to test if n = 1? How long will it take to issue a procedure call? Because we only wish

to calculate time to order we don't have to answer these questions exactly, but we do have to make a distinction between operations which take a constant amount of time, independent of n, and operations whose running time depends on n.

One possibility is to assume that each operation takes constant time independent of n. AHU [Aho, Hopcroft, and Ullman, 1974] calls this assumption the uniform cost criterion. With this uniform cost assumption and letting T(n) be the running time for n disks, we have the difference equation

$$T(n) = 2T(n-1) + c,$$

because there are 2 calls to the same procedure with n-1 rings and c is the sum of the constant running times for the various operations. Letting T(1) be the running time of the algorithm for 1 disk, we find

$$T(n) = (T(1) + c)2^{n-1} - c,$$

which can be verified by direct substitution. This gives

$$T(n) = \theta(2^n),$$

since $\frac{T(1)}{2} 2^n \leq T(n) < (\frac{T(1) + c}{2})2^n$.

Another possibility is to assume that some of the operations have running times which are a function of n. But which function of n should we use? Each of the numbers in the algorithm is between 1 and n, and the disks can also be represented by numbers between 1 and n. Since such numbers can be represented using about log n bits, it seems reasonable to assume that each operation which manipulates numbers or disks has running time which is a constant times log n. AHU calls this the logarithmic cost criterion and suggests using it when the numbers used by an algorithm do not have fixed bounds. Using the logarithmic cost criterion we have the difference equation

$$T(n) = 2T(n-1) + c \log n$$

for the running time of the algorithm. This difference equation has the solution

$$T(n) = 2^n \left[\frac{T(1)}{2} + c \sum_{i=1}^{n} \frac{\log i}{2^i} \right]$$

which can be verified by substitution. Since the summation in this solution converges, as one can demonstrate by the ratio test, and assuming that the constants are positive, we have

$$T(n) = \theta(2^n).$$

Since both cost criteria give the same running time, we conclude:

Proposition 2: The algorithm HANOI has running time $\theta(2^n)$.

Although we have established the running time for a particular algorithm which solves the Towers of Hanoi problem, we have not yet established the time complexity of the problem. We need to establish a lower bound so that every algorithm which solves the problem must have running time greater than or equal to the lower bound. We establish $\theta(2^n)$ as the lower bound in the proof of the following proposition.

Proposition 3: The Towers of Hanoi problem has time complexity $\theta(2^n)$.

Proof: Following the proof of Proposition 1, a straightforward induction shows that the minimal number of moves needed to solve the Towers of Hanoi problem is 2^n-1. Since each move requires at least constant time we have established the lower bound on time complexity.

An upper bound for the time complexity of the problem comes from Proposition 2. Since the upper bound and lower bound are equal to order, we have established the $\theta(2^n)$ time complexity of the problem. ▨

Now that we know HANOI's time complexity we would like to consider its space complexity. First we will establish a lower bound on space which follows from the lower bound on time.

Proposition 4: Any algorithm which solves the Towers of Hanoi problem must use at least n + constant bits of storage.

Proof: Since the algorithm must produce 2^n-1 moves to solve the problem, the algorithm must be able to distinguish 2^n different situations. If the algorithm did not distinguish this many situations, then the algorithm would halt in the same number of moves after each of the two nondistinguished situations, which would result in an error in at least one of the cases.

The number of situations distinguished by an algorithm is equal to the number of storage situations times the number of internal situations within the algorithm. Since the algorithm has a fixed finite size, it can have only a constant number of different internal situations. The number of storage situations (states) is 2 to the number of storage bits. Thus $C \cdot 2^{BITS} \geq 2^n$, and so BITS $\geq$ n - log C = n + constant. ∎

In order to discuss the space complexity of the recursive algorithm, let us now consider the data structure used. Two possible data structures are the array and the stack. An array is a set of locations indexed by a set of consecutive integers so that the information stored at a location in the array can be referenced by indicating the integer which indexes the location. For example, the information at location I in the array ARRAY would be referenced by ARRAY[I]. A stack is a linearly ordered set of locations in which information can be inserted or deleted only at the beginning of the stack.

The towers could each be represented by an array with n locations, and each location would need at most log n bits. So an array data structure with θ(n log n) bits would suffice. Alternately, each tower could be represented by a stack. Each stack location would need log n bits, so again this is an θ(n log n) bit structure. Actually a savings would be made. Since only n disks have to be represented, the stack structure needs only n locations versus the 3n locations used by the array structure. Another possible structure is an array in which the i^{th} element holds the name of the tower on which the i^{th} disk

is located. This structure uses only $\theta(n)$ bits. Yet another possibility is to not represent the towers, but to output the moves in the form FROM __ TO __. Thus we could use no storage for the towers.

The recursive algorithm still requires space for its recursive stack. When a recursive algorithm calls itself, the parameters for this new call will take the places of the previous parameters, so these previous parameters are placed on a stack from which they can be recalled when the new call is completed. Also placed on the stack is the return address, the position in the algorithm at which execution of the old call should be resumed. All of this information, the parameters and the return address, for a single call are referred to as a stack frame. At most n stack frames will be active at any time and each frame will use a constant number of bits for the names of the towers and log n bits for the number of disks. So the recursive algorithm will use $\theta(n \log n)$ bits whether or not the towers are actually represented. We summarize these considerations by the following proposition.

Proposition 5: The recursive algorithm HANOI correctly solves the Towers of Hanoi problem and uses $\theta(2^n)$ time and $\theta(n \log n)$ space.

The recursive algorithm uses more than minimal space. We are faced with several possibilities:

1) Minimal space is only a lower bound and is not attainable by any algorithm;
2) Minimal space can only be achieved by an algorithm which uses more than minimal time;
3) Some other algorithm attains both minimal time and minimal space.

By developing a series of iterative algorithms, we will arrive at an algorithm which uses both minimal time and minimal space.

2.3 Improved Algorithms

As a first step in obtaining a better algorithm, we will consider an iterative algorithm which simulates the recursive algorithm for $n \geq 2$. In this algorithm, RECURSIVE SIM, we have chosen to explicitly keep track of the stack counter because this will aid us in finding an algorithm using even less space.

```
PROCEDURE RECURSIVE SIM (A,B,C,n)
   I:= 1
   L1[1]:= A; L2[1]:= C; L3[1]:= B
   NUM[1]:= n-1 ; PAR[1]:= 1 ; PAR[0]:= 1
   WHILE I ≥ 1 DO
     IF NUM[I] > 1
        THEN L1[I+1]:= L1[I]
             L2[I+1]:= L3[I]
             L3[I+1]:= L2[I]
             NUM[I+1]:= NUM[I] - 1
             PAR[I+1]:= 1
             I:= I+1
        ELSE MOVE FROM L1[I] TO L3[I]
             WHILE PAR[I] = 2 DO
                  I:= I-1
             IF I ≥ 1 THEN MOVE FROM L1[I] TO L2[I]
                           PAR[I]:= 2
                           TEMP:= L1[I]
                           L1[I]:= L3[I]
                           L3[I]:= L2[I]
                           L2[I]:= TEMP
```

The names of the towers are stored in the three arrays L1, L2, L3; the number of disks in a recursive call is stored in NUM; and the value of PAR indicates whether a call is the first or second of a pair of recursive calls.

RECURSIVE SIM sets up the parameters for the call HANOI (A,C,B,n-1). When the last move for this call is made, the arrays will contain the parameters for calls with 1 through n-2 disks, where each of these calls will have PAR=2. The arrays will still contain the parameters for the (A,C,B,n-1) call with PAR=1. The inner WHILE loop will pop each of the calls with PAR=2, leaving the array counter pointing at the (A,C,B,n-1) call. Since I will be 1 at this point, the IF condition is satisfied and the MOVE FROM L1[I] TO L2[I] accomplishes the MOVE FROM A TO C of the recursive algorithm HANOI. The following assignment statements set up the call (B,A,C,n-1) with PAR=2. So when the moves for this call are completed, all of the calls in the array will have PAR=2, and the inner WHILE loop will pop all of these calls setting I to 0. Then the IF condition will be false, so no operations are carried out, and the outer WHILE condition will be false so the algorithm will terminate.

Proposition 6: The RECURSIVE SIM algorithm correctly solves the Towers of Hanoi problem, and uses $\theta(2^n)$ time and $\theta(n \log n)$ space.

Proof: Correctness follows since this algorithm simulates the recursive algorithm which we have proved correct. The major space usage is in the arrays. Since each time I is incremented the corresponding NUM[I] is decremented and since NUM[I] never falls below 1, there are at most n-1 locations ever used in an array. The four arrays L1, L2, L3, and PAR use only a constant amount of space for each element, but NUM must store a number as large as n-1 so it uses $\theta(\log n)$ bits for an element. Thus the arrays use $\theta(n \log n)$ bits.

Now we have to argue about time usage. Most of the operations deal with constant-sized operands so these operations will take constant time. The

exceptional operations are incrementing, decrementing, assigning, and comparing numbers which may have $\theta(\log n)$ bits. A difference equation for the time is

$$T(n) = 2T(n-1) + C \log n,$$

where $T(n)$ is the time to solve a problem with n disks, and $C \log n$ is the time for manipulating the numbers with $\theta(\log n)$ bits. As in the proof of Proposition 1, we have $T(n) = \theta(2^n)$. ∎

Notice that this algorithm does not improve on the recursive algorithm, but study of this form can lead to a saving of space. Storing the array NUM causes the use of $\theta(n \log n)$ space. If we did not have to store NUM, the algorithm would use only $\theta(n)$ space. Do we need to save NUM? NUM is used as a control variable so it seems necessary. But if we look at NUM[1] + 1 we get n. When NUM[I+1] is set, it is set equal to NUM[I] - 1, but then

$$\text{NUM[I+1]} + I + 1 = \text{NUM[I]} - 1 + I + 1$$
$$= \text{NUM[I]} + I = n.$$

Thus the information we need about NUM is stored in I and n. So if we replace the test on NUM[I] = 1 with a test on I = n-1, we can dispense with storing NUM and improve the space complexity from $\theta(n \log n)$ to $\theta(n)$. This replacement does not increase the time complexity of any step in the algorithm, so the time complexity remains $\theta(2^n)$.

Our new procedure is

```
PROCEDURE NEW SIM (A,B,C,n)
   I: = 1
   L1[1]:= A ; L2[1]:= C ; L3[1]:= B
   PAR[1]:= 1  ; PAR[0]:= 1
   WHILE I ≥ 1 DO
     IF I ≠ n-1
        THEN L1[I+1]:= L1[I]
             L2[I+1]:= L3[I]
             L3[I+1]:= L2[I]
             PAR[I+1]:= 1
             I:= I+1
        ELSE MOVE FROM L1[I] TO L3[I]
             WHILE PAR[I] = 2 DO
                  I:= I-1
             IF I ≥ 1 THEN MOVE FROM L1[I] TO L2[I]
                           PAR[I]:= 2
                           TEMP:= L1[I]
                           L1[I]:= L3[I]
                           L3[I]:= L2[I]
                           L2[I]:= TEMP
```

From the above observation we have:

Proposition 7: NEW SIM correctly solves the Towers of Hanoi problem and uses $\theta(2^n)$ time and $\theta(n)$ space.

Although we have reached $\theta(n)$ space we would like to decrease the space even further, hopefully to n + constant bits. If we look at the array PAR, we find that the algorithm scans PAR to find the first element not equal to 2, replaces that element by 2, and then replaces all the previous 2's by 1's. This is analogous to the familiar operation of adding 1 to a binary number, in which we find the first 0, replace it by a 1, and replace all the previous 1's by 0's. So it seems that we can replace the array PAR by a simple counter. The number of bits in the counter will, of course, depend on n.

So far this has not resulted in any saving of space. Will there be enough information in the counter to determine from which tower we should move a disk? The affirmative answer will enable us to achieve a minimal space algorithm. To motivate the design of our minimal space algorithm, we will examine the sequence of 31 moves needed to solve the problem with 5 disks. This sequence is shown in Table 1.

TOWER 0	TOWER 1	TOWER 2	DECIMAL COUNT	COUNT	DISK	FROM	TO
12345	-	-	0	00000	1	0	2
2345	-	1	1	00001	2	0	1
345	2	1	2	00010	1	2	1
345	12	-	3	00011	3	0	2
45	12	3	4	00100	1	1	0
145	2	3	5	00101	2	1	2
145	-	23	6	00110	1	0	2
45	-	123	7	00111	4	0	1
5	4	123	8	01000	1	2	1
5	14	23	9	01001	2	2	0
25	14	3	10	01010	1	1	0
125	4	3	11	01011	3	2	1
125	34	-	12	01100	1	0	2
25	34	1	13	01101	2	0	1
5	234	1	14	01110	1	2	1
5	1234	-	15	01111	5	0	2
-	1234	5	16	10000	1	1	0
1	234	5	17	10001	2	1	2
1	34	25	18	10010	1	0	2
-	34	125	19	10011	3	1	0
3	4	125	20	10100	1	2	1
3	14	25	21	10101	2	2	0
23	14	5	22	10110	1	1	0
123	4	5	23	10111	4	1	2
123	-	45	24	11000	1	0	2
23	-	145	25	11001	2	0	1
3	2	145	26	11010	1	2	1
3	12	45	27	11011	3	0	2
-	12	345	28	11100	1	1	0
1	2	345	29	11101	2	1	2
1	-	2345	30	11110	1	0	2
-	-	12345	31	11111			

Table 1. Towers of Hanoi Solution for 5 disks.

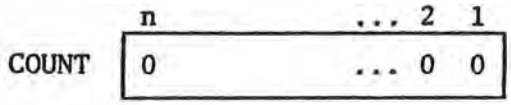
Every other move in the solution involves moving disk 1. So if we know which tower contains disk 1 we would know from which tower to move, in alternate moves, but we might not know which tower to move to. When we consider the three towers to be arranged in a circle, we see from Table 1 that disk 1 always moves in a counterclockwise direction when we have an odd number of disks. Similarly disk 1 always moves in a clockwise direction when we have an even number of disks. Thus by keeping track of the tower which contains disk 1 and whether n is odd or even ,we would know how to make every other move.

For the moves which do not involve disk 1, we know that the move involves the two towers which do not contain disk 1. Looking again at Table 1, we see that the odd numbered disks always move in the same direction as disk 1 and that the even numbered disks always move in the opposite direction. So knowing the towers involved and whether the disk to be moved is odd or even would allow us to decide which way to move.

Can we determine from a counter whether the disk being moved is odd or even? If we look at the COUNT column of Table 1 we see that the position of the rightmost 0 tells us the number of the disk to be moved. Thus a single counter with n bits is sufficient to solve the Towers of Hanoi problem.

We use these facts to construct the algorithm which follows.

```
PROCEDURE TOWERS (n)
   T:= 0   (*TOWER NUMBER COMPUTED MODULO 3*)
 COUNT:= 0   (*COUNT HAS n BITS*)
   P:= { 1 if n is even
       { -1 if n is odd
   WHILE TRUE DO
     MOVE DISK 1 FROM T TO T+P
     T:= T+P
     COUNT:= COUNT + 1
     IF COUNT = ALL 1's THEN RETURN
     IF RIGHTMOST 0 IN COUNT IS IN EVEN POSITION
       THEN MOVE DISK FROM T-P TO T+P
       ELSE MOVE DISK FROM T+P TO T-P
    COUNT:= COUNT + 1
 ENDWHILE
```



A picture of the storage used for COUNT.

Notice that it has n bits, and that we have called the rightmost bit position 1. The positions from right to left are then odd, even, odd, even....

Remarks: We can still improve this algorithm by removing the first COUNT := COUNT + 1 statement and deleting the rightmost bit of COUNT. This would also require changing the numbering of the bits in COUNT so that the rightmost bit is bit 0. An algorithm similar to our TOWERS has recently been published by T. R. Walsh [1982].

We have to show that TOWERS correctly solves the Towers of Hanoi problem. We do this by proving that a certain sequence of moves has been accomplished when COUNT contains a number of the form 2^k-1, so that when k = n, the sequence of moves for HANOI (A,B,C,n) has been completed and the procedure will terminate since COUNT contains all 1's.

Proposition 8: When COUNT = 2^k-1, that is COUNT = $\boxed{00\ldots01\ldots1}$ with k 1's, then:
if $k \not\equiv n \pmod 2$ the correct moves for HANOI (A,C,B,k) have been completed and T contains 1 (which represents B),
if $k \equiv n \pmod 2$ the correct moves for HANOI (A,B,C,k) have been completed and T contains 2 (which represents C).

Proof: If k = 1 the single move T to T + P has been completed, which is A to C if n is odd, and is A to B if n is even, and T contains T + P which is 2 if n is odd and is 1 if n is even. This agrees with our claim.

Notice that COUNT can only take on the value 2^k-1 immediately before the IF ... RETURN statement. Assume that the moves for either HANOI (A,B,C,k) or HANOI (A,C,B,k) have been completed. If $k \not\equiv n \pmod 2$, the next move will be A to C since by assumption T now contains 1; if k is odd, the move is T - P to T + P which is 1 - (1) to 1 + 1 which represents A to C, and if k is even, the move is T + P to T - P which is 1 + (-1) to 1 - (-1) which represents A to C. If $k \equiv n \pmod 2$, the next move will be A to B since by assumption T now contains 2; if k is odd, the move is T - P to T + P which is 2 - (-1) to 2 + (-1) which

represents A to B, and if k is even, the move is T + P to T - P which is 2 + 1 to 2 - 1 which represents A to B.

Next COUNT will be incremented to $\boxed{0\ldots010\ldots0}$, i.e., k trailing 0's. When COUNT = $2^{k+1}-1$, the algorithm will have repeated the same sequence of moves as before since it only "sees" the rightmost information in COUNT, with the difference that T will have started with a different value. The different starting value of T will result in a cyclic permutation of the labels.

If $k \not\equiv n \pmod 2$, then the completed moves will be

HANOI (A,C,B,k)

A to C

HANOI (B,A,C,k),

giving HANOI (A,B,C,k+1) with $k+1 \equiv n \pmod 2$, and T will contain 2 (i.e., 1 + 1).

If $k \equiv n \pmod 2$, then the completed moves will be

HANOI (A,B,C,k)

A to B

HANOI (C,A,B,k) ,

giving HANOI (A,C,B,k+1) with $k+1 \not\equiv n \pmod 2$, and T will contain 1 (i.e., 2 + 2). □

Proposition 9: The algorithm TOWERS uses $\theta(2^n)$ time and n + constant bits of space.

Proof: For space usage, there are n bits in COUNT, and a constant number of bits are used for T and P.

For time, the initialization takes $\theta(n)$ and the WHILE loop is iterated 2^n-1 times. If each iteration took a constant amount of time we would have $\theta(2^n)$, but the test and increment instruction on count could take time $\theta(n)$ giving $\theta(n2^n)$. So we have to show that only $\theta(2^n)$ time is used.

If the value in COUNT is even then incrementing and testing will only require looking at one bit. If the value in COUNT is odd and (COUNT - 1)/2

is even, then the algorithm only looks at 2 bits. In fact, the algorithm will look at k bits in COUNT in 2^{n-k} cases. Thus the time used will be $\theta(\sum_{k=1}^{n} k \cdot 2^{n-k}) = \theta(2^n)$ since $\sum_{k=1}^{\infty} k \cdot 2^{-k}$ converges. □

We summarize these results in the following theorem.

Theorem: Any algorithm which solves the Towers of Hanoi problem for n disks must use at least $\theta(2^n)$ time and n + constant bits of storage. The algorithm TOWERS solves the problem and simultaneously uses minimum time and minimum space.

2.4 Exercises

For the following two algorithms for the Towers of Hanoi problem, prove that the algorithms are correct and compute the time and space these algorithms use.

Exercise 1:

```
PROCEDURE HANOI ITERATIVE (A,B,C,n)
   IF n mod 2 = 0 THEN MOVE[1]:= A TO B
                  ELSE MOVE[1]:= A TO C
   K:= 1
   WHILE n > 1 DO
      n:= n-1; K:= 2*K
      IF n mod 2 = 0 THEN MOVE[K]:= A TO B
                          L1:= C; L2:= A; L3:= B
                     ELSE MOVE[K]:= A TO C
                          L1:= B; L2:= C; L3:= A
      FOR I:= 1 TO K-1 DO
         CASE MOVE[I] OF
              A TO B : MOVE[K+I]:= L1 TO L2
              A TO C : MOVE[K+I]:= L1 TO L3
              B TO A : MOVE[K+I]:= L2 TO L1
              B TO C : MOVE[K+I]:= L2 TO L3
              C TO A : MOVE[K+I]:= L3 TO L1
              C TO B : MOVE[K+I]:= L3 TO L2
```

Hints 1: For correctness you may want to introduce a new variable and prove a statement which says that on each iteration of the WHILE loop the new

variable increases (or if you want decreases), and that at the end of each iteration a Hanoi problem whose size depends on the new variable has been solved. You will need to give the tower names for the problem which has been solved. You will also need to specify the value of the new variable.

For space, you should know that the algorithm is storing each move in the array MOVE.

For time, you may want to consider both the uniform and the logarithmic cost measures.

Exercise 2: (Buneman and Levy [1980])

```
MOVE SMALLEST DISK ONE TOWER CLOCKWISE
WHILE A DISK (OTHER THAN THE SMALLEST) CAN BE MOVED DO
    MOVE THAT DISK
    MOVE THE SMALLEST DISK ONE TOWER CLOCKWISE
ENDWHILE
```

Hints 2: For correctness, you should be careful since this algorithm only solves the original Towers of Hanoi problem when the number of disks is even. You will probably want to introduce a new variable and prove a statement about the configuration of the disks when the number of moves completed is a specific function of your new variable.

For time and space, the above algorithm is incomplete since it doesn't specify the data structure used to determine if a disk can be moved. You might consider representing each tower by a stack of integers with the integers representing the disks on the tower. Alternately you might consider representing the information by an array DISK, so that DISK[I] contains the name of the tower which contains the I^{th} largest disk. You may also find it useful to show that the i^{th} disk is moved 2^{n-i} times.

3. DIVIDE-AND-CONQUER

3.1 What is Divide-and-Conquer?

The algorithm design strategy which breaks a given problem into several smaller problems of the same type is usually called the divide-and-conquer strategy. In the previous section, the recursive algorithm for the Towers of Hanoi problem is an example of a divide-and-conquer algorithm. Given a problem with n disks, this algorithm converts it into two problems with n-1 disks. Each of the subproblems is successively broken into subproblems until problems which can be solved immediately are reached. The Towers of Hanoi algorithm continues forming problems with fewer disks until it reaches problems with 1 disk which can be solved immediately. After the subproblems are solved the divide-and-conquer algorithm then combines the solutions of the subproblems to give a solution to the original problem. In the Towers of Hanoi example, there is no explicit combining, because the necessary combination is simply to solve one subproblem after the other subproblem has been solved. This combination is handled by the ordering of the statements in the algorithm.

The recursive structure of a divide-and-conquer algorithm leads directly to an inductive proof of correctness, and also gives directly a difference equation for the running time of the algorithm.

Consider designing by divide-and-conquer an algorithm to sort the elements of an n-element array. One way to do this is to find the largest element in the array, interchange it with the last element of the array, and then sort the remaining n-1 element array. This algorithm could be written as

```
PROCEDURE SORT(n)
  IF n > 1 THEN LARGEST(n)
                SORT(n-1) ,
```

where LARGEST is an algorithm which handles the largest element. If LARGEST works

correctly then it is easy to prove that SORT works correctly. Similarly if we know how many comparisons LARGEST used, then we could compute the number of comparisons used by SORT from the formula

$$S(n) = S(n-1) + L(n),$$

where S(n) is the number of comparisons used by SORT(n), S(n-1) is the number of comparisons used by SORT(n-1), and L(n) is the number of comparisons used by LARGEST(n). It is easy to design LARGEST(n) so that it uses exactly n-1 comparisons. This gives the difference equation

$$S(n) = S(n-1) + n-1.$$

When there is only 1 element in the array SORT does nothing. This gives the initial condition S(1) = 0. It is easy to check that S(n) = n(n-1)/2 satisfies both the difference equation and the initial condition.

This recursive sorting algorithm can be easily converted to an iterative algorithm because the recursive algorithm is tail-recursive, that is, the only time the algorithm calls itself is at the end of the algorithm. The corresponding iterative algorithm is

```
FOR I := M DOWNTO 2 DO
    LARGEST(I) .
```

It is still easy to write an inductive proof of the correctness of this algorithm. The number of comparisons used by this iterative algorithm can be computed by

$$\sum_{I=M}^{2} L(I) = \sum_{I=M}^{2} (I-1) \quad n(n-1)/2.$$

Both the recursive and iterative sorting algorithms use the same number of comparisons. They also both use space to store the original array. The recursive algorithm has the disadvantage that it uses a stack to keep track of the recursion. This stack requires some space. Further, the recursive algorithm spends some time in manipulating this stack. So in this case, the iterative algorithm would be preferred to the recursive algorithm.

In the above example, we have broken a problem of size n into a single problem

of size n-1. Instead we could try to break the problem of size n into two problems of size n/2. If we could solve the two problems of size n/2, then we would be left with the problem of combining two sorted sequences of size n/2 to form a single sorted sequence of size n. Let us assume that the algorithm MERGE takes as input two sorted sequences and outputs a single sorted array which contains all the elements of the input. From the MERGE algorithm we can construct a divide-and-conquer algorithm MERGESORT:

```
PROCEDURE MERGESORT(A, n)
  IF n > 1 THEN BREAK A into two arrays A1 & A2
                EACH OF SIZE n/2
                MERGESORT(A1, n/2)
                MERGESORT(A2, n/2)
                MERGE(A1, A2) .
```

As usual it is easy to construct an inductive proof of correctness of this algorithm.

To calculate the number of comparisons used by this sort, we need to know the number of comparisons used by MERGE. Although this number will depend on which elements are actually in the two subarrays A_1 and A_2, it is clear that at most n-1 comparisons are used because each comparison results in an element being merged into its proper place in the output array. So in worst case the number of comparisons used by MERGESORT is given by the difference equation

$$C(n) = 2C(n/2) + n-1$$

because the algorithm with input of size n calls itself twice with input of size n/2 and MERGE uses at most n-1 comparisons. For an initial condition we have $C(1) = 0$ since the algorithm does nothing when $n = 1$. The solution to this equation is

$$C(n) = n \log n - n + 1$$

where log means logarithm to the base 2. The solution can be easily verified using induction. This solution can also be written as $C(n) = \theta(n \log n)$.

If the number of comparisons is the measure of resource usage, then MERGESORT is preferred to the previous sorting algorithms because MERGESORT is $\theta(n \log n)$ while the other sorting algorithms are $\theta(n^2)$.

3.2 Divide-and-Conquer Difference Equations

Consider a divide-and-conquer algorithm which breaks a problem of size n into subproblems each of size n/c. Assume that there are a such subproblems and that the algorithm takes $b\, n^m$ time to split the original problem into subproblems and to combine the solutions of the subproblems to give the solution to the original problem. A difference equation for the time used by the algorithm is

$$T(n) = a\, T(n/c) + b\, n^m .$$

The solution for the time used by the algorithm is

$$T(n) = \begin{cases} \theta(n^m) & \text{if} \quad a < c^m \\ \theta(n^m \log n) & \text{if} \quad a = c^m \\ \theta(n^{\log_c a}) & \text{if} \quad a > c^m . \end{cases}$$

The derivation of the solution is not too complicated but it involves a number of details, so we will not give it in full. To derive the solution, you can first convert the divide-and-conquer equation to a standard linear constant coefficient difference equation by the substitution $n = c^r$ and $T(n) = t_r$. Use standard techniques to solve this difference equation. Determine which term in the solution will be largest, and from the assumptions that a, b, and c are positive and the initial condition is nonnegative, show that the coefficient of this largest term is positive.

The above solution for the time used by the algorithm also holds for the divide-and-conquer equation

$$T(n) = a\, T(n/c) + P(n),$$

where P(n) is a polynomial of degree m. This holds because the equation is linear,

so the solutions due to the various terms in the polynomial can be linearly combined, and therefore the solution due to the highest power term in the polynomial will dominate.

Time here should be taken in a broad sense. It could mean actual time as measured by a clock, but it could also mean the number of times a particular operation is used. For example, when we considered sorting, "time" was the number of comparisons used. In many numerical algorithms, time is the number of multiplications and divisions used.

3.3 Some Divide-and-Conquer Examples

For the running time of MERGESORT we have the equation: $T(n) = 2\,T(n/2) + b\,n$. This equation holds for either the number of comparisons or for the total running time. Since $a = 2$, $c = 2$, $m = 1$, we have $a = c^m$ and the solution is $T(n) = \theta(n \log n)$.

A fairly common numeric problem is calculating the product of two polynomials. The input is the coefficients of the two polynomials, and the desired output is the coefficients of the product polynomial. The usual algorithm for this problem proceeds iteratively by multiplying each coefficient of the first polynomial by the first coefficient of the second polynomial, then multiplying each coefficient of the first polynomial by the second coefficient of the second polynomial, and adding these products to the appropriate partial coefficient of the product polynomial; this process is continued until all of the coefficients of the second polynomial have been used. If each of the polynomials have n coefficients then this algorithm will use $\theta(n^2)$ multiplications and $\theta(n^2)$ total operations.

The polynomial multiplication problem can also be solved by a divide-and-conquer algorithm which breaks each polynomial in half, multiplies 4 half-size polynomials, and then adds the half-size products in the appropriate way to give the product polynomial. More explicitly, let $P(X)$ and $Q(X)$ be two polynomials with n coefficients each. Write

$$P(X) = P_0(X) + P_1(X)\ X^{n/2}$$

$$Q(X) = Q_0(X) + Q_1(X)\ X^{n/2}.$$

Then $P(X)\ Q(X) = P_0(X)\ Q_0(X) + (P_0(X)\ Q_1(X) + P_1(X)\ Q_0(X))\ X^{n/2} + P_1(X)\ Q_1(X)\ X^n$. For the number of multiplications M(n), we have

$$M(n) = 4\ M(n/2),$$

and since a = 4, c = 2, m = 0, we have $a > c^m$ and the solution is $M(n) = \theta(n^{\log_2 4}) = \theta(n^2)$. For the total number of operations T(n), we have

$$T(n) = 4\ T(n/2) + b\ n$$

because all of the polynomial additions can be carried out with multiple of n coefficient additions, and the multiplications by powers of X simply shift a sequence of coefficients. Since a = 4, c = 2, m = 1, we have $a > c^m$ and $T(n) = \theta(n^2)$.

There seems little point to this divide-and-conquer approach since it gives us the same running time as the usual algorithm, but it suggests that all 4 of the half-size multiplications might not be necessary because we only need the sum $(P_0Q_1 + P_1Q_0)$ rather than both these products. This sum can be computed using only one multiplication if we have P_0Q_0 and P_1Q_1 because $(P_0Q_1 + P_1Q_0) = (P_0 + P_1)(Q_0 + Q_1) - P_0Q_0 - P_1Q_1$.

Our new divide-and-conquer algorithm is $P(X)\ Q(X) = P_0(X)\ Q_0(X) + [(P_0(X) + P_1(X))\ (Q_0(X) + Q_1(X)) - P_0(X)\ Q_0(X) - P_1(X)\ Q_1(X)]X^{n/2} + P_1(X)\ Q_1(X)\ X^n$. This algorithm uses only 3 half-size multiplications so

$$M(n) = 3\ M(n/2)$$

and $T(n) = 3\ M(n/2) + b\ n$

because the number of additions and subtractions is still proportional to n. For M(n) we have a = 3, c = 2, m = 0, so $a > c^m$ and $M(n) = \theta(n^{\log_2 3})$. For T(n) we have a = 3, c = 2, m = 1, so $a > c^m$ and $T(n) = \theta(n^{\log_2 3})$. This divide-and-conquer algorithm will be faster than the usual $\theta(n^2)$ algorithm because $\log_2 3 < 2$.

Another problem in which divide-and-conquer leads to a faster algorithm is

matrix multiplication. If n x n matrices are broken into four n/2 x n/2 matrices then the problem is to compute the matrix C, where

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

The straight-forward algorithm is

$$C_{11} = A_{11} B_{11} + A_{12} B_{21}$$
$$C_{12} = A_{11} B_{12} + A_{12} B_{22}$$
$$C_{21} = A_{21} B_{11} + A_{22} B_{21}$$
$$C_{22} = A_{21} B_{12} + A_{22} B_{22}.$$

The equation for the running time of this algorithm is

$$T(n) = 8\ T(n/2) + b\ n^2\ ,$$

because there are 8 half-size multiplications and the additions can be done in time proportional to n^2. This equation has a = 8, c = 2, m = 2, so $a > c^m$ and $T(n) = \theta(n^{\log_2 8}) = \theta(n^3)$.

A faster divide-and-conquer algorithm was designed by Strassen[1969]. This faster algorithm uses only 7 half-size multiplications. Strassen's algorithm is

$$M_1 = (A_{12} - A_{22})\ (B_{21} + B_{22})$$
$$M_2 = (A_{11} + A_{22})\ (B_{11} + B_{22})$$
$$M_3 = (A_{11} - A_{21})\ (B_{11} + B_{12})$$
$$M_4 = (A_{11} + A_{12})\ B_{22}$$
$$M_5 = A_{11}\ (B_{12} - B_{22})$$
$$M_6 = A_{22}\ (B_{21} - B_{11})$$
$$M_7 = (A_{21} + A_{22})\ B_{11}$$
$$C_{11} = M_1 + M_2 - M_4 + M_6$$
$$C_{12} = M_4 + M_5$$
$$C_{21} = M_6 + M_7$$
$$C_{22} = M_2 - M_3 + M_5 - M_7$$

It is a simple exercise in algebra to prove this algorithm correct. The very real difficulty was discovering the algorithm in the first place. The running time of Strassen's algorithm obeys the equation

$$T(n) = 7\,T(n/2) + b\,n^2$$

because there are 7 half-size multiplications and the additions and subtractions of matrices can be carried out in time proportional to n^2. Thus Strassen's algorithm has $\theta(n^{\log_2 7})$ running time.

As the final example of this section, consider solving for X, the system of linear equations AX = B. A divide-and-conquer approach to this problem would attempt to break this problem into several subproblems of the same kind. By adding multiples of some n/2 of the rows of A to the other n/2 rows of A, A can be reduced to the form $\begin{bmatrix} A_1 & A_2 \\ 0 & A_3 \end{bmatrix}$ and by using the same transformation the vector B will be transformed to $\begin{pmatrix} B_1 \\ B_2 \end{pmatrix}$. If we also split X into $\begin{pmatrix} X_1 \\ X_2 \end{pmatrix}$ the solution to the original problem can be given as the solutions to

$$A_3X_2 = B_2$$

$$A_1X_1 = B_1 - A_2X_2.$$

The equation for the running time of this algorithm is

$$T(n) = 2\,T(n/2) + b\,n^3$$

because we have two half-size subproblems and it takes time proportional to n^3 to reduce A to the required special form. Since a = 2, c = 2, m = 3, we have $a < c^m$ and $T(n) = \theta(n^m) = \theta(n^3)$. There is nothing that particularly recommends this algorithm--it has the same running time as the standard Gaussian elimination algorithm--but it is an example of a divide-and-conquer algorithm with $a < c^m$. An example of a divide-and-conquer algorithm with $a = c^m$ is MERGESORT. Examples of divide-and-conquer algorithms with $a > c^m$ are the polynomial multiplications and matrix multiplication algorithms.

3.4 Polynomial Multiplication and Fast Fourier Transform

In practice the polynomial multiplication algorithms of the last section are not used because there is a much faster algorithm. The faster algorithm is based on the idea that there are two ways to represent a polynomial. A polynomial with n coefficients may be represented by its coefficients or it may be represented by the values of the polynomial at n distinct points. Either representation can be calculated from the other representation. This can be represented schematically by:

$$\text{coefficients} \underset{\text{interpolation}}{\overset{\text{evaluation}}{\rightleftarrows}} \text{values.}$$

If we have the coefficients of two polynomials each with n/2 coefficients and we want the n-1 coefficients of the product polynomial, we could evaluate each of the input polynomials at the same n-1 points and multiply the corresponding values to obtain the values of the product polynomial at these n-1 points. To obtain the desired coefficients we could interpolate a polynomial with n-1 coefficients through these points.

The difficulty with this approach is that the standard methods for evaluation and interpolation are both $\theta(n^2)$, so using them would result in an $\theta(n^2)$ method for multiplying polynomials. On the other hand, there is nothing in the above discussion which forces us to use any particular set of points as evaluation and interpolation points. It is easy to evaluate a polynomial at certain points. For example, the value of a polynomial at 0 is simply one coefficient of the polynomial, and the value of a polynomial at 1 is simply the sum of the coefficients.

It is difficult to see how to generalize the idea of evaluation at 0, but the idea of evaluation at 1 can be generalized if we are willing to allow complex numbers. A complex number w is an n^{th} root of unity if $w^n = 1$. We know by the fundamental theorem of algebra that there are n complex numbers which are n^{th} roots of unity. A primitive n^{th} root of unity is a complex number w such that $w^n = 1$ and

and $w^j \neq 1$ for $1 \leq j \leq n-1$. A primitive root is useful because its powers w^0, w^1, w^2, ..., w^{n-1} give us the n numbers which are n^{th} roots of unity. We can use as our primitive roots the complex numbers $e^{2\pi i/n}$ which can be calculated by $\cos(\frac{2\pi}{n}) + i \sin(\frac{2\pi}{n})$. These numbers only need to be calculated once and stored in a table. This table will also be useful because a primitive n/2 root of unity is w^2 which will already be in your table.

To evaluate a polynomial $a_0 + a_1X + \ldots + a_{n-1}X^{n-1}$ at the n roots of unity we should compute

$$\begin{bmatrix} w^0 & w^0 & \ldots & w^0 \\ w^0 & w^1 & \ldots & w^{n-1} \\ w^0 & w^{1\cdot 2} & \ldots & w^{(n-1)\cdot 2} \\ \vdots & & & \\ w^0 & w^{1\cdot(n-1)} & \ldots & w^{(n-1)\cdot(n-1)} \end{bmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}.$$

Unfortunately if we do this by the obvious algorithm it will take time $\theta(n^2)$ even if we have already calculated the entries in the matrix. However, the matrix has very special structure and if we can make use of this structure we may be able to construct a faster algorithm.

To display the structure of the matrix we will permute some of the columns. Since we are planning to break things in half we will assume that n is a power of 2. Also we will number the rows and columns from 0 to n-1, so the indices can be represented by using log n bits. Our permutation will interchange column j with column Rev(j), where Rev(j) is a number formed by reading the bits of j in reverse order. In our permuted matrix location i, j will contain $w^{i\ \mathrm{Rev}(j)}$.

In the lower left quadrant the matrix will have $i \geq n/2$, $j < n/2$, and Rev(j) will be even. So

$$w^{i\ \mathrm{Rev}(j)} = w^{(n/2 + i - n/2)\mathrm{Rev}(j)} = [w^{n/2}]^{\mathrm{Rev}(j)}\ w^{(i-n/2)\mathrm{Rev}(j)}$$
$$= (-1)^{\mathrm{Rev}(j)}\ w^{(i-n/2)\mathrm{Rev}(j)} = w^{(i-n/2)\mathrm{Rev}(j)},$$

and the lower left quadrant will be identical to the upper left quadrant.

The lower right quadrant has $i \geq n/2$, $j \geq n/2$, and Rev(j) is odd. So

$$w^{iRev(j)} = -\, w^{(i-n/2)Rev(j)}$$

and the lower right quadrant is the negative of the upper right quadrant.

The upper right quadrant has $i < n/2$, $j \geq n/2$ and $Rev(j)-1 = Rev(j-n/2)$. So

$$w^{iRev(j)} = w^{i}\, w^{i(Rev(j)-1)} = w^{i}\, w^{iRev(j-n/2)}$$

and the upper right quadrant is identical to the upper left quadrant multiplied by the diagonal matrix whose i^{th} diagonal entry is w^i.

The upper left quadrant has $i < n/2$, $j < n/2$, and Rev(j) is even. So

$$w^{iRev(j)} = [w^2]^{iRev(j)/2}.$$

The half-size permuted matrix will contain w^2 raised to the iRev(j) power because w^2 is a principal $n/2^{nd}$ root of unity. Of course in the half-size matrix Rev(j) will have (log n) - 1 bits. In the larger matrix Rev(j)/2 simply removes the low order bit which is 0. Thus the upper left quadrant will be idential to the half-size permuted matrix.

From these considerations we have

$$F_{2n} = \begin{bmatrix} F_n & DF_n \\ F_n & -DF_n \end{bmatrix},$$

where D is a diagonal matrix whose k^{th} entry is w^k where w is a principal $(2n)^{th}$ root of unity. As we have seen, the matrix can be used to evaluate a polynomial at all the 2n roots of unity, and since it turns out that this evaluation gives the coefficients of the discrete Fourier expansion of the polynomial, the matrix is called the (permuted) Fourier matrix.

The form of the above matrix suggests a divide-and-conquer algorithm to compute the discrete Fourier tranform of a vector. Since the algorithm will be faster than a method based directly on the definition of Fourier transform, the algorithm is usually called the Fast Fourier Transform (FFT). Assume X is a vector with 2n components. Let X_1 be the first n components of X, and let X_2 be the last n compo-

nents of X. Then

$$F_{2n}\ X = \begin{pmatrix} F_n X_1 + DF_n X_2 \\ F_n X_1 - DF_n X_2 \end{pmatrix}.$$

Notice $F_n X_1$ and $F_n X_2$ need only be computed once each. Their values can then be combined to give $F_{2n}X$. To obtain the nice form the columns of the original matrix had to be permuted, so to obtain the effect of the original matrix on a vector the vector must be permuted before the permuted matrix is applied. The whole algorithm to compute the discrete Fourier tranform of $\overline{X}$ is:

$$\overline{X} \xrightarrow{\text{PERMUTE}} X = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \xrightarrow{\text{HALF-SIZE F}} \begin{pmatrix} F_n X_1 \\ F_n X_2 \end{pmatrix} \xrightarrow{\text{COMBINE}} \begin{pmatrix} F_n X_1 + DF_n X_2 \\ F_n X_1 - DF_n X_2 \end{pmatrix}.$$

In this process the matrix F never really needs to exist. The F's in the above scheme are simply recursive calls to the procedure F. The nonzero elements of D are powers of w. So the powers of w can be computed once, stored in an array, and used from the array when necessary.

The running time for this algorithm can be considered in two parts: the time to permute, and the time for the recursive procedure F. The permutation uses the bit reversals of the number 0 through 2n-1. Since these numbers can be represented using log n + 1 bits, the permutation can be computed in $\theta(n \log n)$. The permutation can be done in $\theta(n)$ if the bit reversals are pre-computed. The running time for the procedure for F obeys the difference equation:

$$T(2n) = 2\ T(n) + b\ n$$

because there are two half-size recursive calls, and the multiplication by D and the additions and subtractions can be carried out in time proportional to n. Thus

$$T(2n) = \theta(n \log n).$$

To use this fast algorithm for polynomial multiplication, we need a fast way to interpolate, or what is the same, a fast way to multiply a vector by the inverse of the F matrix. For the inverse procedure we need the idea of conjugate. The conju-

gate of a complex number a + bi is a - bi. We represent the conjugate of the complex number Z by Z^*. The product $ZZ^* = a^2 + b^2$, which is the square of the length of the vector which represents Z. For a root of unity w, the product $ww^* = 1$ because the length of w is 1. The conjugate of a complex matrix is the transpose of the matrix with each element replaced by its conjugate. The inverse of F matrix is its conjugate divided by the dimension because

$$F_{2n} \cdot F^*_{2n} = \begin{bmatrix} F & DF \\ F & -DF \end{bmatrix} \begin{bmatrix} F^* & F^* \\ F^*D^* & -F^*D \end{bmatrix}$$

$$= \begin{bmatrix} FF^* + DDF^*D^* \\ FF^* - DFF^*D^* \end{bmatrix} \begin{bmatrix} FF^* - DFF^*D^* \\ FF^* - DFF^*D^* \end{bmatrix} = \begin{bmatrix} 2nI & 0 \\ 0 & 2nI \end{bmatrix}.$$

So F^*X can be quickly computed because

$$F^*X = F^*X^{**} = (F^TX^*)^* = P(FPX^*)^*,$$

where F^T is the transpose of F, and P is bit-reversal permutation matrix. The last equality follows because $F^T = PFP$.

Finally the whole algorithm to compute the coefficients of the product polynomial is:

① Place the n coefficients of the first polynomial in the first n components of the vector X of dimension 2n. The other n components of X will contain 0.

② Similarly place the n coefficients of the second polynomial in the vector Y.

③ Permute both X and Y.

④ Use the recursive algorithm to compute both FX and FY.

⑤ Componentwise multiply FX and FY to obtain a vector Z.

⑥ Permute and conjugate Z.

⑦ Use the recursive algorithm to compute FZ.

⑧ Conjugate and divide each component of FZ by 2n.

The resulting vector will contain the 2n-1 coefficients of the product polynomial as its first 2n-1 components. The last component should contain 0. Since this algo-

rithm will likely be carried out using floating point arithmetic, the last component will likely not be exactly 0, but the value in this component will give us an estimate of how exact the other components are.

3.5 How Practical Are Divide-and-Conquer Algorithms?

After creating algorithms in the abstract world of analysis of algorithms, we should now look back to the real world of programs and ask if the divide-and-conquer algorithms, which are theoretically faster, will be practically faster, and whether they will be practical at all. There are several reasons to be skeptical about the practicality of these algorithms. In our abstract world recursion was available at no cost. In the real world recursion may be unavailable or it may have a high cost. In the abstract world we computed time only to order, not distinguishing $10^{23}n^2$ from $2n^2$, but in the real world these functions are very different.

Let us address recursion first. The tempting thing for a theoretician is to say that all "modern" programming languages have recursion so there is no problem. Unfortunately large amounts of programming are done in older languages which do not support recursion. So there is still some reason for considering converting recursive algorithms to iterative algorithms. Even when recursion is available it may not be a good idea to use it. Theoretically we often assume that passing data to procedures is free. In the analysis in this section we have ignored space usage, and we have particularly ignored extra space used to maintain stacks for the recursive procedures. In the real world neither of these should be ignored. Sometimes by doing a space analysis we find that the recursive stack does not get very big, so we are justified in ignoring it. In other cases, we find that the recursive stack gets very large because we are putting copy after copy of the same data on the stack. This problem can often be overcome by having only a single global copy of the data and, instead of passing the data to the recursive procedure, passing only a pointer to

the particular part of the global copy that is needed. In some cases, we find that data passed to the recursive routine may be replaced by a much simpler global structure. In the Towers of Hanoi example, the number of disks was repeatedly passed to the recursive procedure, but the equivalent information could be maintained in a global counter. There are a number of methods which will suffice to convert a recursive routine to an iterative routine, but if they are general enough to work for all recursive routines, they are too general to result in any saving in time or space. Special features of a particular recursive routine can often be exploited in producing a more efficient iterative routine. We have seen how special features can be exploited in the Towers of Hanoi example. As another example, the MERGESORT algorithm works from the top-down, splitting a big array and passing each half to the recursive routine. This routine can be made iterative by working bottom up. Consider each element as a sorted array of size 1 and merge these 2 by 2 until you have sorted arrays of size 2. Again merge these 2 by 2 until you have sorted arrays of size 4. Continue merging until the entire array is sorted. This can be accomplished without passing any arrays to procedures by keeping track of the size and the beginning and ending indices of the subarrays you are merging.

In summary, it is often worthwhile to convert recursive algorithms to iterative algorithms, but you should exploit the special features of the problem to create an efficient iterative algorithm.

Are faster algorithms always faster? Probably not. In our analysis we have concentrated on asymptotic time order. We expect our faster algorithms to be faster than slower algorithms for big enough inputs, but the slower algorithms may well be faster for small inputs. For example, in sorting there are $\theta(n^2)$ algorithms which are faster than an $\theta(n \log n)$ algorithm for $n < 20$, but for $n > 20$ the $\theta(n \log n)$ algorithm is faster. In this case, unless we are dealing with very small data sets, the faster algorithm really is faster. On the other hand, there are $\theta(n^{2.5})$ algo-

rithms for matrix multiplication. These algorithms don't become faster than the standard $\theta(n^3)$ algorithm until one is dealing with 50,000 x 50,000 matrices. Since no one is presently trying to multiply matrices this large, the $\theta(n^{2.5})$ algorithm is only theoretically interesting.

The algorithms we have designed may fail to be practical in other ways. The algorithms are not fool-proof, that is, the algorithms assume that they will receive the type of data they are expecting, and their behavior on unexpected data may well be rather strange. Practical programs should check the input data to make sure it is of the expected kind and issue a warning if the data is not of the expected kind. The algorithm may also not be practical if it solves the wrong problem. For example, the FFT-based polynomial multiplication algorithm is designed for dense polynomials, that is, polynomials in which all or almost all the coefficients are non-zero, but many large scale polynomial multiplication problems arise in which the polynomials are sparse, that is, have almost all the coefficients equal to zero. For such sparse problems there are algorithms which will easily outperform the FFT-based algorithm. Similarly there are special algorithms for sorting which will be slow in general but will be very fast when the input data is almost sorted.

In summary, there are practical issues which should be considered before a theoretically good algorithm is used as a practical program.

3.6 Exercises

a) You have a large number of coins and a pan balance. You may put any number of coins in each pan of the balance. The balance will tell you if the set of coins in one pan weighs the same as the set of coins in the other pan, or it will tell you which set is heavier. Somewhere among your coins is one coin which has a different weight from the other coins. All the coins except this odd coin have exactly the same weight. The problem is to find the odd coin.

Design a divide-and-conquer algorithm to solve this problem. You may assume that the number of coins is a power of 3. Prove that your algorithm is correct. Give and solve a difference equation for the number of times your algorithm uses the balance.

b) Design a divide-and-conquer algorithm to find the two largest elements in an array. Prove that your algorithm is correct. Calculate the number of comparisons it uses. Show by example that your algorithm uses more comparisons than necessary.

c) Two string C_1 and C_2 commute (that is, $C_1C_2 = C_2C_1$) iff there is a string w so that $C_1 = w^{k_1}$ and $C_2 = w^{k_2}$. Start from an inductive proof that w exists and construct an algorithm to find w.

4. AVERAGE CASE

4.1 What is Average Case?

Up till now we have considered the running time of an algorithm to be a function of the size of the input, but what happens when there are several different inputs of the same size? An algorithm may treat all inputs of the same size in the same way, or it may handle some inputs more quickly and some inputs more slowly. The maximum of the running time over all inputs of the same size is called the worst case running time. The minimum running time over all inputs of the same size is called the best case running time. The running times averaged over all inputs of the same size is called the average case running time. The average case running time is not the same as the average of the worst case and the best case. It is often difficult to calculate the average case time because the probability associated with each of the various inputs of a particular size is unknown. For definiteness and simplicity, it is often assumed that each input with the same size is equally likely to occur. With this assumption average case can be calculated.

4.2 Some Examples of Average Case Behavior

As an example of average case behavior, consider the following algorithm:

```
PROCEDURE LARGETWO
    FIRST := B[1]
    SEC   := B[2]
    FOR I := 2 TO n DO
      IF B[I] > SEC
        THEN SEC := FIRST; FIRST := B[I]
        ELSE IF B[I] >  SEC
                THEN SEC := B[I].
```

This algorithm should find the two largest elements in an array. We will consider the number of comparisons of array elements it uses to accomplish this task. In the FOR loop, for each I the algorithm makes either 1 or 2 comparisons. In best case the algorithm makes 1 comparison for each I giving a total of n-1 comparisons. In worst case the algorithm makes 2 comparisons for each I giving a total of 2(n-1) comparisons.

Let A(n) be the number of comparisons used on average by this algorithm. Since the algorithm proceeds in one direction across the array, we may reasonably assume that for the first n-1 elements the algorithm will on average use A(n-1) comparisons, that is, the same number it would use if the last element did not exist. For the last element it will use at least one comparison. It will use a second comparison exactly when B[n] $\leq$ FIRST, but since FIRST will be the largest of the first n-1 elements, the algorithm will use a second comparison as long as B[n] is not the largest element in the array. The probability that B[n] is not the largest element is $\frac{n-1}{n}$, if we assume that each element is equally likely to be the largest element. From these considerations we have:

$$A(n) = A(n-1) + 1 + \frac{n-1}{n} = A(n-1) + 2 - 1/n.$$

For an initial condition we have A(2) = 3/2, since for two elements the algorithm is equally likely to use one or two comparisons. The solution to this difference equation is

$$A(n) = 2(n-1) - \sum_{j=1}^{n} 1/j$$

and for large n

$$A(n) \to 2(n-1) - \ln n + \text{const.}$$

So we have that the average case of this algorithm is very close to the algorithm's worst case. Notice that if took (worst + best)/2 we would get 3/2(n-1), which will be a severe underestimate of the average case. (As an aside, we should mention that this is a poorly set up algorithm since it assumes that the array has at least two

locations. If this algorithm were used with an array containing a single element, the results would be unpredictable.)

The QUICKSORT algorithm has a more complicated average case analysis. The algorithm is:

PROCEDURE QUICKSORT(A)

PICK AN ELEMENT $\propto$ OF A AT RANDOM

DIVIDE A INTO A_1 (THE ELEMENTS OF A WHICH ARE LESS THAN $\propto$.)

A_2 (THE ELEMENTS OF A WHICH EQUAL $\propto$. WE WILL ASSUME THERE IS ONLY ONE SUCH ELEMENT.)

A_3 (THE ELEMENTS OF A WHICH ARE GREATER THAN $\propto$.)

RETURN(QUICKSORT(A_1) · A_2 · QUICKSORT(A_3)) .

The worst case for QUICKSORT occurs when A_1 or A_3 contains n-1 elements. If we let W(n) stand for the worst case running time of QUICKSORT, we have

$$W(n) = W(n-1) + bn$$

because the separation into A_1, A_2, A_3 takes time proportional to n. From this we have $W(n) = \theta(n^2)$. The best case for QUICKSORT occurs when A_1 and A_3 are each approximately n/2. If we let B(n) be the best case running time, we have approximately

$$B(n) = 2B(n/2) + bn$$

and $B(n) = \theta(n \log n)$. For the worst case to occur each recursive splitting must have one of the sets A_1 or A_3 empty. For the best case to occur each recursive splitting must have A_1 and A_3 of approximately equal size.

Will the average case be like the worst case or like the best case? We might expect nearly equal splits to occur more frequently than one-sided splits, so we could guess that average case is probably closer to best case than to worst case. If we let A(n) be the average case running time and assume that every split is equally likely, we have

$$A(n) = \frac{1}{n} \sum_{k=1}^{n} [A(k-1) + A(n-k) + b(n+1)],$$

where b(n+1) is the time to split and combine. Then

$$nA(n) - (n-1)\,A(n-1) = \sum_{k=1}^{n} A(k-1) - \sum_{k=1}^{n-1} A(k-1) + \sum_{k=1}^{n} A(n-k) - \sum_{k=1}^{n-1} A(n-1-k)$$

$$+ bn(n+1) - bn(n-1) = 2A(n-1) + 2bn$$

and $nA(n) = (n+1)\,A(n-1) + 2bn$ and $\frac{A(n)}{n+1} = \frac{A(n-1)}{n} + \frac{2b}{n+1}$. Letting $Z(n) = A(n)/(n+1)$, we have $Z(n) = Z(n-1) + \frac{2b}{n+1}$, which has the solution $\frac{1}{(j+1)}$.

But $Z(n) \to C_2 + C_3 \log n$, so $A(n) \to C_2(n+1) + C_3(n+1) \log n$, giving $A(n) = \theta(n \log n)$.

In summary, the average case behavior of an algorithm is somewhere between the worst case and the best case behavior, but it may be very close to the best case behavior or very close to the worst case behavior.

4.3 Exercises

a) A number of useful tricks were used in deriving the average case behavior of QUICKSORT. For the procedure LARGETWO set up a difference equation with a summation by assuming that each element is equally likely to be the largest element. Then employ the techniques used on QUICKSORT to obtain the simple difference equation we used for LARGETWO.

b) For the following procedure do an average case analysis. Is the average case nearer to worst case or to best case?

```
PROCEDURE BIGTWO
   FIRST := B[1]
   SEC   := B[2]
   FOR I := 2 TO n DO
     IF B[I] ≥ SEC
       THEN IF B[I] > FIRST
              THEN SEC := FIRST; FIRST := B[I]
              ELSE SEC := B[I].
```

5. LOWER BOUNDS

5.1 Trivial Lower Bounds

A lower bound is a statement that every algorithm which solves a particular problem must use at least so much of a particular resource. We have already met some lower bounds in the Towers of Hanoi example. There we argued that every algorithm for the problem must take at least $\theta(2^n)$ time because the output must contain 2^n-1 moves. Such a lower bound is called a trivial lower bound because it is obvious that an algorithm must take at least as much time as it takes to print its output. Although such a bound is called trivial it may not be trivial to compute the length of the output. For example, it was not immediately obvious that the output for the Towers of Hanoi must have length 2^n-1.

Trivial lower bounds can also depend on the input. If we can show that to compute the correct answer an algorithm must read all of its input, then the algorithm must use time which is at least as great as the length of the input. For example, if an algorithm is supposed to multiply an $n \times n$ matrix times an n component vector, we have the trivial lower bound of $\theta(n^2)$ because if the algorithm does not read some of the input, then there are vectors and matrices for which the algorithm gives the wrong answer. As another example, consider an algorithm which is supposed to find the largest element in an n element array. This algorithm must take time at least $\theta(n)$, since if it doesn't look at all the elements in the array there are arrays for which the algorithm will give an incorrect answer. There are problems in which an algorithm does not have to look at the whole input. For example, to determine if a list is empty an algorithm only has to look at the first element of the list; it does not need to look at all elements of the list.

Input and output lower bounds can vary widely. Sometimes the output bound will be larger than the input bound. Sometimes the input bound will be larger than the

output bound. In the Towers of Hanoi problem the output bound is $\theta(2^n)$, but the input bound is only $\theta(\log n)$ because n and the names of the towers can be specified in $\theta(\log n)$ bits. In the variant of the Towers of Hanoi in which one is asked if a given configuration is used in moving the disks from Tower A to tower C, the output bound is $\theta(1)$ since the yes or no answer can be specified with a single bit, while the input bound is $\theta(n)$ because $\theta(n)$ bits are needed to specify which tower contains which disk.

So far our lower bounds have been best case bounds, that is, even in best case every algorithm must use at least the time specified by the lower bound. Consider the problem of determining if a list contains a particular element. The desired element could be the first element in the list, so the best case lower bound is $\theta(1)$. On the other hand, to be sure that the desired element is not in the list an algorithm must look at every element, so the worst case lower bound is $\theta(n)$.

5.2 Lower Bounds on Specific Operations

The trivial lower bounds merely state that an algorithm must do the required input and output; they say nothing about any computation the algorithm must perform. To obtain lower bounds involving specific operations, the operations which the algorithm is allowed to use must be specified. In this section we will consider algorithms in which the only allowed operations on input elements are comparisons. This will suffice for the problems considered. For other problems, the operations of addition, subtraction, and multiplication might be allowed. We refer the interested reader to Winograd[1980].

Let us consider deriving a lower bound on the number of comparisons needed to find the largest element in an array. We use an input bound argument. If some element is not compared to any other element then either the algorithm says that the uncompared element is the largest element, which will be false in some cases, or the

algorithm says that some other element is largest, which will be false when the uncompared element is largest. Since elements can be compared two at a time, we have that any algorithm which correctly finds the largest using only comparisons must use at least n/2 comparisons.

We have that n/2 comparisons are needed, but we expect that n-1 comparisons are needed because our methods for finding the largest use n-1 comparisons. We can consider an algorithm as forming a directed graph. Each time a comparison is made, a directed edge from the larger to the smaller element is put into the graph. When the largest element has been found then for every other element there will be a directed path in the graph from the largest to the other element. If not, we could replace the element without a path by an element larger than the element that the algorithm reports to be largest. Since the algorithm is assumed to use only comparisons, the algorithm cannot detect this replacement. So if not all these paths exist, then the algorithm will give incorrect answers. If all these paths exist, then the graph formed from the directed graph by replacing the directed edges with undirected edges will be connected. Since a connected graph has at least n-1 edges and since each edge corresponds to a comparison, n-1 comparisons are needed.

A different way to obtain this lower bound is to think of finding the largest as a dynamic system. As an algorithm proceeds it classifies each element as belonging to one of three sets: the set U which contains elements which have been compared to no other element; the set S of elements which have been compared to some other element and have been found to be smaller; the set L of elements which have been compared to some other element and have been found to be larger than any element they have been compared to. The states of the dynamic system will be vectors with three integer components giving the size of the sets U, S, and L. When the algorithm starts the dynamic system will be in state (n, 0, 0), and when the algorithm correctly terminates the dynamic system will be in state (0, n-1, 1).

In each comparison of the algorithm,the state changes in one of the six following ways, where the sets from which the elements to be compared are indicated by the names of the sets above the arrows.

$$(U, S, L) \xrightarrow{U:U} (U-2, S+1, L+1)$$

$$(U, S, L) \xrightarrow{S:S} (U, S, L)$$

$$(U, S, L) \xrightarrow{L:L} (U, S+1, L-1)$$

$$(U, S, L) \xrightarrow{U:S} (U-1, S+1, L) \text{ or } (U-1, S, L+1)$$

$$(U, S, L) \xrightarrow{U:L} (U-1, S+1, L)$$

$$(U, S, L) \xrightarrow{S:L} (U, S+1, L-1) \text{ or } (U, S, L)$$

Since we have a model of the action of any algorithm which finds the largest element using only comparisons, we can establish a lower bound for the algorithm by establishing a lower bound on the number of transitions in the dynamic system required to go from state (n, 0, 0) to state (0, n-1, 1). Consider the middle component. It must increase from 0 to n-1, but this component can increase by at most 1 in any of the transitions. Hence at least n-1 transitions in the dynamic system and n-1 comparisons in the algorithm are required.

It is important to notice that this result really does depend on the type of algorithm being considered. By allowing different operations the largest element can be located using only log n comparisons. Consider the following algorithm:

```
FOR I := 1 TO n DO
   NUM[I] := n**a_I
FUNCTION BIG(i,j)
   IF j = i+1
      THEN IF a_i ≥ a_j
              THEN BIG := i
              ELSE BIG := j
```

$$\underline{\text{ELSE IF}} \quad \sum_{k=1}^{(i+j)/2} \text{NUM[K]} \geq \sum_{k=\frac{i+j}{2}+1}^{j} \text{NUM[K]}$$

THEN BIG := BIG(i, (i+j)/2)

ELSE BIG := BIG((i+j)/2 + 1, j)

END FUNCTION

LARGEST := $a_{BIG(1,n)}$.

In this algorithm we are assuming that the elements are distinct positive integers. The ** indicates exponentiation. Let C(n) be the number of comparisons used by the algorithm with n elements; then C(n) = C(n/2) + 1 and C(2) = 1. Thus C(n) = log n. (If you also want to count the comparison between i and j+1, there are 2 log n comparisons.) The trick here is to make sure that the largest element is in the half-size subset with the largest sum. Consider the case in which LARGEST is in the first half. Then

$$\sum_{k=\frac{n}{2}+1}^{n} \text{NUM[K]} \leq \frac{n}{2} n^{\text{LARGEST}-1} = \frac{n}{2}^{\text{LARGEST}} < n^{\text{LARGEST}} + \frac{n}{2} - 1 \leq \sum_{k=1}^{n/2} \text{NUM[K]},$$

so this algorithm will work correctly. The use of exponentiation allows us to make the largest element so large that we can do a binary search through the subsets and locate the largest element quickly.

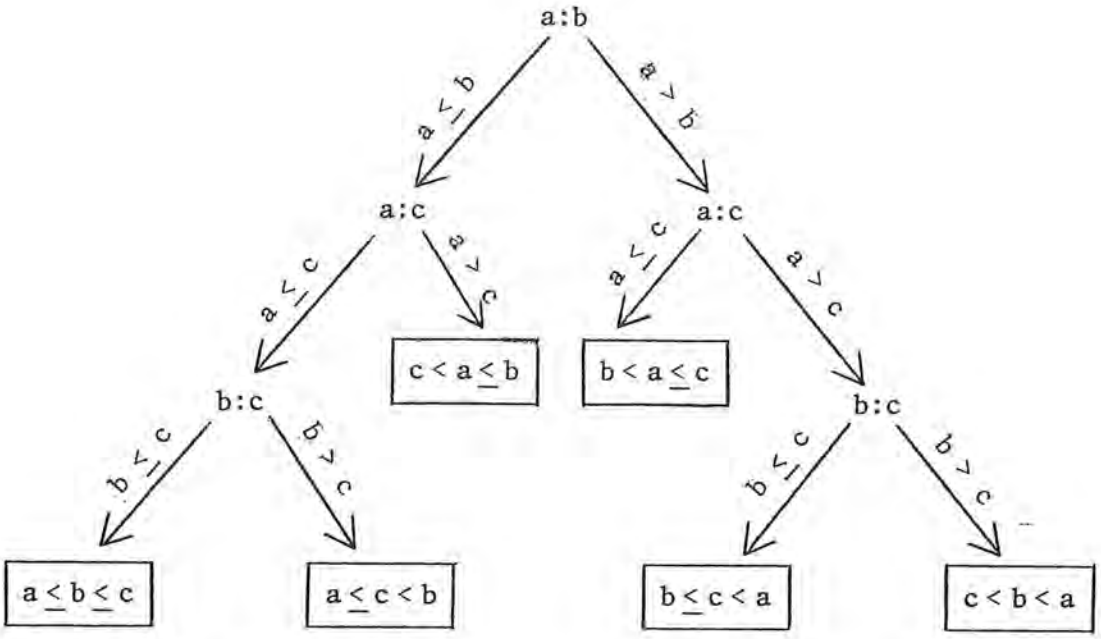
This example should indicate that lower bound proofs are sensitive to operations allowed in the algorithm. It is thus important in lower bound arguments to indicate the class of algorithms being considered.

5.3 Lower Bounds on Sorting

A very traditional and ubiquitous problem is sorting. Given n elements from a totally ordered set put them in order from the smallest to the largest. We have already encountered several algorithms for this problem. Some had $\theta(n^2)$ worst case running time. Some had $\theta(n \log n)$ worst case running time. Some had $\theta(n \log n)$

average case running time. Some had $\theta(n \log n)$ best case running time. From these examples, one might conjecture that an $\theta(n \log n)$ lower bound could be established for best case and hence for average and worst cases, at least for algorithms which sort by using only comparisons. Unfortunately this conjecture is false. If the elements are already sorted, an algorithm could check in only n-1 comparisons that the elements were in order. It is not essential that the elements actually be in order; for any particular arrangement of the elements, an algorithm can be created which tests for this arrangement using n-1 comparisons, and if the elements are in this particular arrangement the algorithm will sort them with no additional comparisons. So any best case lower bound for sorting must be less than or equal to n-1 comparisons. But no smaller lower bound is possible because if an array is sorted then the largest element can be found using no comparisons, since the largest element is the last element in the array. Since we have already demonstrated a best case lower bound of n-1 comparisons to find the largest element, n-1 comparisons is also the best case lower bound for sorting.

It still is reasonable to believe that a lower bound of $\theta(n \log n)$ may be possible for worst case and average case. To obtain these bounds we introduce a model of computation called the comparison tree. Each internal node of a comparison tree contains an expression like a:b which means compare a to b. From a comparison node there are two arrows to other nodes. One arrow is labeled $a \leq b$ and the other arrow is labeled $a > b$. There are also external nodes or leaves which indicate the termination of the tree. The comparison tree should capture the idea of an algorithm which uses only comparisons. The algorithm starts at the root of the tree, does the comparison there, and then follows the appropriate arrow. This process continues until a leaf is reached. The following picture shows a comparison tree for an algorithm which sorts 3 elements.



In this example there are 6 leaves because there are 3! possible orderings of 3 elements. A comparison tree for sorting n elements will have n! leaves. In this example there are sets which will be sorted using only 2 comparisons, but there are also sets for which this algorithm uses 3 comparisons.

Let us define the depth of the root of a tree as depth 0, and the depth of a node in the tree as 1 + depth of its parent node. Then the maximum number of comparisons to sort any set using a particular comparison tree algorithm is the maximum depth of a node in the tree. In the example, there is a node (a leaf) of depth 3 and there is a set on which this algorithm uses 3 comparisons.

To obtain a worst case lower bound, we will consider the depth of comparison trees for sorting. Since there are only two arrows from each node, the number of nodes can only double when the depth is increased by 1. So there are at most 2^D nodes at depth D, and summing the nodes at every depth there are at most 2^{D+1} nodes in a tree with maximum depth D. Since there are n! possible orderings of n things, a comparison tree for sorting must have at least n! leaves. From these two facts we have $2^{D+1} \geq n!$.

Taking logs and using Stirling's approximation that $n! \sim n^n e^{-n} \sqrt{2\pi n}$, we have

$$D \geq \theta(n \log n),$$

and hence $\theta(n \log n)$ is a worst case lower bound for the number of comparisons used by an algorithm which sorts using only comparisons.

The comparison tree model can also be used to produce an average case lower bound. The average number of comparisons will be the average depth of a leaf of the comparison tree. By average here we mean that we assume that each ordering of the n elements is equally likely and hence each ordering has probability 1/n! Now the average depth of a leaf will be at least as great as the average depth of a node. To minimize the average depth of a node, we will consider the full tree with total depth L where $\log n! - 1 < L \leq \log n!$. By a full tree we mean a tree which has exactly 2^i nodes at depth i. This full tree will give us a lower bound because any comparison tree for sorting has at least log n! nodes and we are making their depths as small as possible. The average depth of a node in our full tree is at least

$$\frac{1}{n!} \sum_{i=0}^{L} i2^i = \frac{2}{n!} \{(L-1)\ 2^L + 1\} \geq \frac{2}{n!} \{(\log n! - 2)\ \frac{n!}{2} + 1\}.$$

Again using Stirling's approximation this is asymptotic to $\theta(n \log n)$, and $\theta(n \log n)$ is an average case lower bound for sorting. We summarize these results in the following proposition.

PROPOSITION: Any algorithm which sorts using only comparisons must use at least

a) $\theta(n)$ comparisons in best case

b) $\theta(n \log n)$ comparisons in average case

c) $\theta(n \log n)$ comparisons in worst case.

We close this section with two reminders. One, average case here assumes that all n! orderings are equally likely. If there are significantly fewer orderings which are very likely to occur, then the above average case bound does not hold. For

example, if the inputs are almost-sorted then there are $\theta(n)$ average case algorithms. Two, the bounds in the proposition apply for algorithms which sort using only comparisons. If different types of operations are allowed, then these bounds may not hold.

5.4 Exercises

a) Prove a worst case lower bound of $\frac{3}{2}$ n-2 comparisons for any algorithm which finds the largest and smallest elements in an array by using only comparisons. Devise a divide-and-conquer algorithm for this problem. Show that your algorithm uses $\frac{3}{2}$ n-2 comparisons.

b) Create an algorithm which finds the largest and second largest elements in an array and uses only n + log n-2 comparisons. (HINT: Keep track of the elements which are potentially the largest, and for each such element keep track of those elements which have 'lost' only to this element.)

6. EXHAUSTIVE SEARCH

6.1 Straightforward Exhaustive Search

The algorithms which we have considered so far have been based on the divide-and-conquer strategy, that is, try to break the problem into several smaller problems of the same kind. An exhaustive search is a different strategy for designing algorithms. This strategy is based on the idea of trying all possible answers. Either the solution is located or no solution exists.

A simple problem for which this strategy is reasonable is: given a list, and a value, is there an element of the list which contains the given value? An exhaustive search algorithm would look at each element in turn until either one with the given value is found or until all of the elements have been viewed. If there are n elements in the list, then this algorithm will take $\theta(1)$ in best case and $\theta(n)$ in worst case, and these are the best possible running times for any algorithm for this problem.

There are also problems for which straightforward exhaustive search gives very poor algorithms. Consider sorting. A straightforward exhaustive search would try each possible ordering until the sorted ordering is found. Unfortunately, there are n! possible orderings so in worst case (when the last ordering is the correct one) this algorithm will take at least $\theta(n!)$, which is much, much greater than the $\theta(n \log n)$ taken by algorithms we have already found.

Satisfiability is another problem to which exhaustive search can be applied. The satisfiability problem is: given a Boolean expression, is there an assignment of true and false to the variables which makes the expression true? If there are n variables, there are 2^n possible true/false assignments for the variables, and since a Boolean expression of length L can be evaluated in $\theta(L)$ time, the exhaustive search algorithm will take $\theta(L\ 2^n)$ time in worst case. Unlike sorting, it is not clear that there are

faster algorithms for this problem. Also unlike the find value problem, it is not clear that this is the best running time possible for this problem.

6.2 Backtracking

While straightforward exhaustive search may be useful in some problems, there is usually additional information available which will allow an algorithm to eliminate some of the possibilities. One way to make use of this additional information is called backtracking. In a backtrack algorithm the search is broken into stages, and at each stage only the possibilities which can still lead to a solution are considered. The name backtracking comes from a method for finding a way through a maze. At a choice-point in the maze, pick an arbitrary path which has not yet been used. Continue making such choices until you reach a dead end or a choice-point at which all paths have already been used. Then go back on the path you have come down to the previous choice-point and continue the method. This last step, going back on a path, is called backtracking, and this name is also given to the general method. So in the general backtracking method the search is broken into stages; at each stage a choice is made; when the algorithm determines that none of the choices at a stage can lead to a solution, then the algorithm backtracks by returning to the previous stage and taking one of the unused choices.

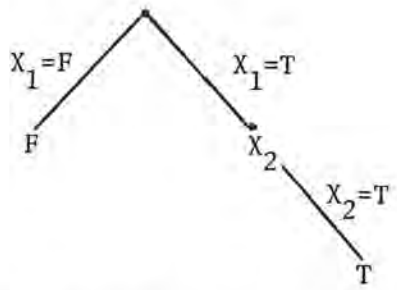
A problem to which backtracking can be directly applied is Hamiltonian path: given a graph, find a sequence of vertices which contains each vertex exactly once and such that adjacent vertices in the sequence are connected by an edge in the graph. At each stage in the backtrack algorithm an unused vertex sharing an edge with the current vertex is chosen. If it is impossible to choose such a vertex, the algorithm backtracks to the previous vertex and tries to choose a vertex other than the one that led to the dead end. If the graph contains a Hamiltonian path, this algorithm will eventually find it, because the algorithm generates all sequences of

vertices with adjacent vertices connected by an edge and no repeated vertices. A straightforward exhaustive search for this problem would generate a permutation of the vertices and then test to see if the vertices adjacent in the permutation were also adjacent in the graph. In worst case all permutations would be generated and the exhaustive search algorithm would take at least n! time. The backtrack algorithm in worst case would generate at most $\prod_{i=1}^{n} d_i$ paths, where d_i is the degree of vertex i. So in worst case we expect the backtrack algorithm to be faster than the exhaustive search algorithm. The backtrack algorithm also has the advantage that it uses information about the graph in generating a path. If the backtract algorithm uses more information about the graph in picking the next vertex, we can reasonably expect the backtrack algorithm to work much more quickly in average case.

We will next consider backtracking and exhaustive search for the satisfiability problem. Consider the Boolean expression: $X_1 \cdot (X_1 \vee \overline{X}_2 \vee \overline{X}_3) \cdot (\overline{X} \vee X_2)$. Since this expression has 3 variables, there are 8 possible truth assignments, which we show in the following table:

X_1	X_2	X_3	EXPRESSION
F	F	F	F
F	F	T	F
F	T	F	F
F	T	T	F
T	F	F	F
T	F	T	F
T	T	F	T
T	T	T	T

An exhaustive search algorithm would generate 7 out of 8 of these assignments before it found that the expression was satisfiable. A backtrack algorithm could proceed according to the following tree:



It picks the first variable X_1 and assigns it F, then X_1=F is substituted in the expression yielding the expression F. Since F is not satisfiable, the algorithm backtracks and tries the assignment X_1=T. When this is substituted into the expression, it yields the expression X_2. A reasonably clever algorithm would notice that this expression can be satisfied by the assignment X_2=T, and hence that the whole expression is satisfiable. A less clever algorithm might try X_2=F, and have to backtrack once more before finding a satisfying assignment. In either case, it seems that the backtrack algorithm will be better than the exhaustive search algorithm.

The backtrack algorithm should simplify the expression at each stage. While this might seem complicated in general, it is very easy when the expression is in clause form. In clause form a Boolean expression is a set of clauses AND-ed together, and within each clause are a set of literals (that is, complemented and uncomplemented variables) OR-ed together. When a variable X is assigned the value T, all the clauses containing the literal X become true and can be removed from the expression, while all the clauses containing $\overline{X}$ are simplified by the removal of $\overline{X}$. Similarly, if X is assigned the value F, then each clause containing $\overline{X}$ can be removed from the expression and each clause containing X can be simplified by the removal of X. One special case should be noted: if a clause containing a single variable has that variable removed, then the whole expression becomes false.

How should the backtrack algorithm choose which variable to assign at each stage? An algorithm could try assigning X_1, then X_2, and so forth, but this method does not take advantage of the structure of the expression. If an expression con-

tains a clause with a single literal, then that literal must be set to true for the entire expression to be true. If an expression contains a variable X which only appears in the form X and never in the form $\overline{X}$, then X can be assigned true and all the clauses containing X can be removed without affecting the satisfiability of the expression. These two rules can be used to simplify the expression. There is one complication: if the expression contains a clause with the single literal X and another clause with a single literal $\overline{X}$, then the expression is not satisfiable.

After the expression has been simplified as much as possible by the above rules, a method for choosing which assignment to make next is still needed. A method which seems reasonable is to pick the literal which appears in the most clauses and to make this literal true. This method is not guaranteed to find a satisfying assignment, but it will simplify the expression. Such a method which is reasonable but not guaranteed to lead to a solution is called a heuristic. A backtrack algorithm will still have to backtrack when the heuristic does not lead to a solution, but a well-chosen heuristic can greatly reduce the number of times the algorithm has to backtrack.

6.3 Knight's Tour

We have already mentioned the Hamiltonian path problem. A specialization of this problem is the knight's tour problem. In the knight's tour problem, the vertices of the graph are the squares of an n x m chessboard. Two squares are connected by an edge iff a knight can legally move from one square to the other. The knight can move two squares in one direction and one square in an orthogonal direction. The knight's tour problem is to find a Hamiltonian path in this graph, or equivalently move the knight legally around the chessboard so that it visits each square exactly once.

If nm is odd then from some starting squares there is no knight's tour. The traditional chessboard has squares of two colors, so that orthogonally adjacent squares are of different colors. If nm is odd, then there are more squares of one color. Since the knight always moves from one color to the other, a tour starting from a square of the less numerous color is impossible.

To find a knight's tour one could use the backtrack algorithm for Hamiltonian path, but as the last section suggests,some sort of heuristic should be used to choose the next square. One plausible idea is based on the degree of a square. The degree of a square is the number of unused squares the knight could go to if the knight were on this square. If the square has large degree then there are many ways into and out of the square, and it is not too important to deal with this square immediately. If a square has degree 2,then there is one way in and one way out of the square, and this square should probably be used as soon as possible. So the heuristic is: next visit a possible square of lowest degree.

Even without this heuristic a backtrack algorithm will find a knight's tour if one exists. With this heuristic,the backtrack algorithm may work faster. As well as picking the next square the degree can also be used in deciding when to backtrack. Clearly a backtrack is called for when the tour reaches a dead end, a square from which no unused square can be reached in one step. Just before the move into this dead end,the dead end square had degree 1. A square of degree 1 should be entered only if it is the last square on the board. So if the lowest degree among possible next squares in 1 and there is more than one unused square,then the algorithm should backtrack. Further,if there are ever two unused squares of degree 1,then the algorithm should backtrack since both of these squares cannot be the last square.

Initially squares near the center of the board will have degree 8, squares nearer an edge of board will have degree less than 8, and the corner squares will have degree 2. As a partial tour is constructed the degrees of the squares will decrease.

How well will this heuristic work? Is backtracking ever needed? We will answer these questions in a moment, but first we want to consider a special case. In the special case n = m = 4k + 1, for k a positive integer, and the tour starts in one of the corner squares. For this special case a knight's tour can be found using a simple rule with no backtracking. The simple rule is: choose as the next square the possible square nearest the edge of the board. The following picture shows a 5 x 5 board with each square containing a number indicating the order in which the knight visits the squares.

1	14	9	20	3
24	19	2	15	10
13	8	25	4	21
18	23	6	11	16
7	12	17	22	5

Notice that the knight starts in a corner and finishes the tour at the center of the board. It is clear that the simple rule needs some amplification. How did the knight at the square numbered 2 decide to go to the square numbered 3 rather than the square numbered 21 or the square numbered 13, and how did the knight at square 8 decide to go to square 9 rather than square 17? The extra tie-breaking rules are: choose the square with lowest initial degree, choose the first square in clockwise order. With these rules one can prove inductively that a knight's tour will be found in the special case. The induction considers a 4(k+1) + 1 x 4(k+1) + 1 board as a 4k+1 x 4k+1 bound surrounded by a two square border.

From the special case, we can see that in the general case we will also need tie-breaking rules to determine which square should be picked when two possible next squares have the same degree. From the special case, we could try the rule which says: choose the first such square in clockwise order. Unfortunately, it can be shown that for some starting squares this tie-breaking rule does not give a knight's tour.

Another possible tie-breaking rule is to "look-ahead"; choose the next possible square of lowest degree which has the lowest degree following square. But again this tie-breaking rule may fail. Further look-ahead would be possible but it would take considerably more computation.

In spite of the fact that the heuristic even with tie-breaking rules does not work in all cases, it still works in a large majority of cases. Some empirical studies show that a backtracking algorithm using the heuristic uses no backtracks in most cases, one backtrack in a few cases, and 2 backtracks in rare cases, and never uses more than 2 backtracks. A proof of any of these empirical results would be interesting, but it is probably very difficult.

When heuristics work so well on a problem one should consider whether there is a reasonable non-exhaustive algorithm for the problem. Cull & DeCurtins[1978] proved that if $\min(n, m) \geq 5$, then there is a knight's tour from at least one starting square on an n x m bound, and if nm is even there is a knight's tour starting from any square on the board. Their proof technique is constructive and takes $\theta(nm)$ time to construct the knight's tour. Their paper does not contain a proof that if nm is odd then there is a knight's tour starting from every square of the more numerous color.

6.4 The n-Queens Problem

Backtracking is an effective method for the n-Queens problem. Can n queens be placed on an n x n board so that no queen can take another queen: that is, can n objects be placed on an n x n board so that no two objects are in the same row, the same column, or the same diagonal? The answer to this questions is "no" when n = 2 or 3, and "yes" otherwise. The problem becomes more difficult when one is asked for all the different ways to place n nontaking queens on an n x n board. Two ways are different if one cannot be obtained from the other by rotating and/or reflecting the board.

Since one queen will be placed in each row, we will take placing a queen in a row as a stage in a backtracking algorithm. The algorithm starts by placing a queen in column 1 of row 1, then it successively places queens in the lowest numbered allowed column of each successive row, until either every row has a queen or there is no allowed place for a queen in the present row. If there is no allowed place to put a queen in the present row, then the algorithm backtracks and tries to place a queen in the next allowed place in the previous row. If a queen has been placed in each row then this solution is output and the algorithm backtracks. The algorithm terminates when it has backtracked to row 1 and the queen in row 1 is in column n. This algorithm can be written as either a recursive or as an iterative program. Since the algorithm is tail-recursive, we suggest writing it as an iterative algorithm. We leave the actual program as an exercise and refer the reader to Wirth[1976].

The algorithm we have outlined will produce all allowed placements, not all different allowed placements. We still need a method which determines if two placements are different. Two placements are the same if one is the rotation-reflection of the other. The rotation by 45^{o} and the reflection generate the 8-element group of the symmetries of a square. By applying each element of this group to a placement, one will obtain the 8 placements which are equivalent. If one had a table of the different placements found so far, then one could look in the table and see if any of these 8 equivalent placements had already been found. But there is an easier way. Since a placement will consist of n numbers indicating the columns in which each of the n queens are placed, and each column number will be between 1 and n, a solution can be viewed as a single number in base n. The algorithm is set up so that it generates the placements in numeric order. Thus to see if a placement is different, apply to it each of the 7 nonidentity elements of the group and determine if any of these equivalent placements gives a lower number. If one of these equivalent placements gives a lower number, then this placement is equivalent to an already generated

placement. If each of the 7 equivalent placements gives a number greater than or equal to the present placement, then the present placement is different from all previous placements.

How fast is this backtrack algorithm for n-queens? A straightforward exhaustive search would generate all permutations of n things and hence take about n! time. Empirical studies suggest that the backtrack algorithm runs in time proportional to $(n/2)^{n/2}$. This is much much smaller than n!. A proof that this algorithm runs in the time suggested by the empirical studies would be very interesting.

6.5 Exercises

a) Use the heuristics of section 6.3 to construct a knight's tour of a 6 x 6 board.

b) Pick starting squares and show that the heuristics of section 6.3 do not find a knight's tour of a 5 x 5 board even when such a tour is possible.

c) Determine all the different placements of 8 nontaking queens on an 8 x 8 board.

7. HARD PROBLEMS

7.1 Classification of Algorithms and Problems

We have encountered various algorithms,particularly of the divide-and-conquer type,which have running times like $\theta(n)$, $\theta(n \log n)$, and $\theta(n^3)$, where n is some measure of the size of the input. We have also encountered algorithms,particularly of the exhaustive search type,which have running times like $\theta(2^n)$ and $\theta(n!)$. For algorithms of the first type,doubling the size of the input increases the running time by a constant factor, while for algorithms of the second type,doubling n increases the running time by a factor proportional to the running time. If we double the speed of the computer we are using, then the largest input size which our computer can solve in a given time will increase by a constant factor if we have an algorithm of the first type; for an algorithm of the second type, the input size our computer can solve will only increase by an additive constant, at best. These considerations led Edmonds to propose that algorithms of the first type are computationally reasonable, while algorithms of the second type are computationally unreasonable. More specifically,he suggested the definitions:

reasonable algorithm: an algorithm whose running time is bounded by a polynomial in the size of the input.

unreasonable algorithm: an algorithm whose running time cannot be bounded by any polynomial in the size of the input.

This suggests that we should try to replace unreasonable algorithms by reasonable algorithms. Unfortunately,this goal is not always attainable. For the Towers of Hanoi problem,any algorithm must have running time at least $\theta(2^n)$. Since some problems do not have reasonable algorithms, we should classify problems as well as algorithms. Corresponding to Edmonds definition for algorithms, Cook and Karp suggested the following definition for problems:

easy problem: a problem which has a polynomial time bounded algorithm.

hard problem: a problem for which there is no polynomial time bounded algorithm.

An easy problem may have unreasonable algorithms. For example, we have seen an $\theta(n!)$ exhaustive search algorithm for sorting, but sorting is an easy problem because we also have an $\theta(n \log n)$ sorting algorithm. A hard problem, on the other hand, can never have a reasonable algorithm.

Some problems like Towers of Hanoi are hard for the trivial reason that their output is too large. To avoid such output-bound problems, Cook suggested considering only yes/no problems; that is, problems whose output is limited to be either yes or no. One such yes/no problem is the variant of the Towers of Hanoi problem in which the input is a configuration and the question is: is this configuration used in moving the disks from tower A to tower C? Furthermore, this variant is an easy problem. Usual easy problems can be transformed into easy yes/no problems by giving as the instance of the yes/no problem both the input and the output of the usual problem and asking if the output is correct. For example, sorting can be converted into a yes/no problem when we give both the input and the output and ask if the output is the input in sorted order. As another example, matrix multiplication can be converted into a yes/no problem in which we give three matrices A, B, and C and ask if C=AB. There are also other ways to transform usual problems into yes/no problems. Examples are the above variant of the Towers of Hanoi, and the variant of sorting in which we ask if the input is in sorted order.

To avoid problems which are hard only because of the length of their output, Cook defined:

P = the class of yes/no problems which have polynomial time algorithms. (Some authors call this class PTIME.)

For many problems, it is easy to show that they are in P. One gives an algorithm for the problem and demonstrates a polynomial upper bound on its running time. It may be

quite difficult to show that a problem is not in P , but it is clear that there are problems which are not in P. For example, the halting problem, which asks if an algorithm ever terminates when given a particular input, is not in P because the halting problem has no algorithm.

Classification of problems would not be very useful if we only could say that some problems have polynomial time algorithms and some problems have no algorithms. We would like a finer classification, particularly one that helps classify problems which arise in practice. In the next section we introduce some machinery which is needed for a finer classification.

7.2 Nondeterministic Algorithms

Up to this point we have discussed only deterministic algorithms. In a deterministic algorithm there are no choices; the result of an instruction determines which instruction will be executed next. In a nondeterministic algorithm choices are allowed; any one of a set of instructions may be executed next. Nondeterministic algorithms can be viewed as "magic" : if there is a correct choice,the magic forces the nondeterministic algorithm to make this correct choice. A less magic view is that if there are several possibilities the nondeterministic algorithm does all of them in parallel. Since the number of possibilities multiply at each choice-point, there may be arbitrarily many possibilities being executed at once. Therefore, a nondeterministic algorithm gives us unbounded parallelism.

This view of nondeterminism as unbounded parallelism makes clear that nondeterminism does not take us out of the realm of things which can be computed deterministically, because we could build a deterministic algorithm which simulates the nondeterministic algorithm by keeping track of all the possibilities. However, a nondeterministic algorithm may be faster than any deterministic algorithm for the same problem. We define the time taken by a nondeterministic algorithm as the fewest

instructions the nondeterministic algorithm needs to execute to reach an answer. (This definition is not precise since what an instruction means is undefined. This could be made precise by choosing a model of computation like the Turing machine in which an instruction has a well-defined meaning, but this imprecise definition should suffice for our purposes.) With this definition of time, a nondeterministic algorithm could sort in $\theta(n)$ time because it always makes the right choice, whereas a deterministic algorithm would take at least $\theta(n \log n)$ time in some cases. In some sense we are comparing the best case of the nondeterministic algorithm with the worst case of the deterministic algorithm, so it is not surprising that the nondeterministic algorithm is faster.

For yes/no problems we give nondeterministic algorithms even more of an edge. We divide the inputs into yes-instances and no-instances. The yes-instances eventually lead to yes answers. The no-instances always lead to no answers. We assume that our nondeterministic algorithms cannot lead to yes as a result of some choices, and to a no as a result of some other choices. For a yes-instance, the running time of a nondeterministic algorithm is the fewest instructions the algorithm needs to execute to reach a yes answer. The running time of the nondeterministic algorithm is the maximum over all yes-instances of the running time of the algorithm for the yes-instances. We ignore what the nondeterministic algorithm does in no-instances.

As an example of nondeterministic time, consider the yes/no problem: given a set of n numbers each containing log n bits, are there two identical numbers in the set? In a yes-instance, a nondeterministic algorithm could guess which two numbers were identical and then check the bits of the two numbers, so the running time for this nondeterministic algorithm is $\theta(\log n)$. On the other hand, even in a yes-instance, a deterministic algorithm would have to look at almost all the bits in worst case, so the running time for any deterministic algorithm is at least $\theta(n \log n)$.

The definition of nondeterministic time may seem strange, but it does measure an interesting quantity. If we consider a yes/no problem, the yes-instances are all

those objects which have a particular property. The nondeterministic time is the length of a proof that an object has a certain property. We ignore no-instances because we are not interested in the lengths of proofs that an object does not have the property.

7.3 NP and Reducibility

Now that we have a definition of the time used by a nondeterministic algorithm, we can, in analogy with the class P, define

NP = the class of yes/no problems which have polynomial time nondeterministic algorithms.

It is immediate from this definition that $P \leq NP$, because every problem in P has a deterministic polynomial time algorithm and we can consider a deterministic algorithm as a nondeterministic algorithm which has exactly one choice at each step. It is not clear, however, whether P is properly contained in NP or whether P is equal to NP.

Are there problems in NP which may not be in P? Consider the yes/no version of satisfiability: given a Boolean expression, is there an assignment of true and false to the variables which makes the expression true? This problem is in NP because if there is a satisfying assignment we could guess the assignment, and in time proportional to the length of the expression we could evaluate the expression and show that the expression is true. We discussed exhaustive algorithms for satisfiability because these seem to be the fastest deterministic algorithms for satisfiability. No polynomial time deterministic algorithm for satisfiability is known. Similarly, the problem: given a graph does it contain a Hamiltonian path?, seems to be in NP but not in P. Hamiltonian path is in NP because we can guess the Hamiltonian path and quickly (i.e., in polynomial time) check to see if the guessed path really is a Hamiltonian path. Hamiltonian path does not seem to be in P, because exhaustive search

algorithms seem to be the fastest deterministic algorithms for this problem and these search algorithms have worst case running times which are at least $\theta(2^n)$, and hence their running times cannot be bounded by any polynomial.

Another problem which is in NP but may not be in P is composite number: given a positive integer n, is n the product of two positive integers which are both greater than 1? Clearly this is in NP because we could guess the two factors, multiply them, and show that their product is n. Why isn't this problem clearly in P? Everyone knows the algorithm which has running time at most $\theta(n^2)$ and either finds the factors or reports that there are no factors. This well-known algorithm simply tries to divide n by 2 and by each odd number from 3 to n-1. While this algorithm is correct, its running time is not bounded by a polynomial in the size of the input. Since n can be represented in binary, or in some other base, the size of the input is only log n bits, and n cannot be bounded by any polynomial in log n. This example points out that we have been too loose about the meaning of size of input. The official definition says that the size of the input is the number of bits used to represent the input. According to the official definition, the size of the input for this problem is log n if n is represented in binary. But if n were represented in unary then the size of the input would be n. So the representation of the input can affect the classification of the problem.

There are a great variety of problems which are known to be in NP, but are not known to be in P. Some of these problems may not seem to be in NP because they are optimization problems rather than yes/no problems. An example of this kind of optimization problem is the traveling salesman problem: given a set of cities and distances between them, what is the length of the shortest circuit which visits each city exactly once and returns to the starting city? This optimization problem can be changed into a yes/no problem by giving an integer B as part of the input. The yes/no question becomes: is there a circuit which visits each city exactly once and

returns to the starting city and has length at most B? At first glance the optimation problem seems harder than the yes/no problem, because we can solve the yes/no problem by solving the optimization problem, and solving the yes/no problem does not give a solution to the optimization problem. But we can use the yes/no problem to solve the optimization problem. Set B to n times the largest distance; then the answer to the yes/no problem is yes. Set STEP to B/2. Now set B to B-STEP, and STEP to STEP/2. Now do the yes/no problem with this new B and this new STEP. If the answer is yes, set B to B-STEP and STEP to STEP/2. If the answer is no, set B to B+STEP and STEP to STEP/2. Continue this process until STEP=0. The last value of B will be the solution to the optimization problem. How long will this take? Since STEP is halved at each call to the yes/no procedure, the number of calls will be the log of the initial value of STEP. But STEP is initially n times the largest distance, so the number of calls is log n + log(largest distance) which is less than the size of the input. Thus the optimization problem can be solved by solving the yes/no problem a number of times which is less than the length of the input.

This example suggests the idea that two problems are equally hard if both of them can be solved in polynomial time if either one of them can be solved in polynomial time. This equivalence relation on problems also suggests a partial ordering of problems. A problem A is no harder than problem B if a polynomial time deterministic algorithm for B can be used to construct a polynomial time deterministic algorithm for A. We symbolize this relation by $A \leq B$. If A is the yes/no traveling salesman problem, and B is the traveling salesman optimization problem, then we have both $A \leq B$ and $B \leq A$. If A and B are any two problems in P then we have both $A \leq B$ and $B \leq A$, because we could take the polynomial time deterministic algorithm for one problem and make it a subroutine of the polynomial deterministic algorithm for the other problem and never call the subroutine. While in these examples, the relation $\leq$ works both ways, there are cases in which $\leq$ only works one way. For example, let

A be any problem in P, and let HALT be the halting problem; then $A \leq HALT$, but $HALT \not\leq A$, because we don't need an algorithm for HALT to construct a polynomial time algorithm for A, and the polynomial time algorithm for A cannot help in constructing any algorithm for HALT,let alone a polynomial time algorithm for HALT.

This relation $A \leq B$ which we are calling A is no harder than B, is usually called polynomial time reducibility,and is read A is polynomial time reducible to B. There are many other definitions of reducibility in the literature. We refer the interested reader to Garey and Johnson[1979] and Hartley Rogers[1967].

The notion of a partial ordering on problems should aid us in our task of classifying problems. In particular, it may aid us in saying that two problems in NP are equally hard. Further, it suggests the question: is there a hardest problem in NP? We consider this question in the next section.

7.4 NP-Complete Problems

In this section, we will consider the relation $\leq$ defined in the last section, and answer the question: is there a hardest problem in NP? Since we have a partial order $\leq$, we might think of two very standard instances of partial orders: the partial (and total) ordering of the integers, which has no maximal element; and the partial ordering of sets, which has a maximal element. From these examples, we see that our question cannot be answered on the basis that we have a partial order. If we consider $\leq$ applied to problems, is there a hardest problem? The answer is no, because the Cantor diagonal proof always allows us to create harder problems. On the other hand, one may recall that the halting problem is the hardest recursively enumerable (RE) problem. So on the analogy with RE, there may be a hardest problem in NP. Cook [1971] proved that there is a hardest problem in NP. A problem which is the hardest problem in NP is called an NP-complete problem. Cook proved the more specific result:

Cook's Theorem: Satisfiability is NP-complete.

The proof of this theorem requires an exact definition of nondeterministic algorithm; since we have avoided exact definitions,we will only be able to give a sketch of the proof. We refer the interested reader to Garey and Johnson[1979] for the details. The basic idea of the proof is to take any instance of a problem in NP which consists of a nondeterministic algorithm, a polynomial that gives the bound on the nondeterministic running time, and an input for the algorithm, and to show how to construct a Boolean expression which is satisfiable iff the nondeterminist algorithm reaches a yes answer within the number of steps specified by the polynomial applied to the size of the input. The construction proceeds by creating clauses which can be interpreted to mean that at step 0 the algorithm is in its proper initial state. Then for each step, a set of clauses are constructed which can be interpreted to mean that the state of the algorithm and the contents of the memory are well-defined at this step. Further, for each step a set of clauses are constructed which can be interpreted to mean that the state of the algorithm and the contents of memory at this step follow from the state and contents at the previous step by an allowed instruction of the algorithm. Finally, some clauses are constructed which can be interpreted to mean that the algorithm has reached a yes answer. The polynomial time bound is used to show that the length of this Boolean expression is bounded by a polynomial in the length of the input.

An interesting consequence of the proof is that satisfiability of Boolean expressions in clause form is NP-complete. This result can be refined to show that satisfiability in clause form with exactly 3 literals per each clause is also NP-complete.

After Cook's result, Karp[1972] quickly showed that a few dozen other standard problems are NP-complete. Garey and Johnson's book contains several hundred NP-complete problems. Johnson also writes a column for the Journal of Algorithms which contains even more information on NP-complete problems.

Why is this business of NP-complete problems so interesting? The NP-complete problems are the hardest problems in NP in the sense that for any problem A in NP, $A \leq$ NP-complete. So if there were a polynomial time deterministic algorithm for any NP-complete problem, there would be a polynomial time deterministic problem for any problem in NP; that is, P and NP would be the same class. Conversely, if $P \neq NP$, then there is no point to looking for a polynomial time deterministic algorithm for an NP-complete problem. Simply knowing that a problem is in NP without knowing that it is NP-complete leaves open the question of whether or not the problem has a polynomial time bounded algorithm even on the supposition that $P \neq NP$. The fact that a number of NP-complete problems have been well known problems for several hundred years and no one has managed to find a reasonable algorithm for any one of them suggests to most people that $P \neq NP$ and that the NP-complete problems really are hard.

One of the virtues of Cook's theorem is that to show that an NP problem A is NP-complete you only have to show that satisfiability $\leq A$. As a catalog of NP-complete problems is built, the task of showing that an NP problem A is NP-complete gets easier because you only have to pick some NP-complete problem B and show that $B \leq A$.

We have already mentioned the traveling salesman problem (TSP). This problem is NP-complete. Let us show that if Hamiltonian circuit is NP-complete then TSP is NP-complete. The Hamiltonian circuit problem (HC) is: given a graph, is there a circuit which contains each vertex exactly once and uses only edges in the graph? Given an instance of HC, we create an instance of TSP by letting each vertex from HC become a city for TSP, and defining the distance between cities by $d(i, j) = 1$ if there is an edge in HC between vertices i and j, and $d(i, j) = 2$ if there is no such edge. Now if there were n vertices in HC, we use $B = n$ as our bound for TSP. If the answer to TSP is yes, then there is a circuit of length n, but this means that the circuit

can only contain edges of distance 1 and hence this circuit is also a Hamiltonian circuit of the original graph. Conversely, if there is a Hamiltonian circuit, then there is a TSP circuit of length n. To complete our proof, we must make sure that this transformation from HC to TSP can be accomplished in polynomial time in the size of the instance of HC. Since HC has n vertices and TSP can be specified by giving the $n(n-1)/2$ distances between the n cities, we only have to check for each of the $n(n-1)/2$ distances whether or not it corresponds to an edge in the original graph. Even with a very simple algorithm this can be done in at worst $\theta(n^4)$, which is bounded by a polynomial in the size of the HC instance.

This is a very simple example of proving that one problem in NP is NP-complete by reducing a known NP-complete problem to the problem. We refer the reader to Garey and Johnson[1979] for more complicated examples.

7.5 Dealing with Hard Problems

In the theoretical world there are hard problems. Some of these hard problems are NP-complete problems and there are other problems which are harder than NP-complete problems. How can these hard problems be handled in the real world?

The simplest way to handle hard problems is to ignore them. Many practical programmers do not know what hard problems are. Their programming involves tasks like billing and payroll which are theoretically trivial, but practically quite important. Ignoring hard problems may be a reasonable strategy for these programmers.

Another way to handle hard problems is to avoid them. To avoid hard problems, you have to know what they are. One of the major virtues of lists of NP-complete problems is that they help the programmer to identify hard problems and to point out that no reasonable algorithms for these problems are known. It is often unfortunately the case that a programmer is approached with a request for a program and the requester has tried to remove all the specific information about the problem and generalize

the problem as much as possible. Over-generalization can make a problem very hard. If the programmer can get the specific information, he may be able to design a reasonable algorithm for the real problem and avoid the hard generalization.

Sometimes real problems are really hard but not too big. For example, many real scheduling problems turn out to be traveling salesman problems with 30 to 50 cities. For these situations an exhaustive algorithm may still solve the problem in reasonable time. The programmer should still try to tune the algorithm to take advantage of any special structure in the problem, and to take advantage of the instructions of the actual computer which will be used. Exhaustive search is a way to handle some NP-complete problems when the size of the input is not too large.

Heuristics are another way to deal with hard problems. We have already mentioned heuristics in discussing backtrack algorithms. A heuristic is a method to solve a problem which doesn't always work. To be useful a heuristic should work quickly when it does work. The use of heuristics is based on the not unreasonable belief that the real world is usually not as complicated as the worst case in the theoretical world. This belief is supported by the observation that creatures which seem to have less computing power than computers can make a reasonable living in the real world. Artificial intelligence has been using heuristics for years to solve problems like satisfiability. These heuristics seem to be very effective on the instances of satisfiability which arise in artificial intelligence contexts. Heuristics are also widely used in the design of operating systems. Occasional failures in these heuristics lead to software crashes. Since we usually see only a couple of such crashes per year these heuristics seem to be very effective.

Sometimes the behavior of heuristics can be quantified so that we can talk about the probability of the heuristic being correct. For example, a heuristic to find the largest element in an n element array is to find the largest element among the first n-1 elements. If the elements of the array are in random order, then this heuristic

fails with probability 1/n, and as $n \to \infty$ the probability that this heuristic gives the correct answer goes to 1. Such probabilistic algorithms are now being used for a wide variety of hard problems. For example, large primes are needed for cryptographic purposes. While it seems to be hard to discover large primes, there are tests which are used so that if a number passes all the tests,then the number is probably a prime.

Many hard problems can be stated as optimization problems: find the smallest or largest something which has a particular property. While actually finding the optimum may be difficult, it may be much easier to find something which is close to optimum. For example, in designing a computer circuit one would like the circuit with the fewest gates which carries out a particular computation. This optimization problem is hard. But from a practical point of view,no great disaster would occur if you designed a circuit with 10% more gates than the optimum circuit. For various hard problems,approximation algorithms have been produced which produce answers close to the optimum answer. We will consider an example of approximation in the next section.

7.6 Approximation Algorithms

Let us consider the traveling salesman optimization problem: given a set of cities and distances between them,find the shortest circuit which contains each city exactly once. This problem arises in many real scheduling situations. We will try to approximate the shortest circuit.

To make an approximation possible, we will assume that the distances behave like real distances, that is, the distances obey the triangle inequality $d(i, j) \leq d(i, k) + d(k, j)$, so that the distance from city i to city j is no longer than the distance from city i to city k plus the distance from city k to city j.

A simpler task than finding the minimum circuit is finding the minimum spanning

tree. The minimum spanning tree is a set of links which connects all the cities and has smallest sum of distances. In the minimum spanning tree, the cities are not all directly connected; several links may have to be traversed to get from city i to city j. There is a reasonable algorithm for the minimum spanning tree because the shortest link is in this tree. So one can proceed to find this tree by putting in the shortest link and continuing to add the shortest link which does not complete a cycle.

A circuit can be constructed from the minimum spanning tree by starting at some city and traversing the links of the tree to visit every other city and return to the starting city. This circuit is twice as long as the sum of the distances of the links in the minimum spanning tree. But this circuit may visit some cities more than once. To "clean up" this circuit, we use this circuit while no city is repeated and, if city j is the first repeated city and if city i is the city before city j and if city k is the next city after city j which has not yet been visited, we connect city i to city k. We continue to use this procedure to produce a circuit in which each city is visited exactly once. From the triangle inequality we have that the length of this new circuit is at most as long as the circuit with cities repeated, and hence that this new circuit is no longer than twice the length of the minimum spanning tree.

The optimum circuit must be at least as long as the minimum spanning tree because the optimum circuit connects every city. Thus we have

$$OPT \leq ALG \leq 2\ OPT$$

where OPT is the length of the optimum circuit and ALG is the length of the circuit produced by the approximation algorithm.

It would be pleasant if all hard optimization problems had approximation algorithms. Unfortunately this is not the case. We really needed the triangle inequality to produce an approximation for the traveling salesman problem. Consider an

instance of Hamiltonian circuit with n vertices. We can convert this to an instance of traveling salesman without triangle inequality by assigning distance 1 to all the edges which are in the original graph, and assigning distance n + 2 to all the edges which were not in the original graph. Now if there were a Hamiltonian circuit in the original graph, then there would be a traveling salesman circuit of length n. If we could approximate this traveling salesman problem within a factor of 2, the traveling salesman circuit would have length at most 2n exactly when the original graph had a Hamiltonian circuit,because if the traveling salesman circuit used even one of the edges not in the original graph,it would have length at least 2n + 1. So approximating the traveling salesman problem without triangle inequality is as hard as Hamiltonian circuit.

This example can be generalized to show that no approximation within a factor of f(n) is possible by assigning each edge not in the graph a distance greater than f(n) - (n-1). To make sure the transformation can be carried out in polynomial time in the size of the instance of Hamiltonian circuit, f(n) must be bounded by $2^{P(n)}$ where P(n) is a polynomial. Thus no reasonable approximation to the traveling salesman problem is possible unless the Hamiltonian circuit problem can be solved quickly; that is, unless P = NP.

7.7 The World of NP

The fact that various NP-complete problems have resisted attempts to find reasonable algorithms for them suggests to many people that $P \neq NP$. Even if we accept this belief, there are still other open questions about classes of problems associated with NP. The class NP is defined in terms of the yes-instances of its problems. A class could also be defined in terms of no-instances. In correspondence with NP,we define the class co-NP as problems whose complements are in NP; that is, for each problem in co-NP there is a nondeterministic algorithm which has polynomial

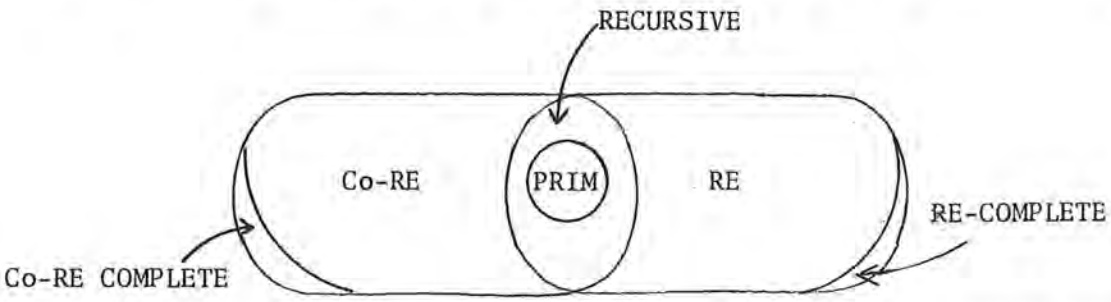
bounded running time for the no instances of the problem. This definition suggests the questions:

Does NP = co-NP?

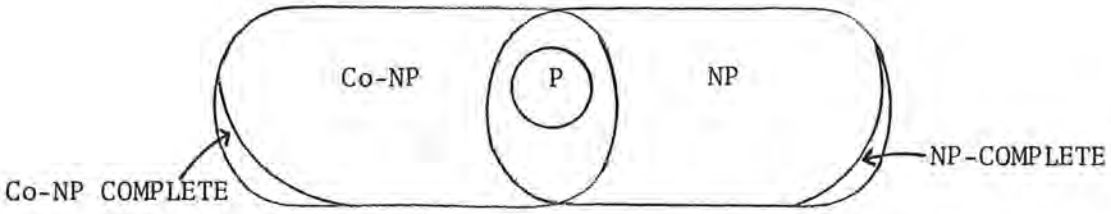
Does P = NP $\cap$ co-NP?

Unfortunately, these questions are unsolved.

While the above questions are unsolved, many people believe that the answer to each of these questions is no. The basis for this belief is the analogy between NP and RE. For RE the following diagram can be shown to be valid:



If the analogy between NP and RE is valid, the world of NP should look like:



A minor difference in the two diagrams is that RE $\cap$ co-RE has the name RECURSIVE, but NP $\cap$ co-NP has not been assigned a name. If this diagram is correct then there are several types of hard problems which are not NP-complete. In particular, there may be problems which are in NP $\cap$ co-NP but are not in P. Composite number is a candidate problem for this status. We know that composite is NP. The complement of composite number is prime number. While it is not immediately obvious, for every prime number there is a proof that the number is prime and the length of the proof is bounded by a polynomial in the log of the number. So the complement of composite is also in NP, and composite is in NP $\cap$ co-NP. But everyone (including the National

Security Agency) assumes that composite and prime do not have reasonable algorithms. Unfortunately, proving that composite/prime is not in P is probably very difficult since this would imply P $\neq$ NP.

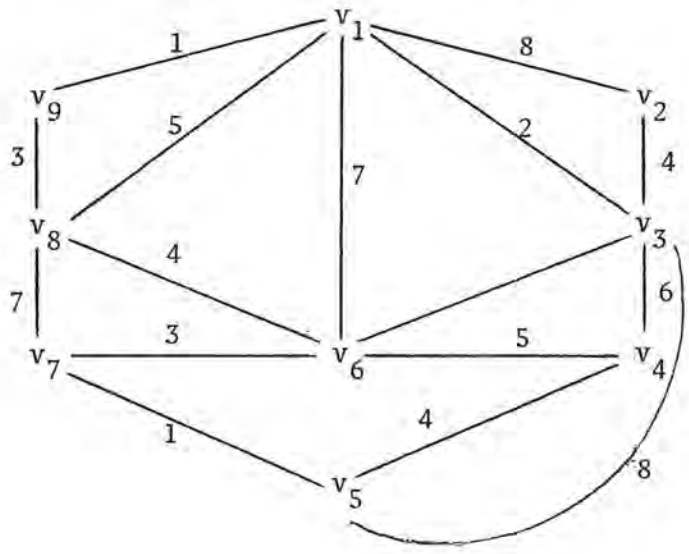
We conclude by mentioning that our diagram of the world of NP may be incorrect, but there is some reasonable circumstantial evidence to support it.

7.8 Exercises

a) Show that if Hamiltonian circuit is NP-complete then Hamiltonian path is NP-complete.

b) Show that if Hamiltonian path is NP-complete then Hamiltonian circuit is NP-complete.

c) For the following graph use the minimum spanning tree method to construct a short traveling salesman circuit. Assume that any missing edges have the shortest distance consistent with the triangle inequality. How close to the minimum circuit is your constructed circuit?



BOOKS FOR FURTHER READING

A.V. Aho, J.E. Hopcroft, and J.D. Ullman
The Design and Analysis of Computer Algorithms.
Addison-Wesley, Reading, MA 1974.

M.R. Garey and D.S. Johnson
Computers and Intractability: A Guide to the Theory of NP-Completeness.
W.H. Freeman, San Franciso, CA 1979.

D.H. Greene and D.E. Knuth
Mathematics for the Analysis of Algorithms.
Birkhäuser, Boston, MA 1981.

E. Horowitz and S. Sahni
Fundamentals of Computer Algorithms.
Computer Science Press, Potomac, MD 1978.

D.E. Knuth
The Art of Computer Programming.
Vol. 1: Fundamental Algorithms.
Vol. 2: Seminumerical Algorithms.
Vol. 3: Sorting and Searching.
Addison-Wesley, Reading, MA.

V. Pan
How to Multiply Matrices Faster.
Lecture Notes in Computer Science 179,
Springer-Verlag, Berlin, 1984.

H. Rogers
Theory of Recursive Functions and Effective Computability.
McGraw-Hill, New York, NY 1967.

R. Sedgewick
Algorithms.
Addison-Wesley, Reading, MA 1983.

J.F. Traub and H. Woźniakowski
A General Theory of Optimal Algorithms.
Academic Press, New York, NY 1980.

S. Winograd
Arithmetic Complexity of Computations.
SIAM, Philadelphia, PA 1980.

N. Wirth
Algorithms + Data Structures = Programs.
Prentice-Hall, Englewood Cliffs, NJ 1976.

ARTICLES

P. Buneman and L. Levy
The Towers of Hanoi Problem.
Information Processing Letters 10, 1980, pp. 243-244.

S.A. Cook
The Complexity of Theorem-Proving Procedures.
Proc. 3rd ACM Symposium on Theory of Computing, 1971, pp. 151-158.

P. Cull and J. DeCurtins
Knight's Tour Revisited.
Fibonacci Quarterly 16, 1978, pp. 276-285.

P. Cull and E.F. Ecklund, Jr.
Towers of Hanoi and Analysis of Algorithms.
American Mathematical Monthly 92, 1985, pp. 407-420.

R.M. Karp
Reducibility Among Combinatorial Problems, in Complexity of Computer Computations.
ed. by R.E. Miller & J.W. Thatcher, Plenum, New York, 1972, pp. 85-104.

V. Strassen
Gaussian Elimination is Not Optimal.
Numerische Mathematik 13, 1969, pp. 354-356.

T.R. Walsh
The Towers of Hanoi Revisited: Moving the Rings by Counting the Moves.
Information Processing Letters 15, 1982, pp. 64-67.