DYNAMIC PROGRAMMING INSIGHTS FROM PROGRAMMING CONTESTS

A Thesis
by
SAMANTHA VALENTINE LAPENSÉE-RANKINE

Submitted to the School of Graduate Studies
at Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

December 2021
Department of Computer Science

DYNAMIC PROGRAMMING INSIGHTS FROM PROGRAMMING CONTESTS

A Thesis
by
SAMANTHA VALENTINE LAPENSÉE-RANKINE
December 2021

APPROVED BY:

Raghuveer Mohan, Ph.D.
Chairperson, Thesis Committee

Mohammad Mohebbi, Ph.D.
Member, Thesis Committee

Alice McRae, Ph.D.
Member, Thesis Committee

Rahman Tashakkori, Ph.D.
Chairperson, Department of Computer Science

Marie Hoepfl, Ed.D.
Interim Dean, Cratis D. Williams School of Graduate Studies

Abstract

DYNAMIC PROGRAMMING INSIGHTS FROM PROGRAMMING CONTESTS

Samantha Valentine Lapensée-Rankine
B.S., Appalachian State University

Chairperson: Raghuveer Mohan, Ph.D.

Competitive programming is a sport for solving highly complex algorithmic problems. These problems range from simple brute-force to advanced algorithms like network flows or dynamic programming. Many problems require the use of several different techniques at once. Practicing for these contests helps improve algorithmic problem solving and programming skills, mathematical reasoning, testing and debugging, and team building skills, and in general, makes students into well-rounded computer scientists and prepares them for a long term career in computer science.

This thesis surveys over 70 problems from the Intercollegiate Programming Competition (ICPC) North American regional contests held during 2019. These problems are fun and intellectually stimulating, and by solving them we have gained many insights into the technique of dynamic programming. These insights may be helpful in identifying and formulating dynamic programming solutions. This thesis can be used as an introductory text or supplemental reading material in the technique of dynamic programming.

The ICPC contests are annual competitions held since 1977, and problems were archived at the ICPC Live Archive website. However, due to recent changes in administration, these contests are no longer archived since 2017. We build a web-based platform to archive problems after the 2017 ICPC contests. We have successfully archived all of the problems from the 2019 North American regional contests, totaling over 70 problems from 12 regional contests. While surveying these problems, we have also classified them based on the algorithmic technique used. A search feature is provided to help search based on technique. We hope that this website continues to archive all of the ICPC problems from all geographic regions, and helps students identify problems for practice and training.

Acknowledgements

This thesis would not have been possible without the support of many people. I first want to thank my thesis advisor, Dr. Raghuveer Mohan, whose teaching style and earnest interest in everything that he teaches inspired my own enthusiastic research into algorithms and dynamic programming. Dr. Mohan's door was always open for any questions, whether on my research, writing, or an interesting but off-topic algorithmic question, and I truly enjoyed getting to know him over the past two years. Without his assistance, this thesis would never have been completed.

I would like to acknowledge the members of my thesis committee, Dr. Alice McRae and Dr. Mohammad Mohebbi, and I am grateful for their very valuable comments on this thesis.

I would also like to thank all the professors in the computer science department who encouraged me during my time at Appalachian State. I would specifically like to mention Dr. Patricia Johann, who was the biggest advocate of my graduate degree, and without whom I would not be here today.

I thank my family and friends for all their support for me. I have to thank my sister for both her frustrating dedication to keeping me on track, and her willingness to help fine-tune the phrasing and syntax of my thesis even late into the night. Finally, I cannot fully convey how grateful I am to my parents for their years of unending love and encouragement through my years as a student, and throughout the process of writing this thesis. I would have never been able to accomplish what I have without them. Thank you.

Table of Contents

Abstract **iv**

Acknowledgements **v**

List of Figures **vii**

1 Introduction **1**
1.1 Popular Programming Contests . . . 3

2 Introduction to Dynamic Programming **9**
2.1 Fibonacci Numbers . . . 10
2.2 Minimum Coin Change . . . 13
2.3 Maximum Sum Sub-array . . . 17
2.4 Longest Increasing Sub-sequence . . . 19
2.5 Integer Knapsack . . . 23
2.6 Computing Shortest Paths . . . 30
2.7 Travelling Salesman . . . 37

3 Dynamic Programming Problems and Insights **41**
3.1 Combinatorial Counting Problems . . . 41
3.2 Dynamic Programming on Strings . . . 49
3.3 Dynamic Programming on Trees . . . 56
3.4 Dynamic Programming Problems in the ICPC . . . 58

4 Problem Classifications and Website **67**
4.1 Classification . . . 67
4.2 Website . . . 73

5 Conclusion **76**
5.1 Final Results . . . 76
5.2 Future Work . . . 77

Bibliography **80**

Vita **82**

List of Figures

1.1 The Different Geographic International Regions of the ICPC [6] 4
1.2 The North American Regions of the ICPC [6]. 4

2.1 Tree of Recursive Calls for Computing Nth Fibonacci Number 11
2.2 Tree of Recursive Calls for Computing Nth Fibonacci Number Using Memoization 12
2.3 Coin Change Using Exhaustive Search 14
2.4 Longest Increasing Subsequence Example 19
2.5 Speeding Longest Increasing Subsequence with Table Method 22
2.6 Speeding Longest Increasing Subsequence with BST Method 23
2.7 Divide and Conquer Dynamic Programming; Reverse and Forwards Formulations 29
2.8 Divide and Conquer Dynamic Programming; Finding the Set of Items in an Optimal Solution 30
2.9 Basic Directed Acyclic Graph 31
2.10 Topological Sorting of a DAG 31
2.11 Longest Increasing Subsequence Topological Graph 32
2.12 Travelling Salesman Tour on a Graph 37
2.13 Travelling Salesman Recursion Tree, with Current Subproblem in White and Other Subproblems Grayed Out 39

3.1 Recursive Binomial Coefficient Tree. 42
3.2 Hats i Through n 44
3.3 Person i Receives Hat j 44
3.4 Person j Receives Hat i 44
3.5 Nobody Receives Hat i 45
3.6 All Faces on a Six-sided Die 46
3.7 All Possible Combinations of Three Dice Whose Faces Sum to 6 47
3.8 Steps to Turn “Friday” into “Thursday” 50
3.9 Longest Common Subsequence between Strings “mistake” and “staircase” 53
3.10 Example Elevation Map 59
3.11 An Optimal Path 60
3.12 Transforming the 2D Map into a Multi-Level Graph 61
3.13 Example of a Rooted Tree with Two Possible Maximum Jumping Paths 63

4.1 Home Page of Thesis Website 74
4.2 Problem Database Inside Caspio 75
4.3 Adding a New Problem to the Database 75

Chapter 1

Introduction

Competitive programming is a sport for solving complex coding problems using algorithms and data structures. Competing sharpens problem-solving and programming skills. Competition environments train programmers in solving complicated problems in stressful environments. These problems range from simple brute-force (try all possible solutions) to advanced algorithms like network flows or dynamic programming. Many problems require the use of several different techniques at once. Doing well in programming competitions can increase a programmer's expertise and prepare them for a career in computer science. In this thesis, we study problems from the premier programming contest for university students – the Intercollegiate Programming Contests (ICPC), specifically the 2019 North American regional competitions. These problems are fun and intellectually stimulating, and by solving them we have gained many insights into the technique of dynamic programming. We outline these insights in this thesis, and hope these help contestants to identify and formulate dynamic programming solutions effectively.

Competitive programming deepens and expands computer science skills in competitors. Deciphering, solving, and writing solutions to contest problems gives experience in the whole software pipeline. Classroom environments can often be too structured, may ask trivial questions to assess fundamental knowledge, and can be based on rote memorization, rather than creatively applying knowledge and skills learned in class. This is why competitive programming is so exciting – all competitive programming problems can be solved with topics from an intermediate undergraduate algorithms course, yet they are so challenging that they can give

Ph.D. students and veteran computer scientists a run for their money. Contest problems train students not only with coming up with fast algorithmic solutions, but also in myriad other skills including; i) algorithms analysis to determine if their solutions are fast enough to be accepted by the contest judge, ii) art of abstracting bigger problems based on smaller subproblems that can be solved efficiently, iii) implementing complex algorithms in code which includes knowledge of the many pre-existing library algorithms and data structures, and iv) testing and debugging which includes writing programs to generate large test cases. All of these skills promote creativity and build a strong foundation that promotes a long term career in computer science.

The competitive programming literature contains well over 5000 problems from many different contests around the world. Many of the problems are lost or not widely known. The problems from the most popular ACM ICPC contests were archived at the ICPC Live Archive [7]. However, recent changes in administration mean that problems are no longer archived here. Our thesis builds a web-based platform to archive problems after the 2017 ICPC contests. We have successfully archived the problems from the 2019 North American regional contests, totalling over 70 problems [15]. While surveying these problems, we also classified these problems based on type of algorithmic technique. The web application allows for easy addition of new problems. The hope is that this site is filled with all problems from both the ICPC contests and the biggest contest for high school students – the USA Computing Olympiad (USACO). Future contestants may find this web application useful as they can find problems based on a particular algorithmic technique, thereby helping them practice applying the technique.

The technique of dynamic programming has been widely used in these contests, for different kinds of problems. We offer some insights and perspectives in applying this technique. We hope that such insights are helpful in identifying problems with dynamic programming, and in using that knowledge to solve these types of problem more efficiently in terms of coding and testing.

The main contributions of this thesis are as follows:

1. Surveys and classifies over 70 problems from the ICPC 2019 North American regional contests, a total of 12 contests.

2. Creates a website where these problems are archived and can be easily searched by problem type.

3. Offers detailed insights into the application of the technique of dynamic programming.

1.1 Popular Programming Contests

In this section, we describe the rules, competition environments, and resources for two popular programming contests – the Intercollegiate Programming Contest and the USA Computing Olympiad.

ACM ICPC

Programming competitions are an incredibly popular part of computer programming; over 50,000 students around the world compete in the Association for Computing Machinery's (ACM) global and regional International Collegiate Programming Contest (ICPC) competitions [4]. Programming competitions increase interest and enthusiasm in computer science studies, and prepares students for job opportunities in computer science fields [1].

We chose to use programming competition problems from the ICPC, since it is the most well-known and reputable international programming competition. The ICPC competitive programming competition is a global programming competition organized by and for universities [4]. Global competitions are split up into eight broad international districts [6]. Those eight international districts are Latin America, Africa and Arabia, Europe, Northern Eurasia, Asia West, Asia Pacific, Asia East, and North America, as shown in Figure 1.1.

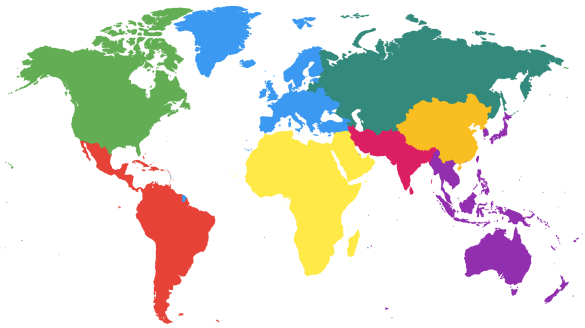


Figure 1.1: The Different Geographic International Regions of the ICPC [6]

As this is an incredibly large global competition, we focus on problems from the North American district in the year 2019. This district is further divided into 11 geographic regions as shown below.

Figure 1.2: The North American Regions of the ICPC [6].

According to the ICPC website [6], those eleven regions, and the states and provinces in those regions are list in Table 1.

Table 1

ICPC North American Regions		
	Region	States and Provinces Within Region
1	Pacific Northwest	Washington, Oregon, North California, British Columbia, North Idaho, Hawaii
2	Rocky Mountain	Arizona, Utah, Colorado, Wyoming, Eastern Nevada, Idaho, Montana, Alberta, Saskatchewan, New Mexico
3	South California	Southern California
4	North Central	Minnesota, Wisconsin, Western Ontario, Manitoba, Iowa, North Dakota, South Dakota, Nebraska, Kansas, the UP of Michigan
5	South Central	Louisiana, Oklahoma, Texas
6	East Central	Ohio, Michigan, eastern Ontario, western Pennsylvania, Indiana
7	Mid Central	Missouri, Arkansas, Illinois, Kentucky, Tennessee
8	Northeastern	Quebec, New Brunswick, Nova Scotia, Prince Edward Island, Newfoundland, Labrador, Maine, New Hampshire, Vermont, Massachusetts, Rhode Island, Connecticut, New York State excluding New York City
9	New York	The greater New York metropolitan area, including areas of New York, New Jersey, Connecticut
10	Mid Atlantic	New Jersey, eastern Pennsylvania, Delaware, Maryland, the District of Columbia, Virginia, West Virginia, North Carolina
11	Southeast	Florida, Alabama, Georgia, South Carolina, Mississippi

Competitions work the same way throughout all the regions for the ICPC. Each team consists of three contestants who represent their respective universities[5]. During the competition, each team is given a single workstation, so only one of the three team members can work

on a computer and write code at a time. The competitions are usually five hours, in which the contestants must solve six to twelve algorithmic word problems. Each regional competition can have hundreds of teams that all compete during the same time period, in one of the many contest sites. All team members may only speak to members of their own team, without access to the Internet and other installed software other than the environment in which their programs are allowed to run. They are not allowed to use imports or similar pre-written code from others to solve the problems. They are however, allowed to carry a "hackpack", any amount of printed resources that includes pretested code on many data structures and algorithms that they can use as building blocks to solve their problems. A part of the challenge of these problems is to abstract the solution and determine the algorithms necessary to solve problems during the competition. Afterwards, all they have to do is copy them from their prepared hackpack and piece them together correctly for their solution. Each region's competition follows the same ICPC regional rules, and regional contest directors can vary their contest's rules somewhat to accommodate different educational systems in their region without superseding the ICPC Regional Rules [5].

After contestants figure out an algorithmic solution to a problem, they can code the solutions in any of the following programming languages – Java, Kotlin, Python, C, or C++ to submit to the judge to be ranked. There is an automatic judging system that determines whether a solution is correct or not. Feedback from the judge is limited – either 'Accept' for a correct solution, or a 'Reject' for an incorrect solution. Other common error feedback given are for compiler syntax error, run-time errors, or errors about exceeding the time or space used by the program. The judge provides no further information on what is wrong with their coded solution. Teams are scored based on the time they take to solve each of the problems. For a small time penalty, they can resubmit their solutions as many times as they like, but this penalty is only added if the team eventually solves the problem. If the team never submits a correct answer, that specific problem's penalties are never added. Teams are ranked first by total number of problems solved, and ties are broken by time scores, including penalties. Further ties (which are exceedingly rare) are broken based on time of last solved problem, then on the time of the second to last solved problem, and so on. The team that solves the most of problems in the least amount of time wins the competition.

The USA Computing Olympiad

Although we haven't classified problems from the USACO, we describe one of the biggest programming competitions for high school students. The USACO is a US-based program that trains students of all levels for the International Olympiad in Informatics (IOI) [20]. The USACO is a national organization that offers a training site, resources and seasonal programming competitions. While these are geared towards high school students, the problems offer a substantial challenge for experienced computer scientists and researchers as well.

The USACO's seasonal contests come in four tiers: Bronze, Silver, Gold, and Platinum. Bronze tier problems tend to involve simulation, sorting, and searching algorithms, Silver tends to use problems with data structures and more advanced algorithmic techniques, Gold has more problems that combine techniques and more advanced concepts like spatial data structures, and Platinum tier uses very advanced problems that offer an extreme challenge to the most experienced of competitors [22]. The top twenty five high school students that do well in these contests are invited to a summer camp at Clemson University. The judges at the summer camp include past IOI contestants, and students go through a rigorous training program to be one of the top four students selected to represent the USA at the IOI. The USACO contests are held online for three to five continuous hours, which start as soon as the competitor logs into the competition and obtains the problems. Final solutions are submitted through a web interface [19].

The USACO offers a free online training site for those who are interested in starting their competitive programming journey. Like the competitions, the problems start out simple, but quickly escalate in difficulty and introduce increasingly more advanced algorithmic concepts. The training site has about one hundred problems and hundreds of hours of training materials to go through. As future work of this thesis, we intent to classify all of the problems from the USACO training site. We hope our classification and dynamic programming insights are useful for both becoming better prepared for these contests and using the dynamic programming technique to solve a wide range of problems.

The International Olympiad in Informatics (IOI) is the most prestigious international computing contest, considered the Olympics of programming contests. It is an international

programming competition organized annually in and by one of the countries that participates in the competitions [8]. In the annual international competition, each participating country sends four contestants and two accompanying adults to compete in the IOI [8]. Unlike the ICPC there are no teams; each contestant competes individually.

The competition consists of two five-hour contests, where each competitor must solve three algorithmic word problems. Each competition follows the same regulations approved by the general assembly, but each individual competition has slightly different rules set by the host country for that year [9]. Contestants have five hours to solve the three competition problems given to them, and must code the solution in C++ or Java. Contestants may submit at most fifty solutions for each task, with at most one submission per minute [10]. There are no penalties for multiple submissions, and the final score for contestants is the sum of all the maximum scores for their problems. Contestants are scored based on the number of solved problems, and the contestant with the highest score wins.

The rest of the thesis is organized as follows. Chapter 2 is an introduction to dynamic programming, where we explain dynamic programming and introduce and describe how to solve the common problems. We also provide some of the insights that can be used to identify and formulate problems that have a dynamic programming solution. Chapter 3 is dynamic programming insights, where we introduce less common dynamic programming problems and give a perspective, and provide editorials to some of the problems from the 2019 ICPC contests that are solved using dynamic programming. Chapter 4 describes our Web Classifier – how it is built, how it can be used, and how this is useful for aspiring competitors in practicing for competitions. We conclude the thesis in Chapter 5 summarizing our contributions and discussing future work.

Chapter 2

Introduction to Dynamic Programming

This chapter introduces the technique of dynamic programming, and discusses some classical problems that use this technique. We provide a well-rounded exposition of these problems, and provide intuition into the technique by developing ideas incrementally. This chapter can be used as a quick refresher or as an introductory text in dynamic programming. We note that dynamic programming is commonly used in solving discrete optimization problems, where we are minimizing or maximizing an objective function. But they are also quite efficient in solving several counting problems.

We assume that the input is a list stored as arrays (either single or multi-dimensional). As this text is intended to be read primarily by computer scientists and professionals, we use the well-known array notation using square brackets to indicate an element in the array. We will be expressing algorithms in pseudocode. Each algorithm describes the set of inputs and outputs, and the steps are described using keywords commonly found in programming languages.

Dynamic programming is a technique that solves solutions to larger sub-problems from smaller ones. Therefore, to express each dynamic program, we start with defining the state space, then provide base cases, inductive cases, and finally describe the optimal solution. We may use math notation to describe the transitions from smaller sub-problems to larger ones in the inductive step, or in expressing the optimal solution.

2.1 Fibonacci Numbers

The infinite sequence of numbers $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ is the famous Fibonacci sequence. It is one of the simplest and earliest known sequences, and is found everywhere in nature and mathematics, like for example, the number of petals on a flower is usually a Fibonacci number, and Fibonacci spirals can be found in sunflowers, pinecones, pineapples, and artichokes. The Fibonacci sequence is also strongly related to the golden ratio, which approximates the ratio of two adjacent numbers in the sequence.

The Fibonacci sequence starts with the zeroth term 0, first term 1, and every other term in the sequence is the sum of the previous two terms. We can compute the nth term in the sequence, denoted $F[n]$, using the following recursive algorithm.

input : Positive integer n

output: The nth Fibonacci number

```
1 Fibonacci(n):
2 if n = 0 or n = 1 then
3 |   return n
4 end
5 else
6 |   return Fibonacci (n-1) + Fibonacci (n-2)
7 end
```

Algorithm 1: Naive Fibonacci

The algorithm makes the tree of recursive calls shown in Figure 2.1 below.

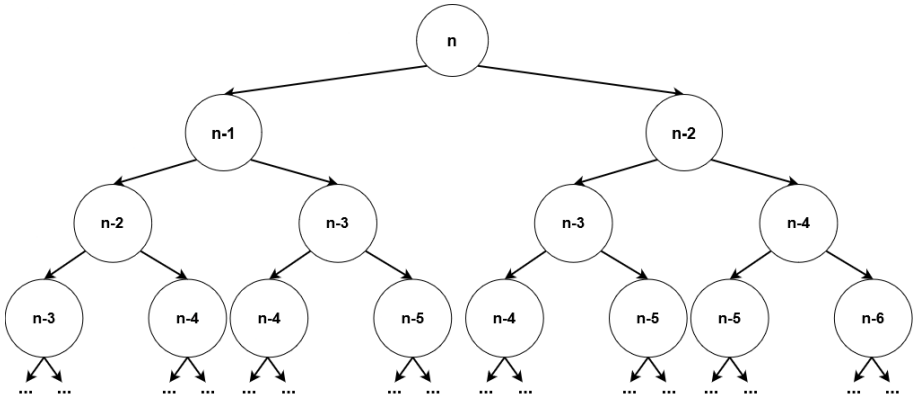


Figure 2.1: Tree of Recursive Calls for Computing Nth Fibonacci Number

It can be seen that this algorithm runs in exponential time. In each step of the recursion, the number of sub-problems doubles, and the height of this tree is n. In this problem, it is easy to see overlapping sub-problems: in order to calculate the n^{th} Fibonacci number $F[n]$, we must first calculate $F[n-1]$ and $F[n-2]$. But to compute $F[n-1]$, we need $F[n-2]$ and $F[n-3]$, and so on. So each sub-problem $F[i]$ is computed several times. An easy way to speed this up is to "remember" solutions to sub-problems. We can use a table, in this case, an array of solutions to each of the sub-problems named `fib`, and whenever we compute a solution to a new sub-problem we store its value in this table.

input : Positive integer n

output: The *nth* Fibonacci number

```
1 Fibonacci(n):
2 if n = 0 or n = 1 then
3 |   return n
4 end
5 if fib[n] is not stored then
6 |   fib[n] = Fibonacci (n-1) + Fibonacci (n-2)
7 end
8 return fib[n]
```

Algorithm 2: Fibonacci with Memoization

As can be seen by the recursive tree in Figure 2.2, each recursive call finishes when the algorithm comes across a sub-problem that has already been computed. All the sub-problems along the left path of the recursion tree are computed, while any other sub-problem is returned from the lookup table. Therefore, the algorithm takes only $O(n)$ time, as each sub-problem is only computed once.

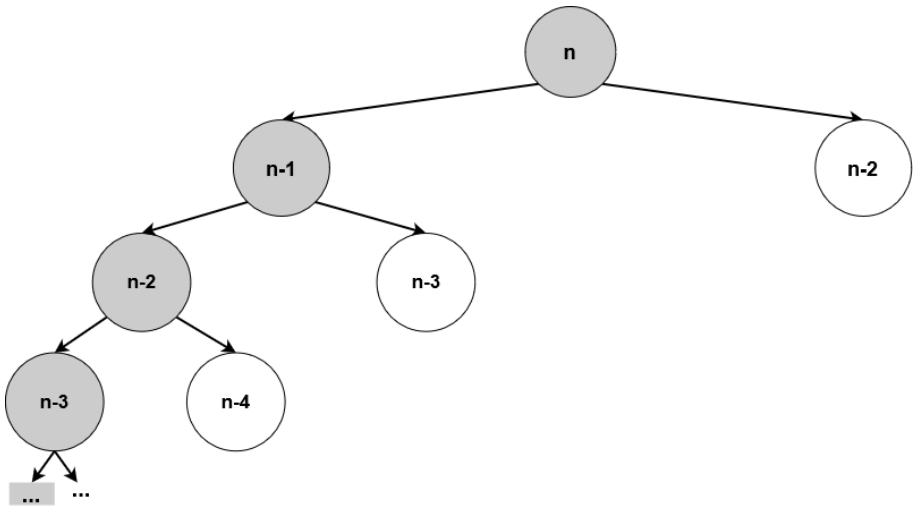


Figure 2.2: Tree of Recursive Calls for Computing Nth Fibonacci Number Using Memoization

The above technique of using a table of solutions to store solutions to smaller sub-problems is known as *memoization*. Memoization stores the results of expensive function calls in arrays or objects so that we can extend from these solutions later instead of repeating the same calculations. Using memoization and recursion together so that larger sub-problems can be solved quickly is dynamic programming. This technique was popularized by Bellman and Ford in their shortest path algorithm, and was used in sequential planning problems in engineering.

Insight #1: Dynamic Programming = Recursion + Memoization

In other words, dynamic programming can be used to speed up some exponential algorithms by using memoization.

2.2 Minimum Coin Change

Another classical problem for which we can apply dynamic programming is the coin change problem. Consider the USA coin denomination system with 1, 5, 10, and 25 cent coins, and we would like to make change for 72 cents. The minimum number of coins to make 72 cents is by using two 25 cent coins, two 10 cent coins, and two 1 cent coins for a total of 6 coins. This can be found using a greedy algorithm; keep picking the largest coin you can until you get the solution. However, if we have a different set of coin denominations, for example, 1, 6, and 10 cents coins, a greedy solution does not work. To make 12 cents in change, a greedy solution uses a 10 cent coin and two 1 cent coins for a total of three coins, while the optimal solution is to use two 6 cent coins.

To generalize the coin change problem, we are given a list of n coin denominations $c[1], \ldots c[n]$, where we can use as many of each coin type as possible, and we would like to determine the minimum number of coins needed to make exactly C cents in change. Like most problems, there is a simple exhaustive search algorithm that assures the optimal solution. Try all possible coin denominations $c[i]$, and for each of them, recursively compute $C - c[i]$. The optimal solution to make C cents in change is therefore to use one coin $c[i]$, and the minimum of completing sub-problem $C - c[i]$.

input : An array c of all coin denominations, and a non-negative integer C

output: The minimum number of coins to make C

```
1  Change(c,C):
2  if C = 0 then
3  |   return 0
4  end
5  if C < 0 then
6  |   return ∞
7  end
8  mincoins = ∞
9  foreach element c[i] in c do
10 |   mincoins = min{Change(c, C − c[i])+1, mincoins}
11 end
12 return mincoins
```

Algorithm 3: Naive Coin Change

The tree for this algorithm with the USA coin denominations is shown in Figure 2.3 below.

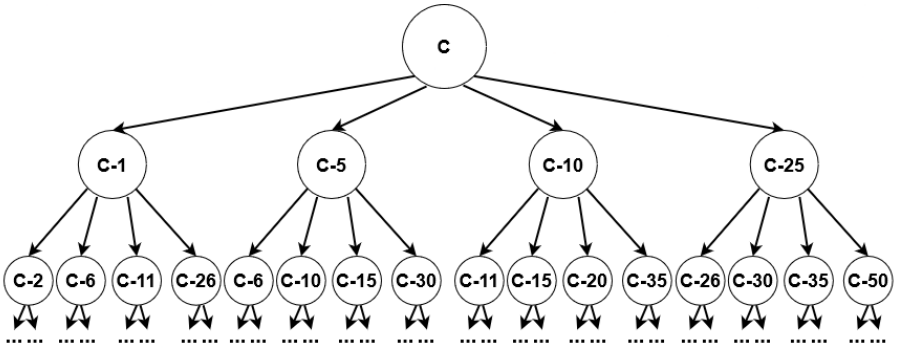


Figure 2.3: Coin Change Using Exhaustive Search

It can be seen that this algorithm recomputes the same sub-problems, and we can use memoization to speed it up. Simply store the solution to each sub-problem in a table, in this

case an array of integers named `coins`, where `coins[x]` stores the minimum number of coins it takes to make x cents in change.

input : An array c of all coin denominations, and a non-negative integer C

output: The minimum number of coins to make C

```
1  Change(c,C):
2  if C = 0 then
3  |  return 0
4  end
5  if C < 0 then
6  |  return ∞
7  end
8  if coins[C] is stored then
9  |  return coins[C]
10 end
11 mincoins = ∞
12 foreach element c[i] in c do
13 |  mincoins = min{Change(c, C − c[i])+1, mincoins}
14 end
15 coins[C] = mincoins
16 return mincoins
```

Algorithm 4: DP Coin Change

This then gives us our second insight into dynamic programming.

Insight #2: Dynamic Programming = Recursive Exhaustive Search + Memoization

The correctness of this algorithm follows from induction. As a base case, when $C = 0$, it is trivial that there are 0 coins. For the inductive case, we assume that the algorithm is correct for sub-problems smaller than C due to the inductive hypothesis. Therefore, taking a minimum of all sub-problems $C - c[i]$ and adding 1 is the best way to make C cents in change.

The above inductive proof is a good template to express dynamic programming solutions. We start with defining a state space and clearly describe what individual elements in the state space store. Then we define the set of base cases followed by the set of inductive cases that can be expressed in terms of smaller sub-problems. We finally describe how the optimal solution can be computed after filling the entire state space. For the coin change problem, the dynamic programming formulation will look like:

Input:	An array of coin denominations c, and a non-negative integer C, which is the amount of change to make
State Space:	An array `coins` where `coins[i]` is the minimum number of coins required to create i cents in change
Base Case:	`coins[0]` $= 0$, `coins[i]` $= \infty, \forall i < 0$
Inductive Case:	`coins[i]` $= \min_{j=1}^{n} \{\texttt{coins}[\texttt{i} - \texttt{c}[\texttt{j}]]\} + 1$
Optimal Solution:	`coins[C]`

From now on, we will use the above dynamic programming formulation to succinctly represent a dynamic program. We will continue to use pseudocode to represent naive solutions that are not dynamic programming. Interested readers are encouraged to take the dynamic program formulation and turn them into implementable programs.

The runtime of this algorithm is $O(nC)$ as there are $O(C)$ sub-problems to compute, and each one takes $O(n)$ time to take a minimum of $O(n)$ smaller sub-problems. This therefore gives us a convenient way to analyze dynamic programming algorithms.

> ***Insight #3:*** The runtime of dynamic programming algorithm = total number of sub-problems $\times$ total time for each sub-problem

Note that the runtime $O(nC)$ is not a polynomial runtime due to the dependence on C. In fact this will be considered an exponential[1] algorithm as it is exponential in the number of bits to represent an integer x.

[1]For algorithms on integer inputs, a polynomial time algorithm is one that runs polynomial in $\log C$, which is the number of bits needed to represent C. So a running time of $O(C)$ is considered to be exponential. Sometimes these are also called pseudo-polynomial running times.

We can also think of dynamic programming in a bottom-up way. Since a sub-problem of size n only depends on smaller sub-problems, we can compute all of the solutions iteratively instead of recursively, from the smallest to the largest sub-problem.

The above dynamic programming algorithm can the thought of as an incremental construction algorithm, where in each step we solve a successively larger sub-problem. Using memoization to store the results allows us to compute solutions to larger sub-problems quickly. Therefore, we have the following insight.

Insight #4:	Dynamic Programming = Incremental Construction + Memoization

2.3 Maximum Sum Sub-array

The technique of dynamic programming is naturally applicable for sequential problems on arrays. Consider the classic problem of computing the maximum sum of a sub-array. Given an array of integers of size n, find the largest contiguous sum of a sub-array, where a sub-array spans contiguous elements from index i to index j. A naive solution to this problem is to use an exhaustive search; for every possible starting index i, calculate the sum from i to every possible ending index j.

```
   input  : Array of integers arr
   output: The max sum of a sub-array in arr
 1 MaxSum(arr):
 2 max = −∞
 3 for i ← 0 to n do
 4 |   for j ← 0 to i do
 5 |   |   sum = 0
 6 |   |   for k ← j to i do
 7 |   |   |   sum += arr[k]
 8 |   |   end
 9 |   |   if max < sum then
10 |   |   |   max = sum
11 |   |   end
12 |   end
13 end
14 return max
```

Algorithm 5: Naive Max Sum

This solution takes $O(n^3)$ time, as there are $\binom{n}{2} = O(n^2)$ sub-problems, and computing the sum of each subarray takes $O(n)$ time. To speed this runtime, we can use dynamic programming again, using an array `prefix` to store prefix sums, i.e., `prefix[i]` stores the sum of the first i elements in `arr`. Note that this can be pre-computed in only $O(n)$ time. The sum of a subarray `arr[i]..arr[j]` is then `prefix[j]-prefix[i-1]`, which takes only $O(1)$ time. Using this, the above algorithm to compute the maximum sum subarray takes $O(n^2)$ time.

We note that there is a divide and conquer algorithm that can solve this problem in $O(n \log n)$ time. We leave this as an exercise for the interested reader.

We now show how we can use dynamic programming to solve this in only $O(n)$ time. Let the array `sums` be the table of solutions, where `sums[i]` is the largest sum of a sub-array

ending at index i. As a base case, `sums[0] = arr[0]`. For the inductive case, we can see that each solution i either extends from the previous solution $i-1$ by adding `arr[i]` to it, or is a subarray containing only one element `arr[i]`. More formally, we have `sums[i] = max{sums[i-1] + arr[i], arr[i]}`. Since the array can contain negative numbers, the optimal solution may not be in `sums[n]`. Therefore we need to search the array of solutions and return the largest one.

Input:	An array of integers `arr`
State Space:	An array `sums` where `sums[i]` is the largest sum of a sub-array ending at index i
Base Case:	`sums[0]` = `arr[0]`
Inductive Case:	`sums[i] = max{sums[i-1] + arr[i], arr[i]}`
Optimal Solution:	$\max\{\texttt{sums[i]}\}, \forall i \in \{0, \ldots, n-1\}$

2.4 Longest Increasing Sub-sequence

Another common problem on sequences that uses dynamic programming is the longest increasing sub-sequence problem. A sub-sequence is a sequence of elements in the original array, retaining their order, that need not be contiguous. Given an array of integers named `arr`, find the length of the arrays longest increasing sub-sequence. For example, if the original sequence is $[14, 15, 12, 11, 4, 8, 7, 9]$, the length of the longest increasing sub-sequence is three. One of the subsequences with this length is $[4, 8, 9]$.

Figure 2.4: Longest Increasing Subsequence Example

This problem can be solved using a similar state space to the maximum value sub-array problem. Let the array `length` be the table of solutions, where `length[i]` is the length of the longest increasing sub-sequence that ends at index i. As a base case, `length[0]` is 1, as any sequence of size 1 counts as increasing. Unlike the maximum value sub-array

problem though, this problem doesn't require a contiguous sub-sequence so a solution can extend from any previous solution, not just the previous one. So we check all solutions `length[j]`, for $0 \leq j < i$, that we can extend from, meaning if $\texttt{arr}[\texttt{j}] \leq \texttt{arr}[\texttt{i}]$, then add one to `length[j]`. The largest such solution is then stored in `length[i]`. More formally, $\texttt{length}[\texttt{i}] = \max\{\texttt{length}[\texttt{j}] + 1\}, \forall j \in \{0, \ldots, i-1\} : \texttt{arr}[\texttt{j}] \leq \texttt{arr}[\texttt{i}]$. The optimal solution could end at any index, therefore, we have $\texttt{longest} = \max\{\texttt{length}[\texttt{i}]\}, \forall i \in \{0, \ldots, n-1\}$.

Input:	An array of integers `arr`
State Space:	An array `length` where `length[i]` is the length of the longest increasing sub-sequence that ends at index i
Base Case:	`length[0]` $= 1$
Inductive Case:	$\texttt{length}[\texttt{i}] = \max\{\texttt{length}[\texttt{j}] + 1\}, \forall j \in \{0, \ldots, i-1\} : \texttt{arr}[\texttt{j}] \leq \texttt{arr}[\texttt{i}]$
Optimal Solution:	$\max\{\texttt{length}[\texttt{i}]\}, \forall i \in \{0, \ldots, n-1\}$

This brings the notion of an optimal substructure. Smaller sub-problems need to solve the same problem as the larger one, and larger sub-problems need to be able to extend from smaller ones. Note that if we did not include the key detail in the definition of our state space where we allow each solution to end at an index, it would be difficult to extend from previous solutions. So, the way we define our state space is key in solving problems using the dynamic programming technique.

The run time of the above algorithm is $O(n^2)$, since there are $O(n)$ sub-problems to solve, and each one requires iterating over $O(n)$ solutions. This can be sped up to only $O(n \log n)$ time as discussed in the next section.

Note that the solution above only returns the length of the longest increasing subsqequence, not the actual subsequence itself. In order to find the elements of the longest subsequence, we can use back pointers. To do this, we only need to make minor adjustments to the algorithm above. First, we need another array `backpointers`, where `backpointers[i]` is the index of the sequence that `length[i]` extends from. In order words, if during the inductive case for `i`, the maximum value is found in `length[j]`, then j would be stored in

`backpointers[i]`. To get the optimal subsequence, use the index c of the optimal length, include `arr[c]`, and then move to $c =$ `backpointers[c]`, include this into the optimal solution, and so on. One can print all the elements in the optimal subsequence in reverse, starting from c and following these backpointers.

As this only adds one more step and therefore only $O(1)$ extra time to the inductive case, this does not change the runtime of the algorithm. It is possible to add backpointers similarly to almost every dynamic programming algorithm without changing the runtime of the algorithm.

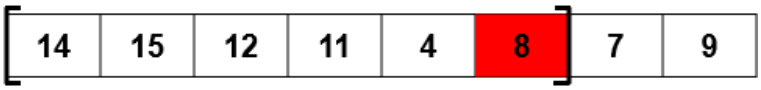
Insight #5:	The actual set of elements belonging to an optimal solution can be obtained using backpointers

Speeding the Longest Increasing Subsequence

It is possible to reduce the runtime of the longest increasing subsequence problem from $O(n^2)$ to $O(n \log n)$ using two different methods. We describe these below.

The first method, called the table method, requires an array `mins`, where `mins[i]` is the minimum value of the last elements of all subsequences of length `i`. In other words, for every subsequence of length i seen so far, `min[i]` holds the smallest of the last elements of all those subsequences. Notice that the array `mins` is sorted. So for the inductive case for element i, we can binary search on `mins` to find the index j that is the predecessor of `arr[i]`. The value in `mins[j]` holds the length of the longest increasing subsequence ending at j. We can compute the length of the longest increasing subsequence ending at i as `length[i]` = `mins[j]` + `1`. This may cause an update in the row `length[i]`. Specifically, if `mins[length[i]]` > `arr[i]`, the we set `mins[length[i]` = `arr[i]`. As the binary search algorithm runs in $O(\log n)$ time, this method reduces the time of the inductive case from $O(n)$ to $O(logn)$, therefore reducing the runtime of computing the longest increasing subsequence to only $O(n \log n)$.

Figure 2.5 shows the state of the array `mins` after completing the case for element 8.



Length	Min Value
1	4
2	8
3	∞

Figure 2.5: Speeding Longest Increasing Subsequence with Table Method

Another possibility is the Binary Search Tree (BST) method, which requires storing the state space `length` as a splay tree instead of an array[2]. A splay tree is a type of BST where on each insert or find, the current element is "splayed", or rearranged so that it is made the new root of the tree. Please refer to the seminal work of Sleator and Tarjan [17] for more information about splay trees.

Augment the splay tree so that each node also has the maximum solution in its subtree. That is, at node x in the tree, we maintain a variable $\max(x) = \max_{y \in \texttt{x'ssubtree}} \{\texttt{length[y]}\}$. Note that this information can be kept up to date after regular BST rotations. In this method, we scan the input array one at a time sequentially and insert `arr[i]` into the splay tree. So when the i^{th} element is inserted, the only elements in the splay tree are those to the left of i, the elements $\texttt{arr}[1], \ldots, \texttt{arr}[\texttt{i}-1]$. When element `arr[i]` is inserted into the splay tree, it is splayed to the root of the tree. Therefore, the only solutions that can extend their increasing subsequences to `arr[i]` are contained in the left subtree of `arr[i]`. To compute the length of the longest increasing subsequence ending at i, we can take $\texttt{length}[\texttt{i}] = \max(\text{left subtree of arr[i]}) + 1$. We can also update $\max(\texttt{arr}[\texttt{i}])$ easily after finding `length[i]`. The final optimal solution is found in $\max(r)$ where r is the root of the tree.

At each step, we take only $O(\log n)$ amortized time to insert `arr[i]` into the tree. So the overall runtime to find the length of the longest increasing subsequences is $O(n \log n)$.

Figure 2.6 shows the state of the BST `length` completing the case for element 8.

[2]Any BST will work, but we find the exposition to be a lot simpler using splay trees.

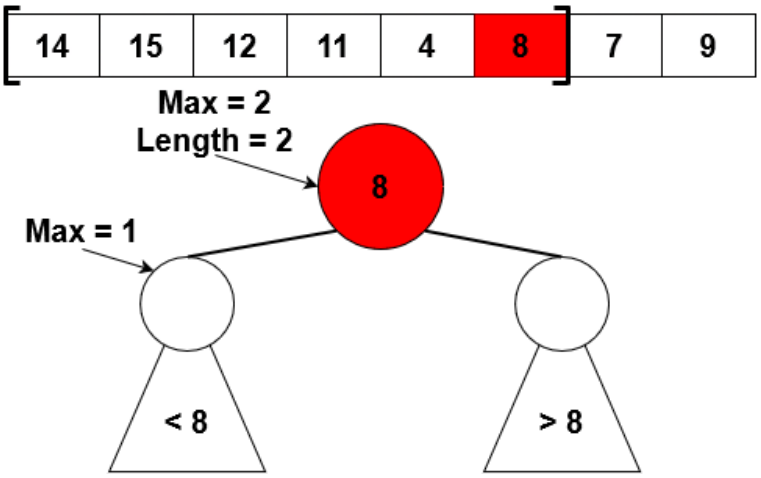


Figure 2.6: Speeding Longest Increasing Subsequence with BST Method

Insight #6:	Binary search trees can help in speeding up determining a maximization (minimization) over a set of solutions.

Longest Increasing Subsequence as Shortest Path

The longest increasing sub-sequence problem can also be viewed as a longest path problem, or rather, a special shortest path problem. We can turn this into a graph problem by constructing a graph with nodes corresponding to the array indices. There is an edge from node j to node i if $j < i$ and `arr[j]` $\leq$ `arr[i]`, so that the only edges that exist to a node are from other nodes that both come before the current node in the list and that have a strictly smaller value. The significance of being able to view this problem as a shortest path problem will be explained in Section 2.6.

2.5 Integer Knapsack

The knapsack problem is commonly used in resource allocations, and uses dynamic programming. Given n items, each with with an associated weight and value, maximize the total value of a set of items that can fit into a capacity C knapsack. As input, we are given an array `wt[1..n]` of weights, an array `val[1..n]` of values, where `wt[i]` and `val[i]` represent the weight and value of item i, respectively, and a capacity C knapsack. All weights are inte-

gers, as having real weights makes the problem NP-hard. As output, we would like to find the maximum value subset of items such that the sum of the weights of the items in the set is at most C.

There are two common versions of the Knapsack problem, one where multiple copies of an item are allowed to be included into the knapsack, and another version where only one copy of an item can be included. We explain both of these versions below.

Knapsack: Multiple Copies Allowed

In this version of the knapsack problem, you are allowed to place multiple instances of the same item in the knapsack. For example, let `val` $= [100, 150, 300]$, `wt` $= [10, 20, 30]$, and $C = 50$. The optimal solution gives a maximum of value of 500 either by using five copies of item 1, or using two copies of item 3. Using dynamic programming, we can the maximum value of items in an optimal solution. Using backpointers, we can also get the set of items in an optimal solution.

Let the array `sack` be our state space, where `sack[i]` is the maximum value that we can pack into a knapsack of capacity i. As a base case, we have `sack[0]` $= 0$. For the inductive case, we check whether we can include or exclude item j and take the maximum of the best value obtained from either choice. When we include item j, then we get its value `val`[`j`], and take the best solution that can fit into a capacity `i` $-$ `wt`[`j`] knapsack, which can be found in `sack`[`i` $-$ `wt`[`j`]]. If we exclude item j, then we simply look at the best solution to capacity `sack`[`i` $-$ `1`] knapsack. So we have:

$$\texttt{sack}[\texttt{i}] = \max\left\{\texttt{sack}[\texttt{i}-\texttt{1}], \max_{j=0}^{n}\{\texttt{sack}[\texttt{i}-\texttt{wt}[\texttt{j}]]+\texttt{val}[\texttt{j}]\} : \forall j \rightarrow \texttt{wt}[\texttt{j}] \leq i\right\}$$

.

The optimal solution for a knapsack of capacity W is stored in `sack`[`W`].

Input:	Integers W and n, integer array `wt`, and array `val`
State Space:	An array `sack` where `sack[i]` is the maximum value that we can pack into knapsack of capacity i
Base Case:	$\texttt{sack}[0] = 0$
Inductive Case:	$\texttt{sack}[\texttt{i}] = \max\left\{\texttt{sack}[\texttt{i}-1], \max_{j=0}^{n}\{\texttt{sack}[\texttt{i}-\texttt{wt}[\texttt{j}]]+\texttt{val}[\texttt{j}]\}\right\} : \forall j \rightarrow \texttt{wt}[\texttt{j}] \leq i$
Optimal Solution:	`sack[W]`

This algorithm runs is $O(nW)$ time, as there are W different sub-problems that each take $O(n)$ time to solve.

0/1 Knapsack

In this version of the knapsack problem, each item i can only be placed once inside the knapsack. The naive solution to this is a recursive exhaustive search, where you consider all possible subsets of items where the weight of the subset is at most W. For each item, there are only two possibilities: the item is included in an optimal subset, or it is not. Therefore, the maximum value from n items in a capacity W knapsack can be obtained by either excluding the n^{th} item, in which case, the best solution is one with $n-1$ elements and a capacity W knapsack, or we include the n^{th} item, and add `val[n]` to the best solution that can be obtained from the first $n-1$ elements in a capacity `W-wt[n]` knapsack.

Consider the example from before where $\texttt{val} = [100, 150, 300]$, $\texttt{wt} = [10, 20, 30]$, and $C = 50$. The optimal solution is to use items 2 and 3 to obtain a value of 450.

input : Integers W and n, integer array `wt`, and array `val`

output: The max value of a subset of items that fit into a capacity W knapsack

```
1 Knapsack(W,n,wt,val):
2 if W == 0 or n == 0 then
3 |   return 0
4 end
5 if wt[n] > W then
6 |   return Knapsack(W,n-1,wt,val)
7 end
8 return max{val[n] + Knapsack(W-wt[n],n-1,wt,val),
    Knapsack(W,n-1,wt,val)}
```

Algorithm 6: Naive 0/1 Knapsack

The runtime of this solution is $O(2^n)$, as the recursion depth is n, and there are two recursive calls in each sub-problem. We can see that this method has the familiar problem of wasting time solving the same sub-problems over and over again. So, we can use memoization again to take advantage of dynamic programming and speed this up. However, we cannot use a one-dimensional solutions table as we have in previous problems. Each state has two separate values that define it, the current item n and the knapsack capacity W, so we can use a 2D array to be our state space. Let the 2D array `sack` be the solution space, where `sack[i, j]` is the maximum value we can pack into a knapsack of capacity i, using a subset of items 1 to j. As a base case, we have $\texttt{sack}[\texttt{0}, \texttt{j}] = \texttt{sack}[\texttt{i}, \texttt{0}] = 0$. The inductive case is the same as the naive solution above, just using the previously stored value instead of recursively calling knapsack: $\texttt{sack}[\texttt{i}, \texttt{j}] = \max\{\texttt{val}[\texttt{j}] + \texttt{sack}[\texttt{i} - \texttt{wt}[\texttt{j}], \texttt{j} - \texttt{1}]), \texttt{sack}[\texttt{i}, \texttt{j} - \texttt{1}]\}$.

Input:	Integers W and n, integer array `wt`, and array `val`
State Space:	A 2D array `sack` where `sack[i, j]` is the maximum value we can pack into a knapsack of capacity i, using a subset of items 1 to j
Base Case:	`sack[i, 0]` $= 0\ \forall i \in \{0, \ldots, W\}$, `sack[0, j]` $= 0\ \forall j \in \{0, \ldots, n\}$,
Inductive Case:	$\texttt{sack}[\texttt{i}, \texttt{j}] = \max\{\texttt{val}[\texttt{j}] + \texttt{sack}[\texttt{i} - \texttt{wt}[\texttt{j}], \texttt{j} - 1], \texttt{sack}[\texttt{i}, \texttt{j} - 1]\}$
Optimal Solution:	`sack[W, n]`

The runtime of the above algorithm is only $O(nW)$. Note that it also takes $O(nW)$ extra space to store the 2D solution array. Since when we fill column j in the memoization table, we only need to keep track of the previous column, we can reduce the space constraints by only storing two columns at a time. That is, we only remember column $j-1$ when filling row j, and forget or rewrite row $j-1$ when column j is completely filled. In order to compute the set of items in an optimal solution, we need to use backpointers, which is another 2D array of size $O(nW)$. Using advanced divide and conquer techniques, we can further reduce the space of backpointers to only $O(n)$. We describe this divide and conquer technique below.

Divide and Conquer with Dynamic Programming

In order to reduce the space to store the backpointers, we first argue that we can provide a dynamic programming formulation in 'reverse order', i.e., starting from row n and filling rows of the state space until we reach row 1. For this we let `sack'[i, j]` be the maximum value we can pack into a knapsack of capacity i using a subset of items j through n. As a base case, we have $\texttt{sack}'[\texttt{i}, \texttt{n}+1] = 0,\ \forall i \in \{0, \ldots, W\}$, and $\texttt{sack}'[0, \texttt{j}] = 0,\ \forall j \in \{0, \ldots, n+1\}$. For the inductive case, we have $\texttt{sack}'[\texttt{i}, \texttt{j}] = \max\{\texttt{val}[\texttt{j}] + \texttt{sack}'[\texttt{i} - \texttt{wt}[\texttt{j}]], \texttt{j}+1]), \texttt{sack}'[\texttt{i}, \texttt{j}+1]\}$.

Input:	Integers W and n, integer array `wt`, and array `val`
State Space:	A 2D array `sack'` where `sack'[i,j]` is the maximum value we can pack into a knapsack of capacity i, using a subset of items j through n
Base Case:	$\texttt{sack}'[\texttt{i}, \texttt{n}+1] = 0,\ \forall i \in \{0, \ldots, W\}$, $\texttt{sack}'[0, \texttt{j}] = 0,\ \forall j \in \{0, \ldots, n+1\}$,
Inductive Case:	$\texttt{sack}'[\texttt{i}, \texttt{j}] = \max\{\texttt{val}[\texttt{j}] + \texttt{sack}'[\texttt{i} - \texttt{wt}[\texttt{j}]], \texttt{j}+1], \texttt{sack}'[\texttt{i}, \texttt{j}+1]\}$
Optimal Solution:	`sack'[W,1]`

Note that we only need to store two columns, since when we fill column j we only need to know column $j+1$, and can forget column $j+1$ after column j is filled entirely.

Therefore we have two different dynamic programming formulations to solve this problem. One moving in the forwards direction from column 1 to column n, and the other moving in the reverse direction from column n down to column 1. Combining the two formulations, we can fill from column 1 to column $n/2$ using the 'forwards' dynamic programming formulation, while running the reverse formulation from column n to column $n/2$. Thus at each capacity i knapsack at column $n/2$, we know `sack[i, n/2]` which is the best way to fill a capacity i knapsack with a subset of items 1 through $n/2$, and `sack'[i, n/2` which is the best way to fill a capacity i knapsack with a subset of items $n/2$ through n. Therefore, the best way to fill a capacity i knapsack with all of the items 1 through n is $\texttt{sack}[\texttt{i}, \texttt{n}/2] + \texttt{sack}'[\texttt{i}, \texttt{n}/2]$. Among all knapsacks i, we get the best crossing point index that maximizes $\max_{i=0}^{W} \left\{\texttt{sack}[\texttt{i}, \texttt{n}/2] + \texttt{sack}'[\texttt{i}, \texttt{n}/2]\right\}$.

Let c be the knapsack capacity for which we have the best crossing point, i.e., c is the index i that maximizes the above expression. The very observant reader may note that the solution with value $\texttt{sack}[\texttt{c}, \texttt{n}/2] + \texttt{sack}'[\texttt{c}, \texttt{n}/2]$ need not be feasible. That is, the sizes of the items in solution `sack[c,n/2]` and the sizes in `sack'[c,n/2]` may end up being more than capacity c. Therefore the maximization quantity is only taken over all feasible solutions i:

$$\max_{i=0}^{W} \left\{\texttt{sack}[\texttt{i}, \texttt{n}/2] + \texttt{sack}'[\texttt{i}, \texttt{n}/2]\right\}, \text{for only feasible solutions i}$$

In order to determine if the solution is feasible at an index i, we keep track of the sizes of items in an optimal solution in addition to the value of the items. This array takes the same space as our dynamic programming state space, and can be updated while computing `sack[i, j]` – add item j if $\texttt{val[j]} + \texttt{sack}[\texttt{i} - \texttt{wt[j]}, \texttt{j} - 1] > \texttt{sack[j} - 1]$ (similarly we can update the size while executing the reverse formulation). If the sizes of the optimal solutions at `sack[i,n/2]` and `sack'[i,n/2]` fit within the capacity i knapsack, then we have a feasible solution. Checking for feasibility of a single solution takes only $O(1)$ time, and therefore does not affect the overall time complexity.

Using both the forwards and reverse formulations, we can identify the crossing point $(c, n/2)$. Item $n/2$ is present in an optimal solution if $\texttt{val}[\texttt{n}/2] + \texttt{sack}'[\texttt{c} - \texttt{wt}[\texttt{n}/2], \texttt{n}/2 + 1] > \texttt{sack}'[\texttt{c}, \texttt{n}/2 + 1]$. Next we need to determine the set of items in an optimal solution of items $1, \ldots, n/2$ that lead to the crossing point $(c, n/2)$, and the set of items in an optimal solution of items $n/2 + 1, \ldots, n$ from the crossing point $(c, n/2)$. These are two recursive subproblems that can be solved using divide and conquer.

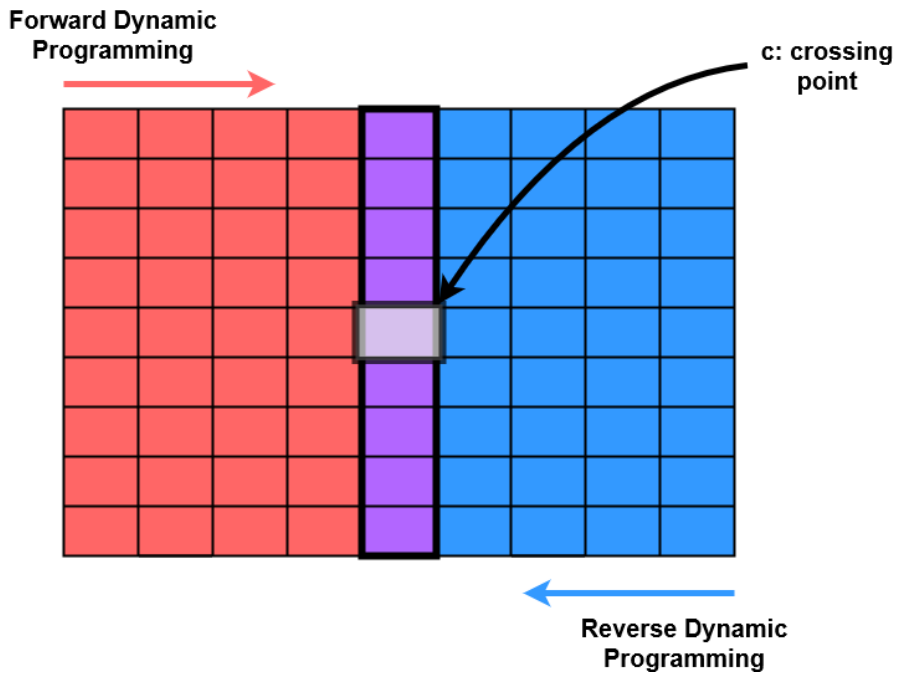


Figure 2.7: Divide and Conquer Dynamic Programming; Reverse and Forwards Formulations

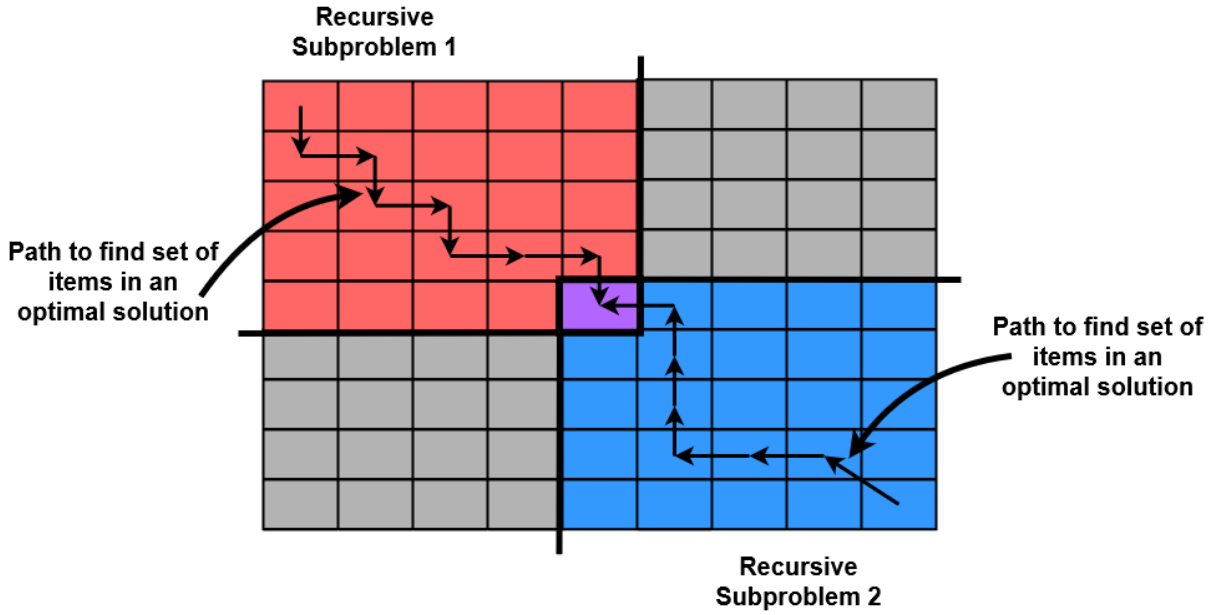


Figure 2.8: Divide and Conquer Dynamic Programming; Finding the Set of Items in an Optimal Solution

It takes $O(nW)$ time to determine the best crossing point $(c, n/2)$, and we have two recursive subproblems each of size $cn/2$ and $(W-c)n/2$ respectively. Expressing the total running time as a recurrence, we get

$$T(nW) = T(cn/2) + T((W-c)n/2) + O(nW)$$

which solves to $O(nW)$. The entire algorithm takes only $O(n)$ space as we only compute two columns at a time both in the forwards and reverse formulations. The items in an optimal solution can also be computed using only $O(n)$ space.

Insight #7: Divide and Conquer can help find the items in an optimal solution with reduced space.

2.6 Computing Shortest Paths

Directed acyclic graphs, or DAGs, are directed graphs that contain no cycles, and they are used to model problems that have precedence constraints, as shown in Figure 2.9. A topological

ordering of a graph gives an ordering of the nodes where all edges point in a consistent direction (see Figure 2.10). Popular algorithms by both Tarjan and Kosaraju can compute a topological ordering in only $O(n + m)$ time, where n is the number of nodes and m is the number of edges. The details of these algorithms can be found in common algorithms textbooks, such as Algorithms by Robert Sedgewick and Kevin Wayne [16].

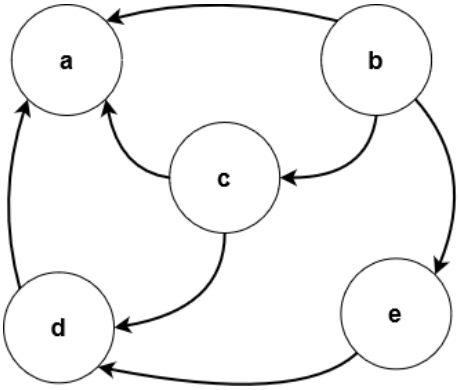


Figure 2.9: Basic Directed Acyclic Graph

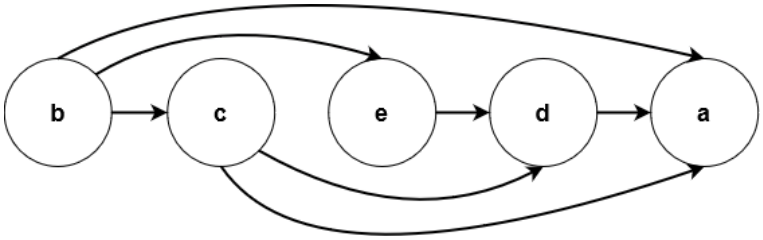


Figure 2.10: Topological Sorting of a DAG

Computing a shortest path in a DAG can be seen as a dynamic program. Let the array `path` be our table of solutions, where `path[i]` is the length of the shortest path from the source node, or the first node in the topological ordering, to node i. As a base case, `path[source]` is 0. For the inductive case, we take the shortest path to an incoming neighbor i, and add the cost of the edge (j, i). The shortest path to i is then the minimum over all such paths – $\texttt{path[i]} = \min\{\texttt{path[j]} + \texttt{cost}(j, i)\}, \forall j : (j, i) \in E$, where E is the edge set of our graph, and $\texttt{cost} : E \to \mathbb{R}$ is the cost function on the edges.

Any dynamic programming problem can be visualized as a problem of computing the shortest path in a DAG. For example, consider the longest increasing subsequence problem from Section 2.4. Let each element in the array correspond to a node, and edges are only directed from smaller elements to larger elements. We let each edge weight be -1. In addition, we add a dummy source node s with edges of weight 0 that point towards every node, and a dummy sink node t with edges of weight 0 coming from every node. One can see that the shortest path in this graph from s to t takes the most number of hops, and therefore corresponds to the longest increasing subsequence of the corresponding array elements. This is demonstrated on the sequence [14, 15, 12, 11, 4, 8, 7, 9] in Figure 2.11. The black arrows are edges from smaller elements to larger elements, with weight -1. The blue arrows are edges from the source node to all elements, with weight 0, and the purple arrows are edge from all elements to the sink node, with weight 0. The elements included in the longest increasing subsequence are highlighted in red.

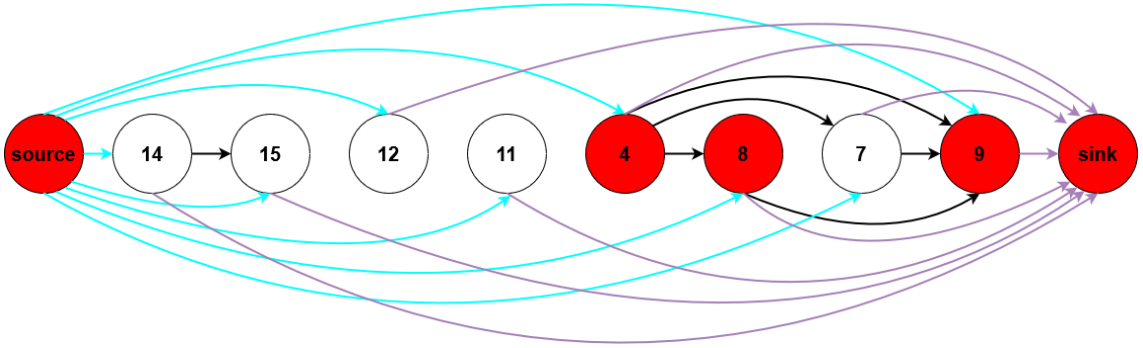


Figure 2.11: Longest Increasing Subsequence Topological Graph

Insight #8: Dynamic Programming = Shortest Path in a DAG

The drawback of interpreting the longest increasing sub-sequence problem as a shortest path algorithm is that we can no longer use the advanced speedups from Section 2.4. However, being able to think about a dynamic programming problem as a shortest path problem is very useful in many situations, and offers more intuitive guidance into the dynamic programming formulation.

Bellman Ford Algorithm

For graphs that are not DAGs, and that have negative edge weights, we can use the famous Bellman-Ford algorithm to compute the shortest path from a single source to all other nodes. This algorithm can also detective negative cycles. A negative cycle is a cycle where the overall sum of the cycle is negative, therefore it is possible to travel the cycle over and over again to decrease the value of the short path infinitely. So shortest paths are not defined in the presence of such cycles.

The Bellman-Ford algorithm is as follows. Let the array `length` be the table of solutions, where `length[i]` is the length of the shortest path from the source node to node `i`. In the beginning, initialize `length[0]` $= 0$ and `length[i]` $= \infty$, $\forall i < n$. We note that all shortest paths follow the triangular inequality, so $\texttt{length[i]} + \texttt{cost(i, j)} \geq \texttt{length[j]}$. If this is not the case, then we can 'tighten' the edge (i, j).

input : Array `length`, where `length[i]` is the shortest path from some source node to i, Array `backpointers` where `backpointers[i]` is the previous node on a shortest path from the source to node i, edge (i, j)

output: Update shortest path cost to node j and its backpointer

```
1 Tighten(length, backpointer, i, j):
2 if length[i] + cost(i, j) < length[j] then
3 |   length[j] = length[i] + cost(i, j)
4 |   backpointer[j] = i
5 end
6 return
```

Algorithm 7: Tighten

The above blackbox can be used whenever the triangular inequality is violated. So the shortest path cost (and its backpointer) to node j will be updated using the tighten operation if a new shortest path is found to j.

In each iteration of the Bellman-Ford algorithm, we iterate over all m edges and tighten

them. We do this repeatedly for a total of $n-1$ iterations.

```
input  : 2D array of edge cost as integers cost, source node s
output: Array length filled with all shortest path costs, and an array
        backpointers with the backpointers to reconstruct the shortest path
1 BellmanFord(cost):
2 length [0] = 0, length [i] = ∞, ∀i ≠ s, backpointers [i] = null, ∀i
3 for i=0 to n do
4 |   foreach edge (i, j) do
5 |   |   Tighten (length, backpointers, i, j)
6 |   end
7 end
8 return length, backpointers
```

Algorithm 8: Bellman-Ford

Note that the Bellman-Ford algorithm can also be interpreted as a dynamic programming algorithm. This is evident when you consider the tightening algorithm. In the first step of the algorithm, most shortest path costs are infinity. So if `length[i]` $= \infty$, `length[j]` $= \infty$, then the new `length[j]` after tightening the edge (i, j) will still equal $infty$. Therefore, the only values of `length` that tighten after the first iteration are those that are the neighbors of the source node s. If some node i has a shortest path that takes only one hop from the source node, then `length[i]` is correct after the first iteration. Applying this logic to the second iteration, we observe that all shortest paths that are two hops from the source are correctly labeled. At the i^{th} iteration of Bellman-Ford, all shortest paths that use i hops from the source are labeled correctly. This also means that the Bellman-Ford algorithm either computes correct shortest paths, or can determine if there are negative cycles in the graph. After the last iteration, we can go over all the edges in the graph one more time and tighten them. If any of the shortest path costs are updated (due to violation of the triangular inequality), then we have a shortest path that takes n hops from the source. So some node on the path must be visited twice and therefore we have a negative cycle.

At this point it is easy to see why this is dynamic programming; at each step of the Bellman-Ford algorithm, the current values being computed rely on the correctness of the values computed before them.

Input: 2D integer array array `cost`

State Space: 2D array `length` where `length[k][i]` is the length of the shortest path from the source node to node `i` using at most k steps

Base Case: `length[k][0]` $= 0, \forall k < n$

`length[k][i]` $= \infty, \forall k < n, \forall i < n$

Inductive Case:

$$\texttt{length[k][i]} = \min \begin{cases} \min\limits_{\forall u \to (u,i) \text{ is an edge}} \{\texttt{length[k-1][u]} \\ \qquad\qquad + \texttt{cost[u,i]}\}, \\ \texttt{length[k-1,i]} \end{cases}$$

Optimal Solution: From source to node `j`: `length[n][j]`

The tightening algorithm takes constant time to check and update each cost, and to do that for all edges takes $O(m)$ time, where m is the number of edges. The tighten algorithm is run a total of n times to check for negative cycles. Therefore, the runtime of the Bellman-Ford algorithm is $O(mn)$.

Floyd-Warshall

We can also compute shortest paths from all nodes to every other node using the famous algorithm of Floyd and Warshall.

Since we want to have available the shortest path length from every node to every other node, we will need a 2D state space to store the length of each path from node `i` to node `j`. Let the array `length` be the 2D table of solutions, where `length[i][j]` is the length of the shortest path from the node `i` to node `j`. In every iteration of the Floyd-Warshall, we compute

the shortest path from nodes i to j that only step into nodes $1, \ldots, k$ as intermediary nodes. In the beginning, we initialize $\texttt{length}[\texttt{i}][\texttt{i}] = 0, \forall i \in \{0, \ldots, n-1\}$ and $\texttt{length}[\texttt{i}, \texttt{j}] = \infty, \forall i, j \in \{0, \ldots, n-1\} \rightarrow i \neq j$. In the k^{th} iteration, we check if the shortest path from node i to node j is shorter if it steps into node k. If so, we update this cost. That is we update $\texttt{length}[\texttt{i}, \texttt{j}]$ if $\texttt{length}[\texttt{i}, \texttt{k}] + \texttt{length}[\texttt{k}, \texttt{j}] < \texttt{length}[\texttt{i}, \texttt{j}]$.

The Floyd-Warshall algorithm is also capable of detecting negative cycles; if at any point `length[i][j]` < 0, then there is a negative cycle.

input : 2D array of edge cost `cost`

output: 2D array of shortest path costs `length`

```
1  FloydWarshall(arr):
2  length [i, i] = 0, ∀i ∈ {0, ..., n − 1}
3  length [i, j] = ∞, ∀i, j ∈ {0, ..., n − 1} → i ≠ j
4  for i ← 0 to n do
5      for j ← 0 to n do
6          for k ← 0 to n do
7              if length[i][k] + length[k][j] < length[i][j] then
8                  length[i][j] = length[i][k] + length[k][j]
9              end
10         end
11     end
12 end
13 return length
```

Algorithm 9: Floyd-Warshall

Unsurprisingly, the Floyd-Warshall algorithm can be interpreted as a dynamic programming problem as well.

Input:	2D integer array array `cost`
State Space:	3D array `length` where `length[k][i][j]` is the length of the shortest path from node `i` to node `j` that only uses nodes 1 through `k` as potential intermediate nodes
Base Case:	$\texttt{length[k][i][i]} = 0,\ \forall k, i \in \{0, \ldots, n-1\}$, $\texttt{length[k][i][j]} = \infty,\ \forall k, i, j \in \{0, \ldots, n-1\} \rightarrow i \neq j$
Inductive Case:	$\texttt{length[i][j]} = \min\{\texttt{length[i][k]} + \texttt{length[k][j]}, \texttt{length[k-1][i][j]}\}$
Optimal Solution:	From node `i` to node `j`: `length[n][i][j]`

2.7 Travelling Salesman

The travelling salesman problem is a famous problem in computer science. Given a set of n cities and distances `dist` between some pair of cities, what is the shortest possible route that visits every city exactly once and returns to the starting city.

An example of a graph of 5 cities and the possible paths is show in Figure 2.12, with the red edges as those that are part of an optimal tour. Black edges are other edges in the graph.

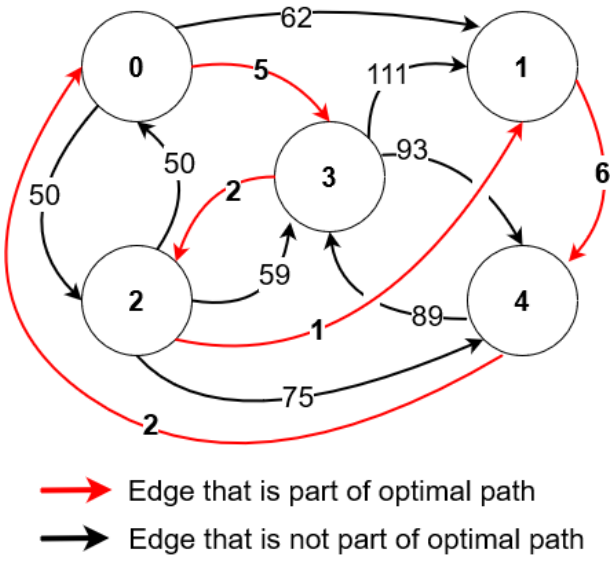


Figure 2.12: Travelling Salesman Tour on a Graph

The naive solution for this is to generate all $(n-1)!$ permutations of cities where city 0 is the starting and ending point, check if the tour is feasible, then measure its cost, while keeping track of the permutation with the minimum cost. This takes $O(nn!)$ time. Using dynamic programming, an algorithm of Held and Karp [3], we can speed this up considerably, although it still takes exponential time and is not practical for even small input sizes. This problem is known to be NP-hard, and only exponential time algorithms exist.

Without loss of generality, we will assume that the cities are numbered 0 through $n-1$, and the starting city is 0. It can be seen that there are overlapping subproblems – the best tour starting with cities 0, 1, 2, and the rest of the $n-3$ cities is similar to the tour starting with cities 0, 2, 1, and the same set of $n-3$ cities. So we can keep track of the last city u, and the set of cities that still need to be visited S. Initially the set S contains all nodes except city 0.

Consider the problem of finding the best path from city u, visiting a set of nodes in S and then reaching the starting city 0. From city u, we ask which city to visit next? We can iterate over all possible cities not yet visited and find the next city v to visit. We are then in a recursive subproblem of finding the best path from city v and a set of unvisited nodes $S-\{v\}$. Once we find the best path from city u to city 0, we can memoize the result so as not to compute this path again.

Figure 2.13 shows a recursive subproblem for the example graph shown in Figure 2.12 above. At node 3, the recursion being calculated is the best path from city 3 through all of the cities that have not yet been visited and back to city 0.

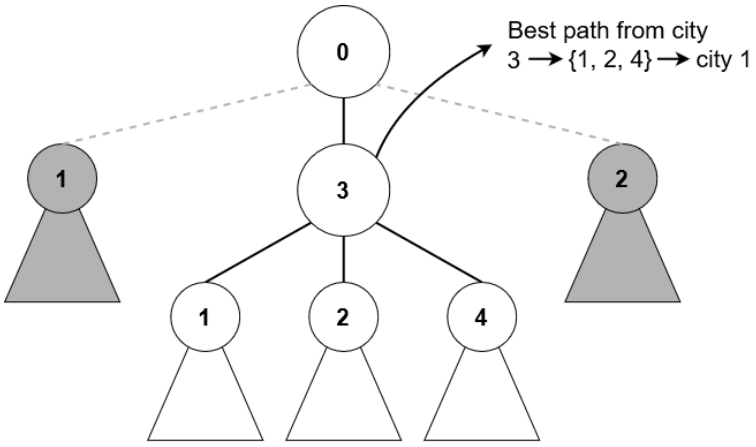


Figure 2.13: Travelling Salesman Recursion Tree, with Current Subproblem in White and Other Subproblems Grayed Out

```
global : 2D array of distances dist, starting city 0, state space tour
input  : city u, a set of unvisited cities S
output: The best path from city u visiting each of the unvisited cities in S and
        reaching the starting city 0

1  Best_Tour(u, S):
2  if S = φ then
3  |   return dist [u,0]
4  end
5  if tour [u, S] already computed then
6  |   return tour [u, S]
7  end
8  mincost = ∞
9  for v ∈ S do
10 |   cost = dist [u, v] + Best_Tour(v, S − {v})
11 |   mincost = min{mincost, cost }
12 end
13 tour [u, S] = mincost
14 return tour [u, S]
```

Algorithm 10: Best Cost Path from City u to City 0 Visiting all Nodes in S

There are $O(2^{n-1} \times n)$ different subproblems to compute, and each one takes $O(n)$ time to iterate over the set of unvisited cities. Therefore the total time is $O(2^{n-1} \times n^2)$. Even for small values of n this could take a lot of time, however this gives a significant speed up from the naive brute force solution of enumerating all possible permutations. Practically, we can implement the set S as a bit vector and represent as a single integer for $n \leq 64$. But the runtime is only suited for $n \leq 20$. The above solution is us another example of Insight #2, dynamic programming = recursive exhaustive search + memoization.

Chapter 3

Dynamic Programming Problems and Insights

This chapter will expand on the insights gathered from the last chapter. Dynamic programming has been quite useful in solving problems involving combinatorial counting, strings and on trees. We describe dynamic programming solutions to problems in these domains as they provide insights into the kind of problems dynamic programming is largely useful at solving. We also explore selected problems from the 2019 North American ICPC contest that can be solved using dynamic programming.

3.1 Combinatorial Counting Problems

Combinatorial mathematics is the study of counting discrete objects from a large set of objects that satisfy certain properties. One way to do so is to generate all possible objects and count the number of them that satisfy the given property. This may be too slow if the solution space comes from finding all subsets (which take $O(2^n)$ time), all permutations (which takes $O(n!)$ time), or generating all combinations of k objects (which take $O(\binom{n}{k}) = O(n^k)$ time). The field of combinatorial optimization further asks us to minimize or maximize an objective function over these sets of objects. The use of memoization allows us to solve some combinatorial counting and optimization problems quickly. Therefore, the technique of dynamic programming lends itself well to solving these types of problems.

Binomial Coefficients

Consider the problem of computing the total number of ways to choose k items from n items. Note that we are forming sets of k items, so the order of the items do not matter. We can use a recursive approach – to choose k items from n items, we can either keep the first item and choose $k-1$ items from $n-1$ items, or exclude the first item and choose k items from $n-1$ items.

input : Size of the original set `n`, size of subsets `k`

output: The number of ways to choose `k` items from `n` items

```
1 Choose(n, k):
2 if n < k then
3 |   return 0
4 end
5 if k == 0 or k == n then
6 |   return 1
7 end
8 return Choose(n − 1, k − 1) + Choose(n − 1, k)
```

Algorithm 11: Naive Recursive Solution to Choosing k Items from n Items

This solution takes $O(2^n)$ time, as the recursion depth is n, and each subproblem has two recursive calls. This solution has the following recursion tree:

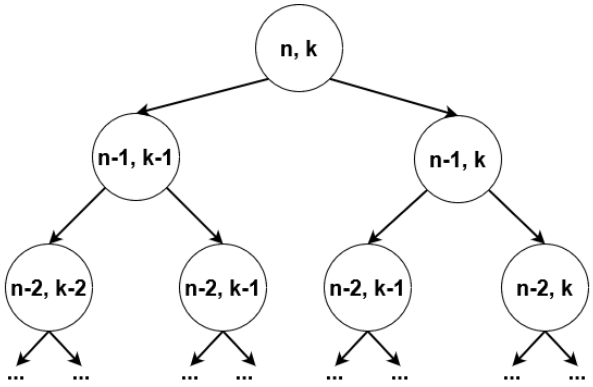


Figure 3.1: Recursive Binomial Coefficient Tree.

This tree shows us a familiar problem of solving the same subproblems multiple times, so to speed this solution we can take advantage of memoization with a two-dimensional state space. Let the 2D array `choices` be the solution space, where `choices[i, j]` is the total number of ways to choose j items from i items. As base cases, `choices[i, 0] = 1`, and `choices[i, i] = 1`. The recursive case is the same as the naive solution above, but we use memoization to lookup solutions to already computed subproblems.

Input:	Size of the original set `n`, size of subsets `k`
State Space:	A 2D array `choices` where `choices[i,j]` is the total number of ways to choose `j` items from `i` items
Base Case:	$\texttt{choices}[\texttt{i}, 0] = 1, \forall i \in \{0, \ldots, n\}$, $\texttt{choices}[\texttt{i}, \texttt{i}] = 1, \forall i \in \{0, \ldots, n\}$
Inductive Case:	$\texttt{choices}[\texttt{i}, \texttt{j}] = \texttt{choices}[\texttt{i} - \texttt{i}, \texttt{j} - 1] + \texttt{choices}[\texttt{i} - \texttt{i}, \texttt{j}]$
Optimal Solution:	`choices[n,k]`

The runtime of the above algorithm is only $O(nk)$. Note that it also takes $O(nk)$ extra space to store the 2D solution array. Similar to the 0/1 Knapsack problem, when we fill row i in the memoization table, we only need to keep track of the previous row, so we can reduce the space constraints by only storing two rows at a time. We only remember row $i-1$ when filling row i, and forget or rewrite row $i-1$ when row i is completely filled.

Counting Derangements

The counting derangements problem, usually called the "hat check problem", is the following: given n hats, find the total number of ways in which those hats can be given to n people so that no hat is returned to its owner. To solve, note that each person p_i may receive any of the hats h_j that is not theirs. So if p_i receives h_j, then h_j's original owner either receives p_i's hat, or somebody else's hat.

Let's say the yellow hat in Figure 3.2 is person p_i's hat, and the blue hat is person p_j's hat.

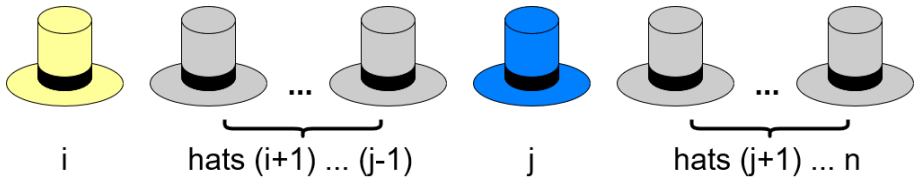


Figure 3.2: Hats i Through n

So, as stated above, person p_i receives person p_j's hat.

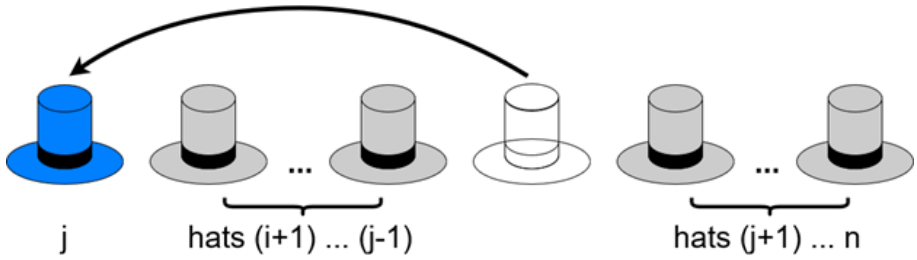


Figure 3.3: Person i Receives Hat j

So the problem has two cases: p_j receives hat h_i, shown in Figure 3.4.

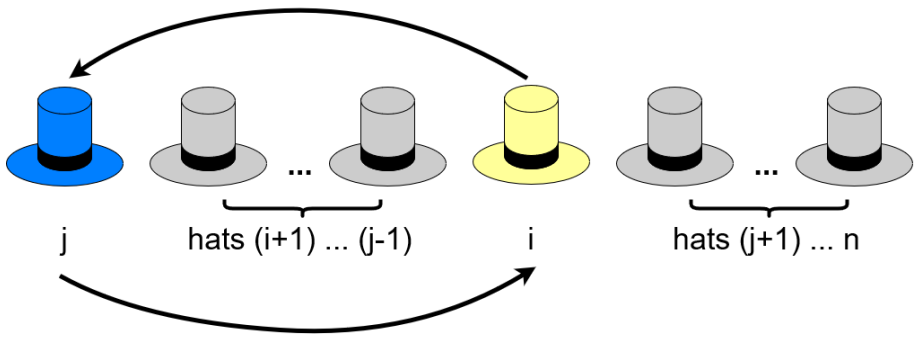


Figure 3.4: Person j Receives Hat i

In this case there are two less hats and people, as both p_i and p_j have hats, so the problem reduces to $i-2$ people and $i-2$ hats.

The second case in which p_i does not receive hat h_j, so hat h_j is still available to be swapped, shown in darker yellow in Figure 3.5.

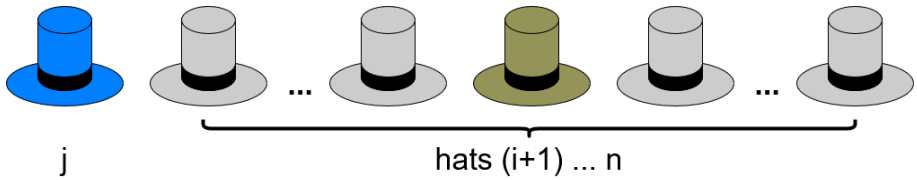


Figure 3.5: Nobody Receives Hat i

In this case, hat h_i doesn't end up with person p_j, but we will still "give" the hat to person p_j. The yellow hat h_i is temporarily given to person p_j, but they will not end up with the yellow hat at the end of the problem. For all recursive problems, we consider the yellow hat h_i to be person p_j's original hat, and they will give away the hat and get a new one in some future subproblem calculated by the solution. This is shown in Figure 3.5 by graying out the original yellow hat. Here the problem reduces to $i-1$ people and $i-1$ hats.

This is multiplied by $(i-1)$, as you could choose a different p_i for each step. Like most counting problems, the naive solution is a recursive exhaustive search, where at each level i, the total number of ways to incorrectly return i hats is $i-1$ times the number of derangements for $i-1+$ the number of derangements for $i-2$.

input : The number of hats `n` and people `n`

output: The number of ways to incorrectly return `n` hats to `n` people

```
1 Derangements(n):
2 if n == 1 or n == 2 then
3 |   return n - 1
4 end
5 return (n − 1)[Derangements(n - 1) + Derangements(n - 2)]
```

Algorithm 12: Naive Derangements

The runtime of this algorithm takes $O(2^n)$ time, as the recursion depth is n, and each subproblem has two recursive calls.

At this point, we can tell that this has the common problem most exhaustive recursive searches have, that of computing the same subproblems over and over again. We can fix this

using memoization and a one-dimensional solution table. Let the 1D array `derangements` be the solution space, where `derangements[i]` is the total number of ways i hats can be distributed to i people so that nobody gets their original hat back. Let the base case be `derangements[1] = 0`, `derangements[2] = 1`. The recursive case is the same as the naive solution above.

Input:	The number of hats and people `n`
State Space:	A 1D array `derangements` where `derangements[i]` is the total number of ways `i` hats can be distributed to `i` people so that nobody gets their original hat back.
Base Case:	`derangements[1] = 0`, `derangements[2] = 1`
Inductive Case:	$\texttt{derangements}[\texttt{i}] = (i-1)[\texttt{derangements}[\texttt{i}-1] + \texttt{derangements}[\texttt{i}-2]]$
Optimal Solution:	`derangements[n]`

This solution takes $O(n)$ time, as each subproblem takes constant time to compute. This solution also takes $O(n)$ extra storage space for the solution table `derangements`, but it is possible to reduce the storage space to constant extra space. If we do this problem bottom-up as described before, starting by calculating `derangements[0]` and working our way up to `derangements[n]`, for each step `i` we only need to keep track of the last two steps, `derangements[i - 1]` and `derangements[i - 2]`.

Throwing Dice

Given n six sided dice, what is the total number of ways to get x, where x is the sum of values on each face when all the dice are thrown. For reference, Figure 3.6 below shows all the possible sides of a six sided die.

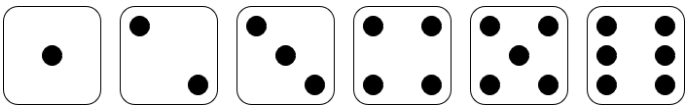


Figure 3.6: All Faces on a Six-sided Die

An example, suppose there are three dice and the sum is 6, then there are 9 different ways as shown in Figure 3.7 below.

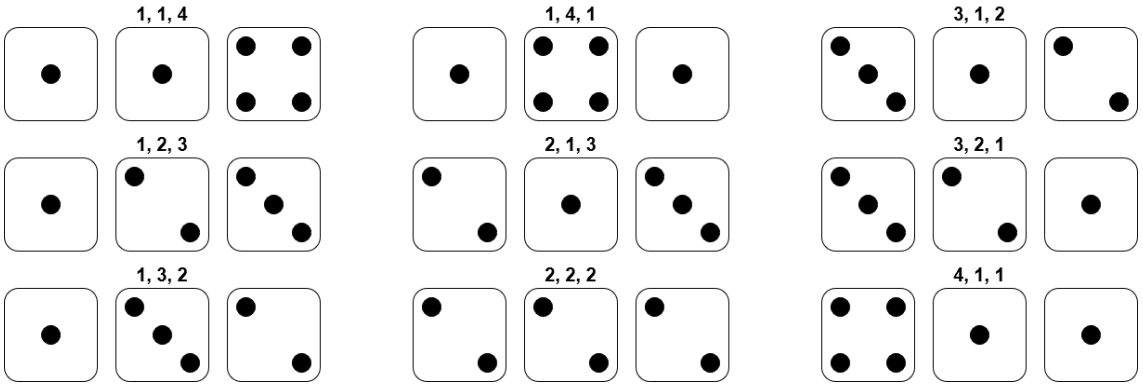


Figure 3.7: All Possible Combinations of Three Dice Whose Faces Sum to 6

The naive, exhaustive search solution is as follows: generate all possible combinations of values from n dice, count the number of combinations that sum to x. We can think of this problem as throwing n dice independently. That is, after throwing the first dice, which can have any of the six numbers as outcomes, we have a sub-problem of throwing $n - 1$ dice and getting a sum of $x - i$, where i is the outcome of the first dice.

```
input  : Number of dice n, sum to obtain x
output: Number of ways to obtain a sum x with n 6 sided dice

1  Sum(n, x):
2  numberOfWays = 0
3  if x < 1 then
4  |  return 0
5  end
6  if n == 1 and x < n then
7  |  return 1
8  end
9  for i ← 1 to 6 do
10 |  numberOfWays += Sum(n − 1, j − i)
11 end
12 return numberOfWays
```

Algorithm 13: Naive 6 Sided Dice

This solution runs in $O(6^n)$ time, as the recursion depth is n, and each subproblem has six recursive calls. Using memoization, we can reduce the runtime of this algorithm. Let the 2D array `sums` be the solution space, where `sums[i, j]` be the total number of ways sum j can be made with i dice. Let the base case be $\texttt{sums}[1, \texttt{j}] = 1, \forall j \in \{1, \ldots, 6\}$. The recursive case is the same as the naive solution above, just using the pre-calculated state space instead of doing the calculations each time: $\texttt{sums}[\texttt{i}, \texttt{j}] = \sum_{k=1}^{6} \texttt{sums}[\texttt{i} - 1, \texttt{j} - \texttt{k}]$.

Input:	Number of dice `n`, sum to obtain `x`
State Space:	A 2D array `sums` where `sums[i, j]` is the total number of ways sum j can be made with i dice
Base Case:	$\texttt{sums}[1, \texttt{j}] = 1, \forall j \in \{1, \ldots, 6\}$
Inductive Case:	$\texttt{sums}[\texttt{i}, \texttt{j}] = \sum_{k=1}^{6} \texttt{sums}[\texttt{i} - 1, \texttt{j} - \texttt{k}]$
Optimal Solution:	`sums[n, x]`

The runtime of this solution is $O(n * x)$, and it also requires extra storage space of size $O(nx)$.

3.2 Dynamic Programming on Strings

Dynamic programming is commonly used for problems on strings. These problems arise in the field of bioinformatics, where we are given DNA or RNA sequences and are asked to find certain types of alignments between them. We look at three well-studied problems in this field.

Edit Distance

Given two strings `string1` and `string2` and three operations that can be performed on the strings – inserting a character, removing a character, and replacing a character, find the minimum number of edits required to turn `string1` into `string2`.

For example, let "Friday" be `string1` and "Thursday" be `string2`. The optimal solution for this problem is four steps; two inserts and two replaces. Figure 3.8 shows the steps to turn `string1` "Friday" into `string2` "Thursday".

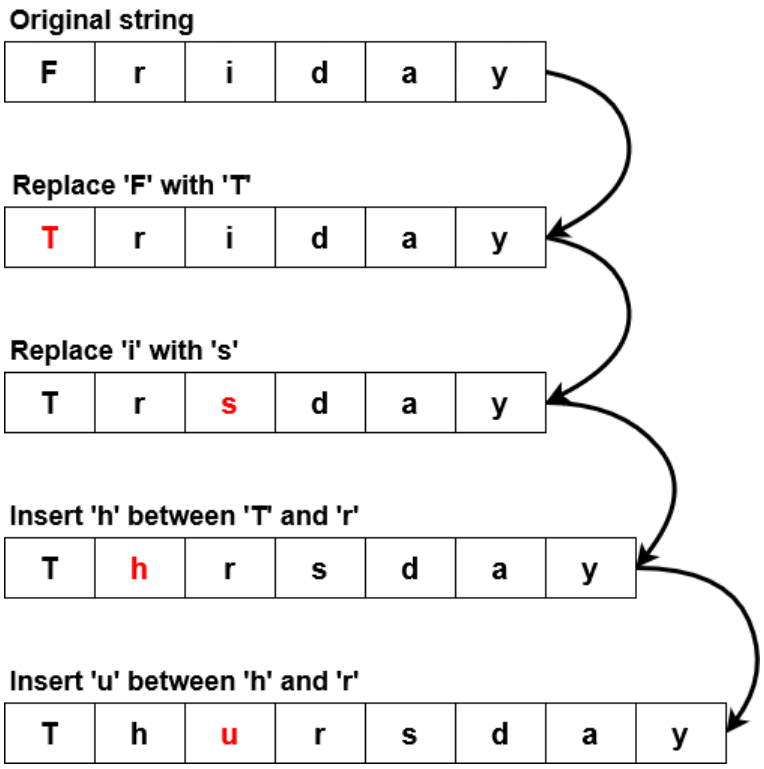


Figure 3.8: Steps to Turn "Friday" into "Thursday"

To solve, let `l1` be the length of `string1` and `l2` be the length of `string2`. Start on the left side, or first character of both strings, and for each character there are two cases. If the first characters of both strings are the same, we do not need to use an operation, so recurse on a subproblem of transforming `string1` of length `l1-1` to `string2` of length `l2-1`, both strings from index 1. If these characters are not the same, then we can either insert a new character, specifically the first character of `string2` to `string1`, and recurse on a subproblem of transforming `string1` to the rest of `string2`, or remove a character, specifically the first character of `string1`, and recurse on a subproblem of transforming the rest of `string1` to `string2`. The last case is that we replace the first character of `string1` to the first character of `string2`, and recurse on a subproblem of transforming the rest of `string1` to `string2`. If we insert, remove or replace a character, we spend 1 unit of 'cost' to make that transformation, and spend the rest of the cost in the recursive subproblems. The minimum cost of these three cases is then returned.

input : Two strings `string1` and `string2`, their lengths `l1` and `l2`, the current index of each `i1` and `i2`

output: The minimum number of edits to get from `string1` starting at `i1` to `string2` starting at `i2`

```
1  EditDistance(string1, string2, l1, l2, i1, i2):
2  if i1 == l1 then
3  |  return l2 − i2
4  end
5  if i2 == l2 then
6  |  return l1 − i1
7  end
8  if string1[i1] == string2[i2] then
9  |  return EditDistance(string1, string2, l1, l2 i1 + 1, i2 +
   |   1)
10 end
11 insert = EditDistance(string1, string2, l1, l2, i1, i2 + 1)
12 remove = EditDistance(string1, string2, l1, l2, i1 + 1, i2)
13 replace = EditDistance(string1, string2, l1, l2, i1 + 1, i2 +
   1)
14 distance = min{insert, remove, replace}
15 return 1 + distance
```

Algorithm 14: Naive Edit Distance

This solution has a time complexity of $O(3^{\texttt{l1}+\texttt{l2}})$ time, as the recursion depth is `l1`+`l2`, and each subproblem has three recursive calls. As before, we can use memoization to speed this up. Let the 2D array `distance` be the solution space, where `distance[i, j]` is the minimum edit distance to turn `string1[i1..]` into `string2[i2..]`. Let the base cases be $\texttt{distance}[0, \texttt{j}] = l2 - j, \forall j \in \{0, \ldots, l2\}$ and $\texttt{distance}[\texttt{i}, 0] = l1 - i, \forall i \in \{0, \ldots, l1\}$. The recursive case is the same as the recursive solution above: $\texttt{distance}[\texttt{i}, \texttt{j}] = distance[i+1, j+1]$

if `string1`[i] == `string2`[j], `distance`[i, j] = $1 + \min\{$`distance`[i, j + 1], `distance`[i + 1, j], `distance`[i + 1, j + 1]$\}$ otherwise.

Input: Two strings `string1` and `string2`, their lengths `l1` and `l2`

State Space: A 2D array `distance` where `distance[i, j]` is the minimum edit distance to turn `string1[i1..]` into `string2[i2..]`.

Base Case: `distance`[0, j] $= l2 - j, \forall j \in \{0, \ldots, l2\}$,

`distance`[i, 0] $= l1 - i, \forall i \in \{0, \ldots, l1\}$

Inductive Case:

$$\texttt{distance}[\texttt{i}, \texttt{j}] = \begin{cases} \texttt{distance}[\texttt{i}+1, \texttt{j}+1], \\ \qquad \text{if } \texttt{string1[i]} == \texttt{string2[j]}, \\ 1 + \min(\texttt{distance}[\texttt{i}, \texttt{j}+1], \texttt{distance}[\texttt{i}+1, \texttt{j}], \\ \qquad \texttt{distance}[\texttt{i}+1, \texttt{j}+1]), \\ \qquad \text{otherwise} \end{cases}$$

Optimal Solution: `distance[l1, l2]`

The runtime of this solution is $O($`l1` $\times$ `l2`$)$, and it also requires extra storage space of size $O($`l1` $\times$ `l2`$)$. To know the actual set of operations (insert, remove or replace), we can use backpointers as discussed in Chapter 2.

Longest Common Subsequence

Given two strings `string1` and `string2`, find the length of the longest subsequence that both strings have in common. Recall from Chapter 2.4 that a subsequence need not be contiguous.

For example, let "mistake" be `string1` and "staircase" be `string2`. The longest common subsequence between `string1` and `string2` in this case is "stae" highlighted in red in Figure 3.9 below.

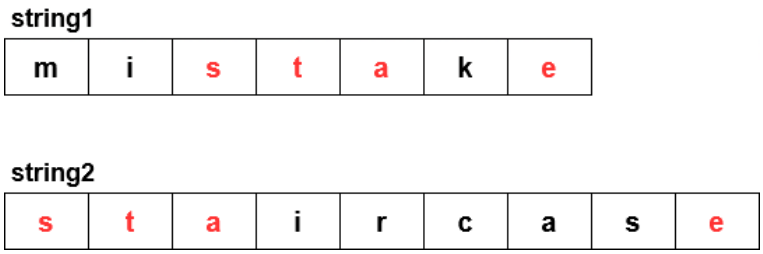


Figure 3.9: Longest Common Subsequence between Strings "mistake" and "staircase"

To solve, let `l1` be the length of `string1` and `l2` be the length of `string2`. We can solve this problem using a two-dimensional state space. Let `length` be the solution space, where `length[i, j]` is the length of the longest common subsequence of `string1[1..i]` and `string2[1..j]`. That is, it is the longest common subsequence between the first i characters of `string1` and the first j characters of `string2`. As base cases, we have $\texttt{distance}[0, \texttt{j}] = 0, \forall j \in \{1, \ldots, l2\}$ and $\texttt{distance}[\texttt{i}, 0] = 0, \forall i \in \{1, \ldots, l1\}$. For the inductive case, there are two options; if `string1[i]` and `string2[j]` match, then we recurse on a subproblem of finding the longest common subsequence between the first $i - 1$ characters of `string1` and the first $j - 1$ characters of `string2`. We add one to reflect that the characters matched. Otherwise, we consider substrings that end on different indices that might match with the current indices, namely `length[i-1, j]` and `length[i, j-1]`. This gives us $\texttt{length}[\texttt{i}, \texttt{j}] = 1 + \texttt{length}[\texttt{i} - 1, \texttt{j} - 1]$ if $\texttt{string1}[\texttt{i}] == \texttt{string2}[\texttt{j}]$, and $\texttt{length}[\texttt{i}, \texttt{j}] = \max\{\texttt{length}[\texttt{i} - 1, \texttt{j}], \texttt{length}[\texttt{i}, \texttt{j} - 1]\}$ otherwise.

Input: Two strings `string1` and `string2`, their lengths `l1` and `l2`

State Space: A 2D array `length` where `length[i, j]` is the length of the longest common subsequence of `string1[1..i]` and `string2[1..j]`

Base Case: $\texttt{length}[0, \texttt{j}] = 0, \forall j \in \{1, \ldots, l2\}$,

$\texttt{length}[\texttt{i}, 0] = 0, \forall i \in \{1, \ldots, l1\}$

Inductive Case:

$$\texttt{length}[\texttt{i}, \texttt{j}] = \begin{cases} 1 + \texttt{length}[\texttt{i} - 1, \texttt{j} - 1], \\ \qquad\qquad \text{if } \texttt{string1}[\texttt{i}] == \texttt{string2}[\texttt{j}] \\ \max\{\texttt{length}[\texttt{i} - 1, \texttt{j}], \texttt{length}[\texttt{i}, \texttt{j} - 1]\}, \\ \qquad\qquad \text{otherwise} \end{cases}$$

Optimal Solution: `length[l1, l2]`

This solution has a runtime of $O(l1 \times l2)$, and it also requires extra storage space of size $O(l1 \times l2)$. As before, we can reduce the space by storing only two rows at a time. The actual subsequence can be computed using backpointers. Moreover, using divide and conquer techniques discussed in Chapter 2.5, we can compute the elements in the longest common subsequence with reduced space.

String Alignment

Given two strings `string1` and `string2`, a value `gapscore`, and a similarity function between characters `similarity(x, y)`, find the maximum alignment score.

For example, make `string1` = "GACC", `string2` = "CAC", `gapscore` = -20, and

$$\texttt{similarity}(\texttt{x}, \texttt{y}) = \begin{cases} 20, & \text{if } \texttt{x} == \texttt{y} \\ -10, & \text{otherwise} \end{cases}$$

In this example, the best alignment value is -10, and the two possible alignments that give that score are "GACC" / "CAC-" and "GACC" / "CA-C".

Let the state space be `align`, where `align[i, j]` is the maximum score aligning the two strings `string1[1..i]` and `string2[1..j]`. For the base case, set $\texttt{align}[0, 0] = 0$. To compute `align[i, j]`, there are three possibilities. The first is to match `string1[i]` to `string2[j]`; in this case we get the similarity score for `string1[i]` and `string2[j]` and add it to the previous alignment value for `string1[i-1]` and `string2[j-1]`.

The second possibility is to introduce a gap in `string1`; in this case we get the previous alignment value for `string1[i]` and `string2[j-1]` and add it to `gapscore`. The final possibility is to introduce a gap in `string2`; in this case we get the previous alignment value for `string1[i-1]` and `string2[j]` and add it to `gapscore`. Pick the maximum of these three values to get the value for the current subproblem. This is shown as $\texttt{align}[\texttt{i}, \texttt{j}] = \max\{\texttt{align}[\texttt{i} - 1, \texttt{j} - 1] + \texttt{similarity}(\texttt{string1}[\texttt{i}], \texttt{string2}[\texttt{j}]), \texttt{align}[\texttt{i}, \texttt{j} - 1] + \texttt{gapscore}, \texttt{align}[\texttt{i} - 1, \texttt{j}] + \texttt{gapscore}\}$.

Input:	Two strings `string1` and `string2`, their lengths `l1` and `l2`, a similarity function `similarity(x, y)`,
State Space:	A 2D array `align` where `align[i, j]` is the maximum score aligning the two strings `string1[1..i]` and `string2[1..j]`
Base Case:	`align[1, 1]` $= 0$
Inductive Case:	`align[i,j]` $= \max\{\texttt{align}[\texttt{i} - 1, \texttt{j} - 1] + \texttt{similarity}(\texttt{string1}[\texttt{i}], \texttt{string2}[\texttt{j}]), \texttt{align}[\texttt{i}, \texttt{j} - 1] + \texttt{gapscore}, \texttt{align}[\texttt{i} - 1, \texttt{j}] + \texttt{gapscore}\}$
Optimal Solution:	`align[l1, l2]`

This solution has a runtime of $O(l1 * l2)$, and it also requires extra storage space of size $O(l1 * l2)$. Using similar techniques discussed previously, we can reduce the space complexity.

3.3 Dynamic Programming on Trees

The technique of dynamic programming can be used on trees as trees are recursive structures. For this, we can process the nodes in a tree with either a pre or post order traversal, and construct solutions to the original problem from its recursive subproblems on subtrees. We explore two popular problems, that of computing the maximum independent set and the maximum weighted independent set on trees in this section, and solve an interesting dynamic programming problem on trees from the 2019 ICPC contest in the next section.

Maximum Independent Set

Given a graph, the maximum independent set (MIS) problem asks to find the largest subset of nodes S such that two nodes $x, y \in S$ are not adjacent to each other. The MIS problem is NP-Hard on general graphs, but the field of graph theory is filled with results, even linear time algorithms on special graph classes like trees.

To compute a MIS of a tree rooted at a node i, we observe that there are two possible cases: a MIS includes node i along with some subset of nodes from its subtree, or it excludes node i and only includes nodes from its subtrees. We can solve this problem using dynamic programming; let the state space be a 2D array of nodes `set`, where `set[i,1]` is the size of a MIS of the subtree rooted at node i that includes node i, and `set[i,2]` be the size of a MIS of the subtree rooted at node i that excludes node i. As a base case, we have $\texttt{set}[\texttt{i}, 0] = 1$, and $\texttt{set}[\texttt{i}, 1] = 0$ if node `i` is a leaf. To compute `set[i,1]`, we cannot include any of the children, so we must extend from all `set[j, 2]`, where j is a child of i. So we have $\texttt{set}[\texttt{i}, 1] = 1 + \sum_{\text{j is a child of i}} \texttt{set}[\texttt{j}, 2]$. To compute `set[i,2]`, we pick the larger of `set[j, 1]` or `set[j, 2]` for each child of node `i`. So we have $\texttt{set}[\texttt{i}, 2] = \sum_{\text{j is a child of i}} \max\{\text{set[j,1], set[j,2]}\}$.

The final solution is the larger of the two solutions at the root node r, which is max{set[r, 1], set[r, 2]}. To obtain the elements in the MIS, we can now use a preorder traversal of the tree starting from the root node. If $\texttt{set}[\texttt{r}, 1] \geq \texttt{set}[\texttt{r}, 2]$, then include r into the MIS and recursively find the sets on the grandchildren of r. Otherwise r is not part of the MIS and we recursively find the sets on the children of r.

Input:	A tree `tree` of size `n`
State Space:	A 2D array of nodes `set`, where `set[i,1]` is the size of a MIS on node `i's` subtree that includes `i`, and `set[i,2]` is the size of a MIS on node `i's` subtree that excludes `i`.
Base Case:	`set[i, 1]` $= 1$, `set[i, 2]` $= 0$ for all leaves i
Inductive Case:	$\texttt{set}[\texttt{i},1] = 1 + \sum_{\text{j is a child of i}} \texttt{set}[\texttt{j},2]$, $\texttt{set}[\texttt{i},2] = \sum_{\text{j is a child of i}} \max\{\texttt{set}[\texttt{j},0], \texttt{set}[\texttt{j},1]\}$
Optimal Solution:	$\max\{set[r,1], set[r,2]\}$ for root node r

It can be seen that at each node, we spend time proportional to degree of that node. So the total time is $\sum_i \texttt{degree}(\texttt{i})$. In a tree this is $O(n)$.

Maximum Weighted Independent Set

It is also possible to find a maximum weighted independent set (MWIS), where each node has a weight attached to it, and we would like to find a largest independent set that maximizes the sum of the weights of all nodes in the set. We can solve this problem on a tree using dynamic programming; let the state space be a 2D array of nodes `sum`, where `sum[i,1]` is sum of weight of all nodes in a MWIS on `i's` subtree where node i is included, and `sum[i,2]` be the sum of weight of all nodes in a MWIS on `i's` subtree where node i is excluded from the MWIS. We let the function $w(i)$ be the weight of node i. As a base case, we have $\texttt{sum}[\texttt{i},1] = w(i)$, and $\texttt{sum}[\texttt{i},2] = 0$ if node i is a leaf. For the inductive case, we have $\texttt{sum}[\texttt{i},1] = w(i) + \sum_{\text{j is a child of i}} \texttt{sum}[\texttt{j},2]$ and $\texttt{sum}[\texttt{i},2] = \sum_{\text{j is a child of i}} \max\{\text{sum[j,1], sum[j,2]}\}$.

The optimal solution is found in $\max\{sum[r,1], sum[r,2]\}$, where r is the root node of the tree. We can compute the actual nodes in the MWIS by using a preorder traversal similar to computing a MIS.

Input:	A tree `tree` of size `n`, a weight function w where $w(i)$ is the weight of node i
State Space:	A 2D array of nodes `sum`, where `sum[i,1]` is sum of the weights of a MWIS node `i's` subtree, where node `i` is included, and `sum[i,2]` is the sum of weights of a MWIS of node `i's` subtree, where node `i` is excluded
Base Case:	$\texttt{sum}[\texttt{i}, 1] = w(i)$, and $\texttt{sum}[\texttt{i}, 2] = 0$, if i is a leaf
Inductive Case:	$\texttt{sum}[\texttt{i}, 1] = w(i) + \sum_{\text{j is a child of i}} \texttt{sum}[\texttt{j}, 2]$, $\texttt{sum}[\texttt{i}, 2] = \sum_{\text{j is a child of i}} \max\{\text{sum[j,1], sum[j,2]}\}$
Optimal Solution:	max{sum[r, 1], sum[r, 2]} for root r

Similar to computing a MIS, the runtime of this algorithm is $O(n)$.

3.4 Dynamic Programming Problems in the ICPC

In this section, we describe some of the problems from the 2019 ICPC North American Regional competitions. Our experience with programming competitions is that problems that use dynamic programming are very common. In the past, there have been at least two problems in every contest that applies this technique. However, in the 2019 ICPC regionals over twelve contests, we have only identified four problems that use this technique. There may be two reasons for the surprisingly low number of dynamic programming problems in 2019. Firstly, some of these problems appear in multiple contests, and secondly, we have been able to solve some problems using other techniques even if a dynamic programming solution exists. We provide detailed editorials for these problems and describe some of the insights that may be useful in solving these with dynamic programming.

Just Passing Through

Imagine going eastward along a two dimensional terrain. We are given a 2D array `map` with `r` rows and `c` columns, where `map[i, j]` gives an elevation, the height of the terrain at row

i and column j. Some of these grid cells are "un-passable" and denoted with a -1, while all other grid cells contain a positive integer height. A grid cell (i, j) is a "pass" if `map[i-1, j]` and `map[i+1, j]` are strictly larger than `map[i, j]`, and `map[i, j-1]` and `map[i, j+1]` are strictly smaller than `map[i, j]`. We would like to traverse the terrain from West (column 1) to East (column c), but we would like to traverse through exactly n passes. From each grid cell (i, j), we can only step into a grid cell East that is adjacent to it, that is to grid cells $(i-1, j+1)$ or $(i, j+1)$ or $(i+1, j+1)$. If we assume that the effort or cost to pass a grid cell (i, j) is proportional to the height of the terrain at that grid cell, then we would like to find a minimum cost path from any point in column 1 to any point in column c. For more information on the problem, please see [11].

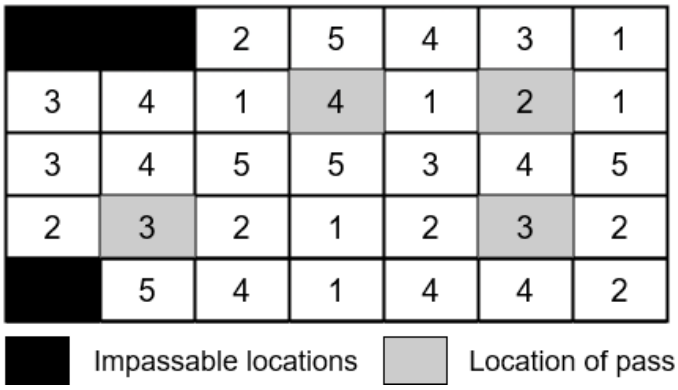


Figure 3.10: Example Elevation Map

Figure 3.10 shows an example of an elevation map. An optimal path where we want to travel through exactly 2 passes is shown in the figure below. The total cost of this path is 14.

Figure 3.11: An Optimal Path

As we want to find an optimal path that minimizes its cost, we see that this is a shortest path problem, and looking at this problem as computing a shortest path in a directed acyclic graph from Insight #8 will be helpful. Thus we transform the input 2D array into a graph by creating a dummy start node 0 and a dummy sink node $t = rc+1$. Each grid cell in the map is a node in the graph. We create r edges from node 0 to all nodes in the first column, and create r edges from all nodes from column c to node t. We also create edges from (i, j) to $(i-1, j+1)$, $(i, j + 1)$ and $(i + 1, j + 1)$. In this graph, the weight of a node is its elevation. We can move the weight of a node to its incoming edges. So the edges from node 0 to $(1, i)$ will contain the cost of the elevation at $(1, i)$. We set the weight of node t to 0, so all edges going into the sink node has a cost of 0.

We can see that a shortest path from the source to the sink node almost solves the problem. The only catch is that we need to step into exactly n passes. The shortest path from the source to the sink may contain more than n passes. In order to solve this, we will create $n+1$ copies of the same graph we created, one each for the number of passes we have travelled through. So we have graphs $\texttt{map}_0, \ldots, \texttt{map}_{n+1}$. From each grid cell (i, j) that steps into a pass (x, y) on the k^{th} such graph, we step from $\texttt{map}_k[i, j]$ to $\texttt{map}_{\texttt{K+1}}[\texttt{x}, \texttt{y}]$ with a weight of the grid cell (x, y). This means that the only way to step from a level k graph to a level $k + 1$ graph is to step into a pass. Passes in the level $n + 1$ graph will have no outgoing edges. Now if we compute a shortest path from the source node in the level 0 graph to the sink node in the level

$n+1$ graph, we find the minimum cost path that steps through exactly n passes. Note that the graph produced is a DAG, as edges either go from left to right or from one level to the next. The total space for the graph is $O(rcn)$ – there are $O(rcn)$ nodes and each node has at most three edges, so the number of edges is also $O(rcn)$. For completeness, we provide a dynamic programming formulation below, but a shortest path black box that takes the above graph will suffice.

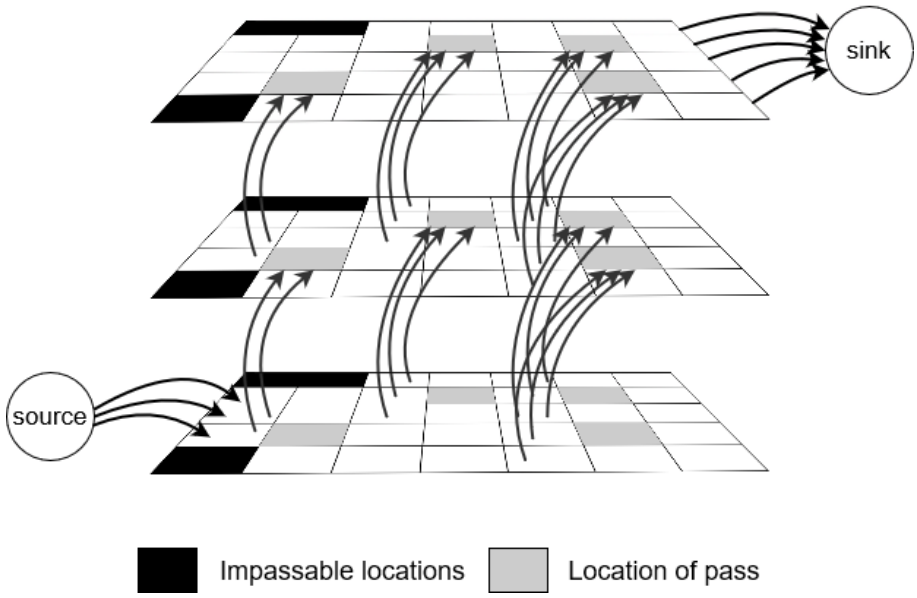


Figure 3.12: Transforming the 2D Map into a Multi-Level Graph

Input: A 2D array `map` with `r` rows and `c` columns, `n` the number of passes we have to travel through.

State Space: A 3D array `path` where `path[i,j,p]` is the minimum cost path from the start node 0 to grid cell (i, j) that goes through exactly p passes.

Base Case: `path[i,0,0]` = `map[i, 0]`, $\forall i \in r$

Inductive Case:

$$\texttt{path}[\texttt{i},\texttt{j},\texttt{p}] = \begin{cases} \min\{path[i-1,j-1,p], path[i,j-1,p], \\ \quad path[i+1,j-1,p]\} + map[i,j], \\ \qquad\qquad \text{if (i, j) is not a pass} \\ \min\{path[i-1,j-1,p-1], path[i,j,p-1], \\ \quad path[i+1,j+1,p-1]\} + map[i,j], \\ \qquad\qquad \text{otherwise} \end{cases}$$

Optimal Solution: $\min_{\forall i \in c} \{\texttt{path}[\texttt{i},\texttt{c},\texttt{n}]\}$

The total runtime for this algorithm is $O(rcn)$ since there are $O(rcn)$ subproblems to compute, and each one takes $O(1)$ time. If we were to build the level graphs, then this takes $O(rcn)$ space as well. But notice in the above dynamic programming formulation that each level only depends on the level before it. So storing only two levels at a time, we can reduce the space to $O(cn)$. Using divide and conquer techniques discussed in Chapter 2.5, we can also find the actual grid cells in an optimal solution with only $O(rcn)$ space.

Never Jump Down

We are given a rooted tree with `n` vertices where each vertex v_1 through v_n is labeled with a corresponding non-negative whole number u_1 through u_n. Vertex v_1 is the root. A jumping path is a sequence a_1, a_2, ..., a_k of length `k` where vertex v_{a_i} is an ancestor of v_{a_j} for all $i \leq j$.

Find the length of the longest non-decreasing jumping path, and the number of paths of that length. For more information on the problem, see [12].

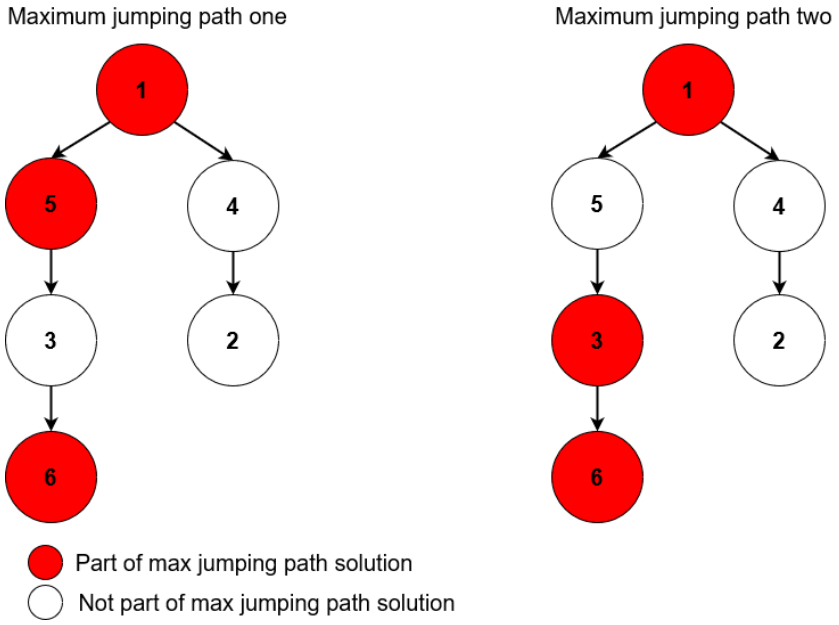


Figure 3.13: Example of a Rooted Tree with Two Possible Maximum Jumping Paths

The solution to this problem is the longest increasing subsequence in a tree, where each root to leaf path can be considered as a single dimensional array. We also need to compute the total number of such paths. We can compute the total number of paths by changing the max quantity into a sum in the inductive case of computing the LIS (see Chapter 2.4). We will leave this as an exercise to the reader, and in this section we will only focus on computing the length of a LIS along any root to leaf path.

As discussed in Chapter 2.4, we can use a splay tree to speed up computing the length of a LIS in an array in only $O(n \log n)$ time. We can use the exact same approach to this problem on a tree by doing a preorder traversal. At each node x, suppose we also pass in the splay tree T of all nodes on the path from the root of the tree to node x. Then using the same approach as the one discussed in Chapter 2.4, we insert x into T and splay it to the root. Its left tree contains all subsequences that can be extended from. Augmenting each node with the max LIS in the left subtree, we can compute the length of a LIS ending at node x. We then

pass the tree T to all of x's children, which are the recursive subproblems. After coming out of all recursive calls, we simply delete x from the splay tree and return. This ensures that the tree passed back to the parent does not contain any node from x's subtree.

We spend only $O(\log n)$ time per node. Therefore the entire algorithm takes only $O(n \log n)$ time. After augmenting each node with the length of a LIS to that node, we can traverse the entire tree to find the longest one.

Since this problem computes a LIS on a tree treating each root to leaf path as a one dimensional sequential problem, we have gained the following insight.

Insight #9:	Some dynamic programming problems on sequences can also be solved on trees that represent sequences along its paths with the same asymptotic complexity.

Watch Later

We are given a list `videos` of length `n` that contains `k` different types of videos to watch. We want to watch all videos of one type at a time before watching videos of a different type, and are allowed to watch any type of video first. To watch, we must click a video to play it. When a video finishes playing, it is automatically deleted from the list. If the video after it is of the same type as the previously watched video, then it will automatically start playing. Otherwise, we must click on the next video to start watching it. We would like to find the minimum number of clicks required to watch all videos in the list. For more information, please see [14].

For example, let `n` = 5, `k` = 3, and `list` = [a,b,b,c,a]. The optimal solution requires 3 clicks, and the steps for those clicks follow below.

1. Click on the second item of `list`, a video of type b. This video is watched and deleted from `list`. Since the next video is also of type b, it is automatically played and then deleted from `list`. `list` now contains [a,c,a].

2. Click on the second item of `list`, a video of type c. This video is watched and deleted from `list`. `list` now contains [a,a].

3. Click on the first item of `list`, a video of type a. This video is watched and deleted from `list`. Since the next video is also of type b, it is automatically played and then deleted from `list`. `list` is now empty.

Note that we are not given the order to watch the videos. Therefore, once we determine an order, we can compute the number of clicks to watch all the videos in the list. However, part of the challenge is to enumerate all subsets of video types quickly. We can have a dynamic programming formulation similar to the travelling salesman problem in Chapter 2.7. Let S be the subset of video types not yet watched. Given a list `list` of videos which contain only video types in S, we would like to know what is the minimum number of clicks to watch all videos.

At each step, we can choose any of the video types $i \in S$ to watch. If we choose to watch all videos of type i next, then we first find the number of clicks c to watch all videos of type i. Then we recurse into a subproblem to find the minimum number of clicks d to watch all videos where video types come from $S - \{i\}$. The total number of clicks to watch all videos in S is thus $c+d$. We minimize this quantity over all possible videos i we can watch first.

```
global : Array of clicks clicks, where clicks [S] is the minimum number of clicks
         to watch all video types in S
input  : Array list of videos, a set of unwatched video types S
output: The minimum number of clicks to watch all remaining video type S

1  Watch_Later(list, S):
2  if S = φ then
3  |   return 0
4  end
5  if clicks [S] already computed then
6  |   return clicks [S]
7  end
8  mincost = ∞
9  for i ∈ S do
10 |   c = number of clicks to watch video type i
11 |   d = Watch_Later (list, S − {i})
12 |   cost = c + d
13 |   mincost = min{mincost, cost }
14 end
15 clicks [S] = mincost return clicks [S]
```

Algorithm 15: Best Way to Watch Video Types in S

To solve the above problem, we have used Insight #2. We are doing an exhaustive search, but memoizing the results allows us to prevent computing the same subproblems over and over again.

The size of S is at most k. Therefore, in each step, we are iterating over $O(k)$ video types, and we take $O(n)$ time to determine the number of clicks. With one linear scan we can compute the number of clicks to watch all videos in S. There are $O(2^k)$ subproblems, and in each step we spend only $O(n+k)$ time. So the total time is $O(2^k(n+k))$. Since the number of video types can be upper bounded by n, we have a runtime of $O(2^k n)$.

Chapter 4

Problem Classifications and Website

The dynamic programming insights in this thesis were developed after solving over seventy contest problems from the 2019 ICPC North American regional programming contests. In doing so, we have also classified problems based on the main algorithmic techniques used. This chapter describes how these problems were classified, the different classification categories, and a table of all classified ICPC problems. We have implemented a website that allows us to find problems based on their classification easily. We describe the many features of this website in this chapter.

4.1 Classification

The ICPC problems were classified based on the data structure, algorithm, or technique that we used to solve the problem within the problem's specified time limit. Each problem can have one or more classifications, depending on what is necessary to solve the problem. For example, if the problem required simply trying all possible solutions, and there were no tricks to speed up the solution, then the problem would only be classified as "Brute Force." If a problem involved using a sweep line to insert items into a BST in a specific order, then the problem would have two classifications, "Sweep Line" and "Binary Search Tree."

Multiple, more specific classifications were added to the problem if that would prove useful for searching for problem types. For example, problems classified as "Connected Components" and "Shortest Path" would both have the additional classification of "Graph Algorithm."

This makes searching for problem types easier; those who search for the more specific connected components or shortest path problems will not get any of the other specific classifications returned, but those looking to practice all kinds of graph problems will get every possible graph problem returned from their search query.

All problem classifications are listed below. The meanings of these categories are well understood by competitive programmers and students who have taken an intermediate algorithms course at their respective universities.

1. Ad-hoc
2. Algebra
3. All-Pairs Shortest Path
4. Angular Sorting
5. Binary Search
6. Binary Search Tree
7. Bipartite Graph
8. Brute Force
9. Closest Pair
10. Combinatorics
11. Computational Geometry
12. Connected Components
13. Context-Free Grammar
14. Counting
15. Depth First Search
16. Direct Access Table
17. Directed Acyclic Graph
18. Dynamic Programming
19. Dynamic Programming in Trees
20. Geometry
21. Graph Algorithm
22. Graph Traversal
23. Greedy
24. Hash Table
25. Hashing
26. Linear Programming
27. Linear Search
28. Linear System of Equations
29. Markov Chains
30. Math
31. Number System
32. Number Theory
33. Permutation
34. Polynomial Hashing
35. Priority Queue
36. Probability
37. Projection
38. Queue
39. Recursion
40. Shortest Path
41. Simulation
42. Sliding Window
43. Smallest Enclosing Circle
44. Sorting
45. Steady State
46. String
47. Suffix Tree
48. Sweep Line
49. travelling Salesman
50. Tree Algorithm

The tables below show that there were 131 problems overall in the 2019 North American regional competitions. However, we note that some of these problems are duplicates with the same name, and some of these problems are the same underlying problem with a different name. In total, there were 86 unique problems in the 2019 North American regional competitions. While we surveyed all 86 problems, some of these problems were so challenging that while we

could tell what techniques would be necessary to solve them, we could not finish solving them in time for this thesis. This is why there are more problems classified as dynamic programming than we explored in Chapter 3. Only 5 problems were not classified as we found them to be too challenging. This is shown in the tables below by blank classification cells. In the end, we classified over 80 problems, and solved over 70 problems over the course of this thesis. As future work, we hope to solve and classify all 86 unique problems.

The list of all the 2019 North American Regional ICPC problems that were classified during this thesis, and their classifications, are shown in the tables below. For the purposes of presentation, we divide these tables based on regional contests.

Table 1

Pacific Northwest Region		
1	Radio Prize	All-Pairs Shortest Path
2	Perfect Flush	Hash Tbale, Suffix Tree
3	Coloring Contention	Graph Algorithm, Shortest Path
4	Dividing by Two	Simulation
5	Rainbow Strings	Combinatorics
6	Carny Magician	
7	Glow, Little Pixel, Glow	Sweep Lines, Binary Search Trees
8	Pivoting Points	Angular Sorting
9	Error Correction	Hash Table
10	Interstellar Travel	Simulation
11	Computer Cache	Simulation
12	Carry Cam Failure	Algebra, Brute Force
13	Maze Connect	Graph Algorithm, Connected Components

Table 2

Mid Atlantic Region		
1	Kafkaesque	Ad-hoc
2	The Deal of the Day	Recursion, Combinatorics
3	This Ain't Your Grandpa's Checkerboard	Simulation
4	Pixelated	Binary Search Tree, Sweep Line
5	Never Jump Down	Dynamic Programming in Trees
6	Weird Flecks, But OK	Computational Geometry, Smallest Enclosing Circle
7	On Average They're Purple	Graph Algorithm, Shortest Path
8	Don't Fence Me In	Graph Algorithm, Connected Components

Table 3

Rocky Mountain Region		
1	Piece of Cake	Ad-hoc
2	Fantasy Draft	Simulation, Queue
3	Folding a Cube	Brute Force, Simulation
4	Integer Division	Combinatorics, Hash Table, Sorting
5	Hogwarts	Graph Algorithm, Graph Traversal
6	Molecules	Ad-hoc, Simulation
7	Typo	Hashing
8	The Biggest Triangle	Brute Force, Geometry
9	Tired Terry	String, Sliding Window
10	Watch Later	Dynamic Programming, Combinatorics
11	Lost Lineup	Ad-hoc, Sorting, Permutation

Table 4

Southern California Region		
1	Lychrel Search	Simulation
2	Morse Redux	Ad-hoc, Context-Free Grammar
3	Stairs	Linear Programming
4	Computer Cache	Simulation
5	Swap Madness	Hash Table
6	Maze Connect	Graph Algorithm, Connected Components
7	Flaw Removal	Computational Geometry Smallest Enclosing Circle
8	Permits in Kafkatown	Ad-hoc
9	Visibility	Linear Search, Computational Geometry
10	Pixel Activation	Binary Search Tree, Sweep Line

Table 5

North Central Region		
1	Weird Flecks, But OK	Computational Geometry, Smallest Enclosing Circle
2	Code Names	Hash Table
3	New Maths	Algebra, Brute Force
4	Some Sum	Counting, Combinatorics
5	Early Orders	Suffix Tree, Hash Table
6	Pulling Their Weight	Tree Algorithm
7	Birthday Paradox	Probability, Math
8	On Average They're Purple	Graph Algorithm, Shortest Path
9	Full-Depth Morning Show	All-Pairs Shortest Path, Graph Algorithm
10	This Ain't Your Grandpa's Checkerboard	Simulation
11	Solar Energy	Simulation

Table 6

South Central Region		
1	All is Well	Graph Algorithm, Shortest Path
2	Birthday Paradox	Probability, Math
3	Code Names	Hash Table
4	Flipping Patties	Ad-hoc
5	Full Depth Morning Show	All-Pairs Shortest Path, Graph Algorithm
6	Don't Fence Me In	Graph Algorithm, Connected Components
7	On Average They're Purple	Graph Algorithm, Shortest Path
8	Pulling Their Weight	Tree Algorithm
9	Some Sum	Combinatorics, Counting
10	This Ain't Your Grandpa's Checkerboard	Simulation
11	Never Jump Down	Dynamic Programming in Trees
12	Weird Flecks, But OK	Computational Geometry (smallest enclosing circle)

Table 7

East Central Region		
1	Retribution	Closest Pair, Sweep Lines, Greedy
2	Bio Trip	Shortest Path
3	Cheese, If You Please	Dynamic Programming, Linear Programming
4	Follow the Bouncing Ball	Computational Geometry
5	Just Passing Through	Dynamic Programming, Directed Acyclic Graph
6	Musical Chairs	Simulation, Binary Search Tree
7	Out of Sorts	Simulation
8	Remainder Reminders	Algebra
9	Square Rooms	Recursion
10	Taxed Editor	
11	Where Have You Bin?	

Table 8

Northeastern Region		
1	Add 'Em Up	Hash Table
2	The Colonization of El-gă-rizm	Bipartite Graph
3	Cutting the Necklace	Sliding Window, Ad-hoc
4	Fair Divisions	Angular Sorting, Brute Force, Hash Table
5	Guess the Digits	Linear System of Equations
6	Monument Maker	Ad-hoc, Simulation
7	Protecting the Collection	Simulation
8	The Sock Pile	Probability
9	Stickers	Dynamic Programming
10	The Wire Ghost	Sorting, Brute Force, Simulation

Table 9

Mid Central Region		
1	Basketball One-on-One	Simulation
2	Commemorative Race	Graph Traversal, Depth First Search
3	Convoy	Priority Queue, Greedy
4	Crazed Boar	Angular Sorting, Projection
5	Dance Circle	Linear Programming
6	Dragon Ball I	travelling Salesman
7	Dragon Ball II	Dynamic Programming, travelling Salesman
8	Farming Mars	
9	Soft Passwords	Ad-hoc
10	Sum and Product	Brute Force
11	True/False Worksheet	
12	Umm Code	Simulation, Ad-hoc

Table 10

New York Region		
1	FYI	Ad-hoc
2	Remdoku	Recursion
3	Pass The Buck	Markov Chains, Steady State
4	Gerrymandering Criterion	Simulation, Geometry
5	Circle Garden	Geometry
6	Origami Fold	Geometry, Binary Search
7	Simple Collatz Sequence	Simulation
8	Problematic Public Keys	Number Theory
9	Integers in Rational Bases	Number System
10	How Many Unicycles in a Wheel?	Combinatorics
11	Regional Team Names	Ad-hoc

Table 11

Southeast Division 1 Region		
1	Carryless Square Root	Algebra, Brute Force
2	Computer Cache	Simulation
3	Elven Efficiency	Direct Access Table
4	Swap Free	Hash Table
5	Fixed Point Permutation	
6	Interstellar Travel	Simulation
7	Jumping Path	Dynamic Programming in Trees
8	Levenshtein Distance	Sorting, Binary Search Tree
9	Maze Connect	Graph Algorithm, Connected Components
10	One of Each	Suffix Tree, Hash Table
11	Windmill Pivot	Angular Sorting

Table 12

Southeast Division 2 Region		
1	Balanced Animals	Tree Algorithm
2	Carryless Square Root	Algebra, Brute Force
3	Checkerboard	Simulation
4	From A to B	Simulation
5	Elven Efficiency	Direct Access Table
6	Even or Odd	Combinatorics
7	Glow, Pixel, Glow!	Binary Search Tree, Sweep Line
8	Jumping Path	Dynamic Programming in Trees
9	Levenshtein Distance	Binary Search Tree, Sorting
10	Rainbow Strings	Combinatorics
11	ReMorse	Ad-hoc, Context-Free Grammar

4.2 Website

In the past, the ICPC Live Archive [7] acted as an archive for all past regional and global ICPC competition problems. However, since 2017, due to recent administrative changes, this archive has been discontinued. The main motivation for the creation of our website was to create a new archive of ICPC competition problems starting from the 2018 regional contest. Since classification of contest problems based on techniques are rarely found, we provide a search feature that allows to search for problems based on multiple factors, which are

- Problem Name
- Classification (algorithmic technique used)
- Regional Competition
- Year of Contest

The website's base was created in HTML, starting with a free template from Bryant Smith [18]. The template was adjusted to create a simple two-page interface with a 'home page' that describes the purpose of the site, and a 'search problems' page that allows visitors to the website to search through the database of problems by name, classification, regional competition, or competition year.

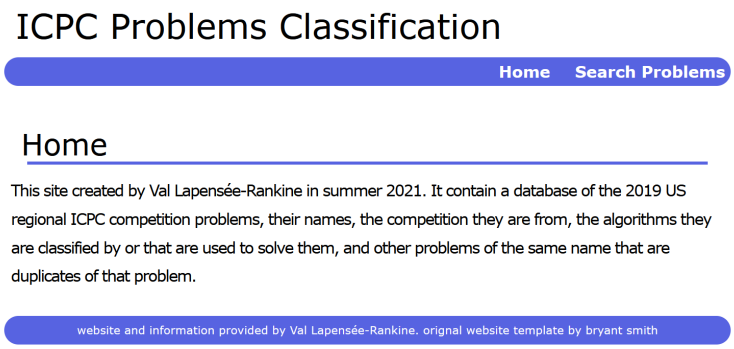
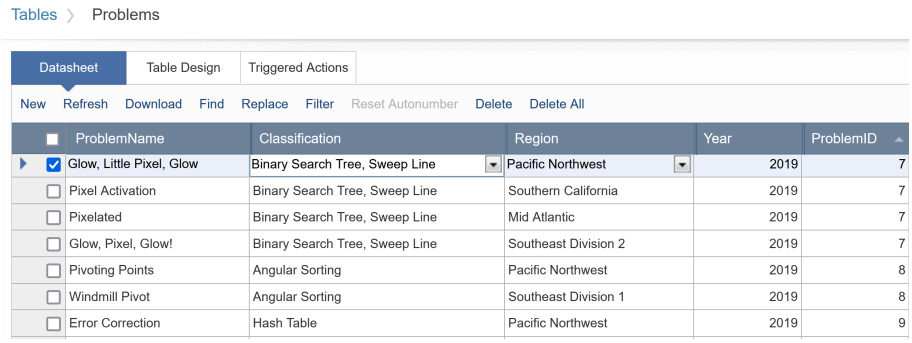


Figure 4.1: Home Page of Thesis Website

The underlying database used is a free database hosted by the site Caspio [2]. Each entry in the database has six fields; "ProblemName," "Classification," "Region," "Year," "ProblemID," and "Link." The "ProblemName" field is text data type, which is required to be unique – this field is the unique identifier for each problem. "Classification" is a list of strings, and allows a problem to have zero or more classifications from the classification types listed in Section 4.1. The "Region" field is also a list of strings, and allows a problem to have zero or more regions from those listed in Section 1.1, as problems can appear in multiple regions. The "Year" field is a simple number that contains the competition year that the problem appeared in. The field "ProblemID" is an integer that each problem is assigned. When classifying, we found that some problems, despite having different names, were the same problem. If the problem is unique, it is a given a new, unique problem ID. If the problem is the same as one previously inserted with a different name, it is given the same problem ID as the previous problem. For example, the problems "On Average They're Purple" and "Coloring Contention" are the same underlying problem, so they have the same problem ID. The final field, "Link," is a text field that holds the URL of the description of each problem, so that users can click on the click and read the problem for themselves.



ProblemName	Classification	Region	Year	ProblemID
Glow, Little Pixel, Glow	Binary Search Tree, Sweep Line	Pacific Northwest	2019	7
Pixel Activation	Binary Search Tree, Sweep Line	Southern California	2019	7
Pixelated	Binary Search Tree, Sweep Line	Mid Atlantic	2019	7
Glow, Pixel, Glow!	Binary Search Tree, Sweep Line	Southeast Division 2	2019	7
Pivoting Points	Angular Sorting	Pacific Northwest	2019	8
Windmill Pivot	Angular Sorting	Southeast Division 1	2019	8
Error Correction	Hash Table	Pacific Northwest	2019	9

Figure 4.2: Problem Database Inside Caspio

To add a new problem to the database, we simply scroll to the bottom of the entries, and click in the last, empty spot. From there, we can enter the problem's name, select all of its classifications, select its region, enter the year, and enter its problem ID.

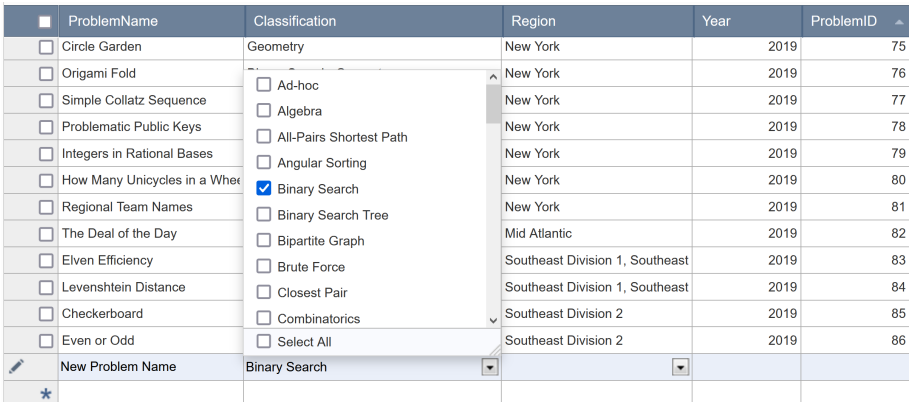


Figure 4.3: Adding a New Problem to the Database

This website is hosted at `https://cs.appstate.edu/~mohanr/ICPC_Problem_Site`. We invite competitive programmers and students of algorithms to visit this website, and hope that the search feature is useful in not only practicing for ICPC contests, but also for improving algorithmic problem solving and programming skills.

Chapter 5

Conclusion

5.1 Final Results

In this thesis, we surveyed and solved over 70 problems from the premier programming contest for university students – the Intercollegiate Programming Contests (ICPC). While surveying, we also classified these problems based on the data structure or algorithmic technique used to solve them, as explored in Section 4.1. These problems were fun and intellectually stimulating, and by solving them we have gained many insights into the technique of dynamic programming. These insights were explained in detail and illustrated with example problems throughout Chapter 2 and Chapter 3. The nine insights gained and explored in this thesis are outlined below.

1. Dynamic Programming = Recursion + Memoization.
2. Dynamic Programming = Recursive Exhaustive Search + Memoization.
3. The runtime of dynamic programming algorithm = total number of sub-problems $\times$ total time for each sub-problem.
4. Dynamic Programming = Incremental Construction + Memoization.
5. The actual set of elements belonging to an optimal solution can be obtained using back-pointers.
6. Binary search trees can help in speeding up determining a maximization (minimization) over a set of solutions.

7. Divide and Conquer can help find the items in an optimal solution with reduced space.
8. Dynamic Programming = Shortest Path in a DAG.
9. Some dynamic programming problems on sequences can also be solved on trees that represent sequences along its paths with the same asymptotic complexity.

We hope that such insights help in identifying problems with a dynamic programming solution, and in using them to formulate dynamic programming solutions effectively.

We also built a web-based platform to archive problems after the 2017 ICPC contests, described in Section 4.2. We have successfully archived the problems from the 2019 North American regional contests, totaling over 70 problems. The web application allows for easy addition of new problems, and has a search feature to search for problems by problem name, regional competition, problem classification, and problem year. The hope is that this site will be filled with all problems from all contests, both regional and global, from the ICPC contests. Moreover, we hope to fill this site with problems from the biggest contest for high school students – the USA Computing Olympiad (USACO). Future contestants may find this web application useful as they can find problems based on a particular algorithmic technique, thereby helping them practice applying the technique.

5.2 Future Work

This thesis leads to potential for many different kinds of future work. One avenue that has been briefly touched on in the previous section, is to continue filling the database with ICPC and USACO problems.

While we focused on the 12 North American regions to keep the scope of this thesis reasonable, there are 46 ICPC regions worldwide, listed below.

1. Africa/Middle East - Arab Contest
2. Africa/Middle East - South Africa
3. Asia - Amritapuri
4. Asia - Beijing
5. Asia - Chennai
6. Asia - Daejeon
7. Asia - Dhaka
8. Asia - East Continent League Finals
9. Asia - Gwalior Site
10. Asia - Gwalior Site
11. Asia - Ho-Chi-Minh City Site

12. Asia - Hong Kong
13. Asia - Hua-Lien
14. Asia - Jakarta
15. Asia - Kabul
16. Asia - Kharagpur
17. Asia - Kolkata-Kanpur Site
18. Asia - Lahore
19. Asia - Manila
20. Asia - Nanning
21. Asia - Nakhon Pathom Site
22. Asia - QingDao
23. Asia - Shenyang
24. Asia - Tehran
25. Asia - Tsukuba
26. Asai - Urumqi
27. Asia - Xian
28. Asia - Yangon
29. Europe - Central
30. Europe - Northeastern-Eurasia
31. Europe - Northwestern
32. Europe - Southeastern
33. Europe - Southwestern
34. Latin America
35. North America - East Central NA
36. North America - Greater NY
37. North America - Mid-Atlantic USA
38. North America - Mid-Central USA
39. North America - North Central NA
40. North America - Northeast NA
41. North America - Pacific Northwest
42. North America - Rocky Mountain
43. North America - South Central USA
44. North America - Southeast USA
45. North America - Southern California
46. South Pacific

Note that the ICPC competitions are annual competitions for university students. The first ICPC contest was held in 1977 [4]. Moreover, the ICPC World Finals, treated as the olympics of programming competitions, are conducted to find the undisputed winners of all geographic regions every year. So a complete archive of all problems from the ICPC contest alone would lead to thousands of problems. In addition, the USACO and the IOI contests for high school students are also annual competitions, and adding these problems regularly to this site makes for a lifelong task.

Some of the contest problems are too challenging that beginner students may be overwhelmed with the scope and difficulty of the problems. Therefore, providing editorials for contest problems would go a long way towards mastery in algorithmic problem solving. We call out to the competitive programming community to provide more detailed and accessible expositions of some of the best problems from these contests.

Another avenue of possible future work is to add more information about each problem to the database to make our site more useful. One option is to add a perceived difficulty level to

each problem, so that contestants can filter problems based on their skill level instead of wasting time on problems either too simple or too difficult for them to solve. Another option is a star rating to determine the quality of a problem. Here, quality does not mean difficulty, rather it is how good the problem is presented, and how well algorithmic techniques can be applied to solve the problems. For example, a problem with a trivial solution, or one that involves difficult parsing of input and output will be deemed as low quality, while a problem that uses a clever insight, or uses successive layers of abstraction or the use of multiple techniques will be deemed as high quality.

The final option for future work is an in-depth research into other algorithm types, in order to find insights that would allow easier identification and formulation of solutions with algorithms other than dynamic programming. Many of the problem classifications listed in Section 4.1 are viable candidates for an in-depth exploration, but based on the type of problems we found in ICPC competitions during this thesis, we recommend exploring graph algorithms or string algorithms to be the most useful for contestants.

Bibliography

[1] Aaron Bloomfield and Borja Sotomayor. A Programming Contest Strategy Guide. *SIGCSE '16: Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 609–618, February 2016.

[2] Caspio. Build online database apps - low code platform — Caspio. `https://www.caspio.com/`. [Online; Last accessed on: 01-Nov-2021].

[3] M. Held and R. M. Karp. A Dynamic Programming Approach to Sequencing Problems. *J. Soc. Indust. Appl. Math.*, 10:196–210, 1962.

[4] ICPC. About ICPC. `https://icpc.baylor.edu/regionals/abouticpc`. [Online; Last accessed on: 01-Nov-2021].

[5] ICPC. Icpc Regional Rules for 2019 Regionals. `https://icpc.baylor.edu/regionals/rules`. [Online; Last accessed on: 01-Nov-2021].

[6] ICPC. Region Finder. `https://icpc.baylor.edu/regionals/finder`. [Online; Last Accessed on: 01-Nov-2021].

[7] ICPC Live Archive. Welcome to the ICPC Live Archive. `https://icpcarchive.ecs.baylor.edu/`. [Online; Last accessed on: 01-Nov-2021].

[8] IOI. Contests. `https://ioinformatics.org/page/contests/10`. [Online; Last accessed on: 01-Nov-2021].

[9] IOI. Regulations. `https://ioinformatics.org/page/regulations/9`. [Online; Last accessed on: 01-Nov-2021].

[10] IOI 2020. Competition Rules. `https://ioi2020.sg/rules`. [Online; Last accessed on: 01-Nov-2021].

[11] Kattis. Just Passing Through. `https://ecna19.kattis.com/problems/justpassingthrough`. [Online; Last accessed on: 01-Nov-2021].

[12] Kattis. Never Jump Down. `https://mausa19.kattis.com/problems/neverjumpdown`. [Online; Last accessed on: 01-Nov-2021].

[13] Kattis. Stickers. `https://nena19.kattis.com/problems`. [Online; Last accessed on: 01-Nov-2021].

[14] Kattis. Watch Later. `https://rmc19.kattis.com/problems/watchlater`. [Online; Last accessed on: 01-Nov-2021].

[15] Samantha Valentine Lapensée-Rankine. ICPC Problems Classification Site. https://cs.appstate.edu/~mohanr/ICPC_Problem_Site/. [Online; Last accessed on: 01-Nov-2021].

[16] Kevin Wayne Robert Sedgewick. *Algorithms*. Addison-Wesley Professional, 4 edition, 2011.

[17] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):562–686, July 1985.

[18] Bryant Smith. Bryant Smith Design Group. http://www.bryantsmith.com/. [Online; Last accessed on: 01-Nov-2021].

[19] USACO. Contests. http://usaco.org/index.php?page=contests. [Online; Last accessed on: 01-Nov-2021].

[20] USACO. History. http://usaco.org/index.php?page=history. [Online; Last accessed on: 01-Nov-2021].

[21] USACO. Overview. http://usaco.org/. [Online; Last accessed on: 01-Nov-2021].

[22] USACO. Platinum. https://usaco.guide/plat. [Online; Last accessed on: 01-Nov-2021].

Vita

Samantha Valentine Lapensé-Rankine was born in Ontario, Canada to Tina Lapensé and James Rankine. She graduated from Mooresville High School in June 2014. She entered Appalachian State University, Boone, in the fall of 2015 and received a Bachelor of Science in Computer Science with a minor in Mathematics in May 2019. The following fall, she began study towards a Master of Science degree with a concentration in Theoretics, also at Appalachian State University. While pursuing her Master's degree, Ms. Lapensé-Rankine worked as a graduate teaching Assistant, and in 2021 was awarded the Outstanding Graduate Student Teaching Award.

Currently, Ms. Lapensé-Rankine is an Adjunct Lecturer in the Computer Science Department at Appalachian State University. She lives in Mooresville, North Carolina.