TJ IOI 2017 Study Guide

TJ IOI Officers

Thomas Jefferson High School for Science and Technology

Saturday, May 13, 2017

Contents

Preface

The Thomas Jefferson Intermediate Olympiad in Informatics (TJ IOI) is a high-school computer science competition, to be held on Saturday, May 13, 2017, at TJHSST. We hope to be able to share our passion for computer science with our peers, in hopes of inspiring our fellow students through exposure to concepts not usually taught in the classroom. In order to help our participants gain a bit of background before the event, we have created a study guide to help ensure that students are equipped with the tools they need to be successful. If any mistakes are found in this study guide, or if you simply have questions, do not hesitate to contact us at tjioiofficers@gmail.com.

We'd like to thank Mr. Thomas Rudwick, our faculty sponsor, for his tireless efforts in helping us plan the event, along with the countless times we've used his room for lunch meetings. We'd also like to thank Mrs. Nicole Kim for her guidance and wisdom from having sponsored TJ IOI in the past, as well as Mr. Stephen Rose for making sure our planning was airtight. Finally, we'd like to thank Dr. Evan Glazer, our principal, and the entire TJ Computer Science Department, for creating a wonderful environment where we can foster our passion for Computer Science.

We hope that you find this study guide useful, and we look forward to seeing you at TJ IOI 2017!

The TJ IOI Officers

Chapter 1

Fundamentals

1.1 Algorithms

In the field of computer science, programmers use **algorithms** to solve complicated problems. What is an algorithm? An algorithm is a procedure, or series of problem-solving operations, in order to accomplish a certain task. Algorithms can range from simple operations, such as finding the smallest value in an array, to incredibly complex jobs, such as Facebook's facial recognition or Google's PageRank.

Algorithm 1 Finding the Maximum Value in an Array

```
$M \leftarrow 0$
for all $a_i$ in $A$ do                    ▷ iterate through elements of $A$
    if $a_i > M$ then
        $M \leftarrow a_i$
```

For example, here is an algorithm to find the largest value in an array. We assign a value of 0 to M, and then **iterate** over the contents of the array. Iteration is a computer science term, meaning we examine each element of the array one at a time. While we iterate over the array, we check each value against our existing maximum value. If the value is greater than the largest value we've already found, then we will update our maximum to reflect that value. Once we have iterated over all elements in the array, we are guaranteed to have found the largest value in the array.

Note that this algorithm was not written in a formal programming language above, nor was it described in a programming language. This is because an algorithm is an idea, not lines of code. It is important to remember that before one writes code to accomplish a task, one first thinks through the problem, and comes up with a solution in words rather than code. The language that one uses is merely the medium of communication to the computer: just as one can express the same ideas in English and French, an algorithm should be able to be implemented in Java, C/C++, Python, or any language.

1.2 Computational Complexity

Remember when we talked about how algorithms can range from simple to complex? How can we tell the difference between a simple algorithm and a complicated one? The answer lies in **Computational**

Complexity. Computational Complexity is a measure of how efficiently a program will run. Although computers appear to be instant, all operations take time. Computers are limited to a finite, yet extremely large, number of operations per second. In most modern computers, approximately 10^8 operations can be done per second. This seems like a huge number, but in practice, we find that we quickly use it up, especially when dealing with large amounts of data. This is why it is incredibly important to write **efficient** algorithms - a poorly written algorithm that solves a task in three days of computation time could be completed in under a second if written efficiently! Thus far, we have only developed a qualitative idea of efficiency, but computer scientists want rigor, and thus use **Big-O notation** in order to quantify the efficiency of their algorithms.

1.2.1 Big-O Notation

Big-O notation is the computer scientists way of describing the efficiency of an algorithm. Think back to our first algorithm (finding the maximum in a list). We examine each element of the array, one at a time. Thus, if our array is of length n, we will need n operations to complete our algorithm. This is true regardless of the value of n, so the operation is said to be $O(n)$. But what about the initial assignment? Shouldn't the algorithm be $O(n+1)$? In computer science, we care about the efficiency when the data sets are **large**, and as n becomes large, the number of operations will grow at the same rate that n grows, and the 1 will become negligible. We also only care about the **order** of the growth, that is, $O(cn)$ is the same thing as $O(n)$, where c is any constant factor.

An algorithm grows only as fast as its slowest operation. For example, a nested for-loop will be $O(n^2)$, and so another for-loop afterwards with $O(n)$ efficiency will be negligible compared to the nested loop. Note that an algorithm with $O(n^2)$ efficiency is not guaranteed to run slower than one of $O(n)$, as the faster operations may have constant terms associated with them that are very large. However, we know for certain that for sufficiently large data set size, the $O(n^2)$ algorithm will be slower.

Other common runtimes include $O(\log n)$ or $O(n \log n)$. A good question to ask may be, "what is the base of the logarithm in question?" The answer is that it does not matter - the difference between two logarithms in different bases is merely a constant factor, which we disregard in our complexity analysis.

In a more theoretical sense, one can determine the Big-O efficiency of an algorithm mathematically. Given an algorithm, we can write some function, which takes the size of the data as input, and outputs the number of operations. For example, consider an algorithm which loops over an array twice, then, for **each** element of the array, loops over the **entire** array three times, then loops over the first three terms of the array. Then, assuming the length of our array is given by n (where $n > 3$), our function is given by $f(n) = 2n + 3n^2 + 3$. We now define the Big-O efficiency of the function, $O(g(n))$, to be a function satisfying:

$$\frac{|f(n)|}{|g(n)|} = M$$

where M is a finite, positive, non-zero value. In other words, $f(n)$ grows on the order of $g(n)$. Choosing $g(n) = n^2$ will result in a value of 3 for M, and so the function $f(n)$ and the associated algorithm is said to be of order $O(n^2)$. Note that this choice of $g(n)$ is not unique, but is the "simplest" choice. This process is similar to a leading term test, often used when taking the limit of a rational function in BC Calculus, in which the term that grows fastest "dominates" the rest as the independent variable becomes large.

As an example, consider the following algorithm:

Algorithm 2 Finding Pairs

```
$X \leftarrow 0$
for all $a_i$ in $A$ do
    for all $a_j$ in $A$ and $j > i$ do
        if $a_i = a_j$ then
            $X \leftarrow X + 1$
```

What is the efficiency of the algorithm? We can see that there is one loop that iterates over the entire array A, and another loop that iterates over all the elements after the first element chosen. If both loops were over the entire array, then it would clearly be $O(n^2)$, but in this case, the inner loop does not iterate over the entire array each time! From this analysis, we can see that the efficiency of the program seems to be faster, as looping over the entire array in both loops means that we compare every pair twice, as for any i and j, we will check $i = j$ and $j = i$, but looping the way we do in this algorithm ensures we check each pair only once. This means that our program only needs half the number of operations as an $O(n^2)$ algorithm! Unfortunately, as we have previously discussed, when using Big-O analysis, we discard constant factors and only care about the order of the growth. Despite only needing $\frac{1}{2}n^2$ operations, the number of operations still grows as n^2. Thus, this algorithm is also $O(n^2)$.

1.2.2 Program Speed

One of the largest differences between different programming languages is that some are faster than others. For example, compiled languages like C and C++ are significantly faster than languages like Python. Usually, contests will allow for different time limits based on the language used, but it is often best to choose a fast language as the increase in speed will more than compensate the difference in time allotted.

Another thing to keep in mind is that most problems have an intended solution, and the size of the test cases will reflect the efficiency of the intended solution. The following are standard guidelines for time complexity (assuming 1 second of runtime):

- $n \le 10$: $O(n!)$
- $n \le 25$: $O(2^n)$
- $n \le 50$: $O(n^4)$
- $n \le 500$: $O(n^3)$
- $n \le 5000$: $O(n^2)$
- $n \le 100000$: $O(n \log n)$
- $n \le 1000000$: $O(n)$

When given a problem, by looking at the size of the test cases, we can infer information about what kind of algorithm to use. For example, consider the following problem:

Problem. Devon has N cookies ($1 \le N \le 6000$), each with between 0 and M chocolate chips ($1 \le M \le 1000000$). Devon would like to put these cookies in order, from most to least chocolate chips. Please help Devon do so!

Without solving the problem, we can guess that the intended solution will likely be $O(n^2)$, which will give us some information regarding how we should approach this problem. We know that this is the case as the number of cookies is bounded by 6000, which is approximately the limit for an $O(n^2)$ solution. Note that the number of chocolate chips does not matter, as we are only using that number as a tool for comparison. Because we have bounded the solution to be $O(n^2)$, we know that we are able to afford a nested for-loop to solve our problem (a runtime of $O(n^2)$ is generally an indicator of a nested for-loop, as we run n operations n times each, for a total of n^2).

What if the number of cookies was bounded by $1 \leq N \leq 100000$? In that case, we must find a faster solution - an $O(n^2)$ solution will no longer be sufficient! In this case, we will need an $O(n \log n)$ solution. A logarithm in the runtime usually denotes a recursive or binary search solution, topics that we will discuss later in this guide. The correct solution for this problem is to use a sort, and there are many different ways to accomplish this task. We will discuss sorts more expansively, as well as the runtimes, advantages, and disadvantages of each sort, in a later section.

Chapter 2

Data Structures

What exactly are data structures? Why would you use them? As the name implies, data structures are ways of handling information and values in an efficient manner in order to improve computational speed. Typically, they are used to store lots of values that are somehow related instead of having to create a normal variable for each one. Good knowledge of data structures is essential to being able to solve a problem in the fastest amount of time possible.

In most cases, you won't ever have to write a data structure from scratch unless you are building a custom one tailored to fit a problem or are doing it to better learn the theory. Almost all languages will have have very efficiently written implementations in their libraries and all you have to do is to know which one works the best for your situation.

2.1 Arrays

Arrays are the simplest of data structures. They are a bunch of objects put in some order and stored in the memory. This is a big advantage over normal variables because if you wanted to store something such as the alphabet, one array could contain all of this information efficiently instead versus 26 different variables. Arrays are best for querying information. For example, if I wanted to know what the 14th letter was, I can do this in $O(1)$ time. However, if I want to add or remove something, arrays become very cumbersome as they have a fixed length. This means I have to create a second array and copy all of the information into this second one, adding or deleting any information I would like to change.

2.2 ArrayList or Dynamic Array

As mentioned above, arrays arrive at problems when information wants to be added or removed in an efficient manner. ArrayLists, otherwise known as Dynamic Arrays, attempt to solve this by sacrificing memory for computational speed. When an ArrayList is created, it will store more space than is needed. So for example if I wanted to store the names of all of my friends and had 13 of them, when I create an ArrayList while it will appear to me as if there are only 13 slots, the library running in the backend will actually create 16 slots and leave the extra ones as null (the smallest power of 2 greater than the length I ask for). Now, when I want to add a new friend I have made, the ArrayList will increment its displayed size and turn one of the next null slots into a person. It can conversely do the opposite for removing a friend. In this way, ArrayLists allow for $O(1)$ addition and subtraction for most cases, only occasionally being $O(n)$ when the length has to be expanded. This would happen if I were to go beyond 16 friends in the previous example, in which case a new array of length 32 would be created. This is limited however to adding or removing from the end, as otherwise all the elements have to shifted to fill gaps, therefore being $O(n)$.

2.3 Linked Lists

A list is a set of items that have some way of sorting themselves linearly. For example, if I was creating a list of numbers, they would be sorted by size yielding something such as 0, 1, 2, etc. In computer science, each item in a list is called a node. Each node has its value and a pointer, which references its neighbors. A good way to visualize this is a bunch of people in a line. Each person in the line knows what their name is and can also ask either of their neighbors for their name, but cannot directly ask anyone else.

So why do people use linked lists? Well ArrayLists are very effective in removing things from the end, but if items start being removed from the beginning then the data structure quickly becomes very messy. In addition, removing from the middle requires a lot of shifting as mentioned above. However, linked lists can add and remove in $O(1)$ time as a node simply has to change its pointer. For example, in the case of the line of people where person A is in front of B who is in front of C, if person B was to leave, then person C would simply treat the person in person A as the person in front of them instead of person B. However, the drawbacks of linked lists are that they take $O(n)$ time in retrieving data as each node must be iterated through before the target can be found.

As a side note, there are actually two types of linked lists. The more common one is a doubly linked list where a node will point to what is in front of it and behind it. On rare occasions, a singly linked list will be used where a node only knows what is in front of it in order to save memory.

2.4 Stacks

Stacks are an alternative method to storing data such that insertion and deletion is $O(1)$ but access is $O(n)$. A stack utilizes LIFO (last in, first out). Basically, it can be treated as a stack of books. If I was to put book A down, and then book B down on top of it, and so on all the way up to book Z, I would have a stack of books. Now if I wanted to take off a book, I would first have to remove book Z to look at anything below it. In this way, it becomes very cumbersome to retrieve previous items. However, stacks can be very useful in cases of where you only want to look at the last location. One example of this is in card games such as Pokemon or Magic: The Gathering. In these games, the last card to be played comes into play first, so that cards can be played to block other things. In such a case, all the cards are added to a stack time-sorted, and then resolved starting from the top of the stack. So if you were to play a card that said “You win the game!” and your opponent played a card that said “Ignore another card’s effect” in response to what you played, your opponent’s card would block your card and take effect first, preventing you from winning as the cards would be added to a stack.

2.5 Queues

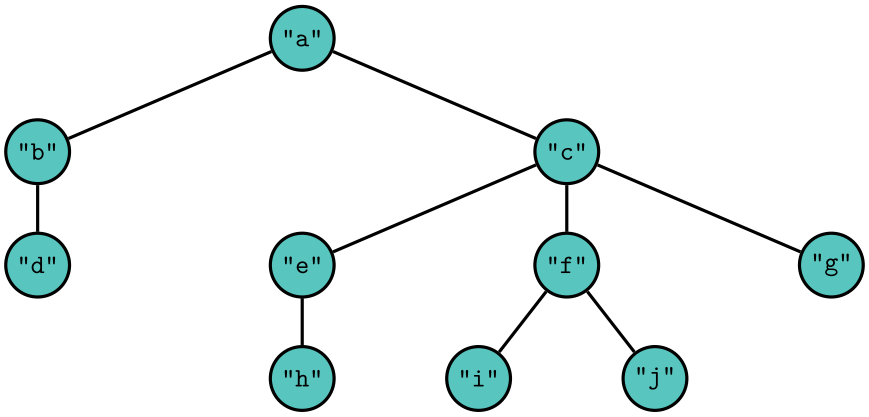
A queue by contrast employs FIFO (first in, first out). This is similar to a line at a store, where the first person to begin waiting is the first person to be served by the counter. It also has $O(1)$ insertion and deletion and $O(n)$ querying like a stack, but is built to address different problems.

2.6 Deque

A deque, pronounced as deck, is a combination of a queue and a stack. Essentially, things can be added onto the back but can be removed from either the front or the back, utilizing either queue or stack principles depending on the situation. This can be useful in emulating games in which the rules often change, and therefore require a flexible data structure.

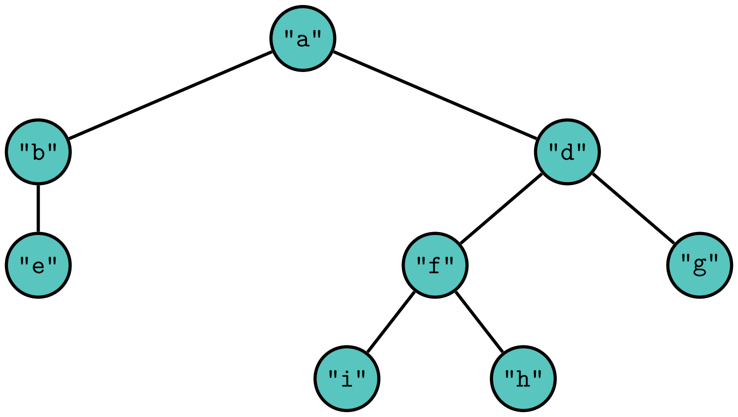
2.7 Trees

A tree is a non-linear data structure. Data in a tree is stored in "nodes." Each node in a tree can have multiple successors/children, and all nodes (except the root) have exactly one predecessor. A leaf node is a node with no children. When a tree is drawn, the root is placed at the top, with arrows pointing to its child nodes. Here is an example tree:



2.7.1 Binary Trees

Generally, in a tree each node can have any number of children, but a binary tree is a specific type of tree in which each node has at most 2 children. [Note that a binary tree is different from a binary search tree (which is explained later) because the binary tree is not organized based on the values of the nodes.] The following is an example of a binary tree:



In the tree above, “a” is the root, and “e”, “g”, “h”, and “i” are leaf nodes because they have no children.

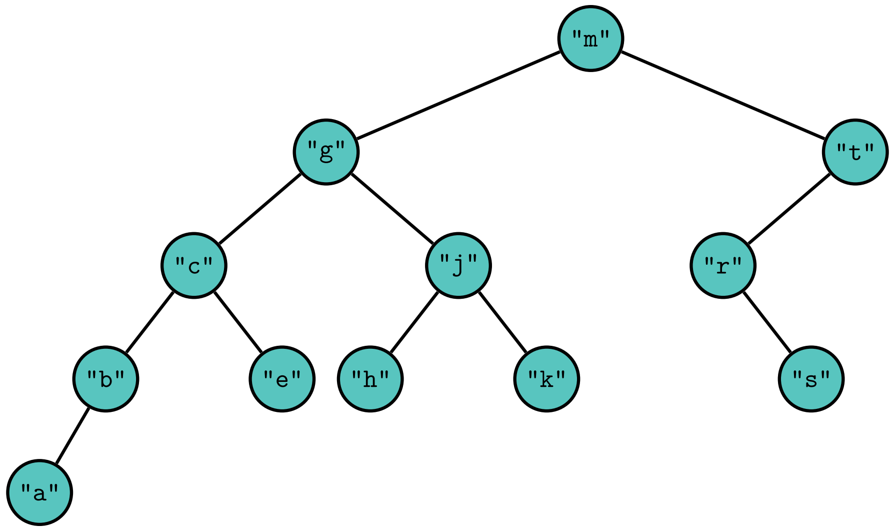
The tree has various properties. The height of a tree (also called depth) is the number of links needed to get from the root to the node furthest away from the root. For example, in the example tree, “h” (or “i”) is furthest away from the root. If we trace the path between the root and “h”, [trace: a to d, d to f, f to h], 3 links are made so the height is 3. The width of the tree is the longest path from one leaf to another (without crossing a path more than once). In this tree, the path is from "e" to "h" (or "i") and the width is 5.

Another property is the internal path length. To calculate the internal path length, we add up the path lengths from the root to each node. For the example tree, the path from the root to itself is 0. From the root to its immediate children is 1. If we do this for all the nodes and sum the paths, the internal path length is 14.

Another property is the external path length. First of all, an external node is where a node does not exist but could be drawn. For example, a leaf node has 0 children, so there are 2 external nodes there. The external path length is the sum of the paths to each external node. The path to an external node that stems from “e” would have length 3. If we do that for all the external nodes and sum the lengths, the external path length is 22.

2.7.2 Binary Search Trees

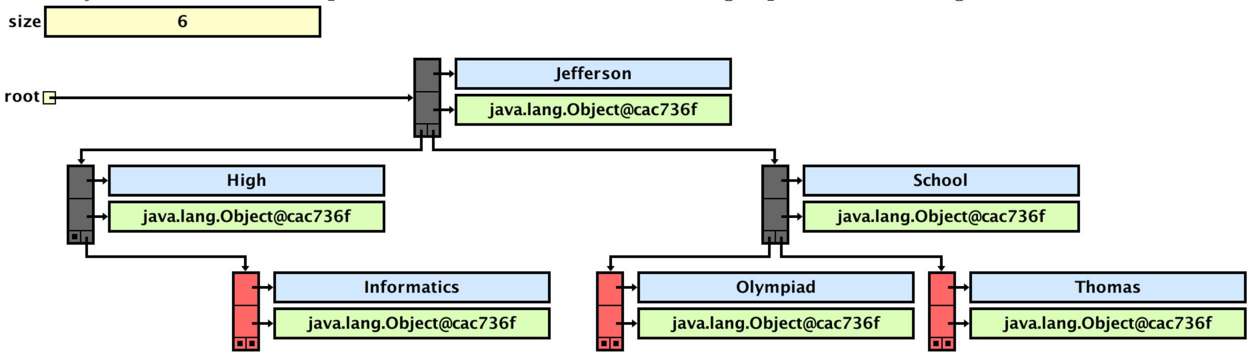
A binary search tree is a tree where every node is greater than every node in its left subtree and less than or equal to every node in its right subtree. To use a BST, we need to impose some kind of ordering on the elements stored. Typically with characters and strings, a lexicographic comparison is made. Because there is no definite bound on the number of elements of the tree, we typically create our own object or structure to represent the tree with left and right pointers. When searching for an element in the tree, we employ a binary search and find the element we desire in $O(log(N))$ time. Note that this time complexity is only for a well balanced tree. For example, if the tree’s elements are strictly increasing, then the tree becomes more like a LinkedList structure (therefore $O(N)$ in the worst case). Below is an example of a Binary Search Tree:



2.8 Sets

2.8.1 Ordered Sets

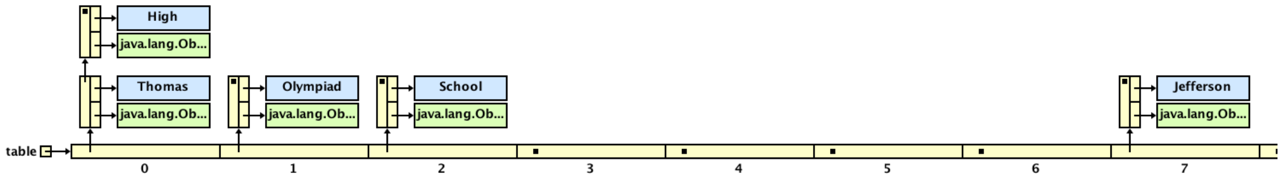
A set is a data structure that can be thought of as a bundle of unique elements. There are two distinct types of sets: one that retains order and one that does not. If a set has order, it is typically ordered in the form of a tree. In Java this is referenced as an TreeSet and in C++ it is can be invoked by std::set. In an ordered set, look up is typically $O(log(N))$ where N is the total number of elements in the set. The logarithm comes from the nature of the tree being used to keep track of the ordering of the set. Below you can see an example of an ordered set in Java being represented through a tree-based structure.



2.8.2 Unordered Sets and Hashing

A set without ordering is typically faster because of a wide-used trick known as Hashing. In brief, Hashing is a way of storing items so they can be found later on efficiently. The items are stored in a hash table at a position specified by a hash code. A hash code is an integer value calculated from any given data value or object, and is used to determine where a value should be stored in a hash table. This hash code value is generated by a hash function, a function that will always produce the same hash code given a specific data value or object. If two objects produce the same hash code, this is known as a collision.

A collision is typically handled by creating buckets that contain LinkedLists. If there is a collision and there is intent to do a look up, the run time would increase as the program would still have to go through all of the elements in the bucket. Unordered Sets use this trick to attain their $O(1)$ look up. In Java unordered sets are referenced as HashSets and in C++ they can be invoked by std::unordered_set. In the representation below, you can see the hash table for this particular unordered set (notice the collision at index 0).



Chapter 3

Basic Algorithms

Now that we've discussed some of the tools programmers use to write algorithms, lets discuss the actual algorithms! Coders use many different techniques in order to solve problems, and many of the basic methods will be discussed in this section.

3.1 Brute Force

Brute force is, to be put simply, exactly what it sounds like. In some problems, when the number of possibilities is sufficiently small, the best option may be to check each case individually. This is known as **brute force**, as we make no attempt to reduce the possibilities that we must check. For example, consider the following problem:

Problem. Larry would like to visit N cities ($1 \leq N \leq 8$). Given the distance between any two cities, determine the shortest possible path that Larry must take in order to visit every city exactly once.

As an initial approach to this problem[1], we would generate each different permutation of the N cities (such as 1, 2, 4, 5, 6, 7, 8, 3) and then determine the length of the path if we were to travel to those cities in that order, keeping track of the minimum path length. The number of ways to choose the first city to visit is given by N, the second by $N - 1$, the third by $N - 2$, and so on, and so the total number of ways to order N cities is given by $N!$, the factorial of N. The largest value of N possible is $N = 8$, and so the most number of cases to check would be $8! = 40320$. This number is well less than 10^8, the number of operations per second that a computer is capable of performing, and so using a brute force solution to this problem is sufficient.

Note that in many cases, a brute force solution will **not** be fast enough - prompting us to learn and develop more efficient algorithms. However, a brute force approach may not be a bad starting point. It is often a good idea to first attempt a problem using a brute force approach, and perform an analysis on whether or not it is sufficient to solve the problem within the time limit. If it works (like the example above), then the problem is solved; otherwise, it may provide a good basis for further refinement. It is possible that the desired solution is merely the brute force approach with some minor optimizations!

[1]Experienced readers may notice that this is the Traveling Salesman Problem, a very difficult problem to solve efficiently for large N. However, in this case, we are dealing with very small values of N to emphasize the use of the brute force technique.

3.2 Greedy Algorithm

The Greedy Algorithm, while called an algorithm, is more of a strategy. The premise of the strategy is as follows: in order to find the best solution, at each step, we will choose the option that appears to best lead us in the right direction. The assumption made by the Greedy Algorithm is that if, at each intermediate step, we make the best possible choice at that step, we will arrive at the best overall result.

This seems like a fairly reasonable strategy, as by following the best choice at each step, we should be led in the right direction. For example, consider this problem:

Problem. Kevin has $\$N$ ($1 \leq N \leq 1000$) and would like to make change with the least number of bills possible. If there are bills worth \$1, \$5, \$10, \$20, and \$100, determine the least number of bills needed to make exact change.

The optimal strategy here would be to first take as many 100-dollar bills as possible, then 20-dollar bills, then 10-dollar bills, and so on. We can see this is true as each increasing bill amount is a multiple of the previous amount. Therefore, if we have enough bills of a denomination to "trade in" for one bill of the next denomination, it will always reduce the total number of bills. Suppose we begin with N 1-dollar bills. We will trade in as many as possible for the largest denomination possible. Using the remaining bills, we will trade in as many as possible for the next largest denomination. We will repeat this process until we can no longer do so, and we have arrived at our solution. Note that this method only works when increasing bill denominations are multiples of the previous denomination. For example, if one only had bills of \$1, \$3, and \$4, and one wished to get change for \$6, the optimal solution would be 2 bills of \$3, whereas the greedy solution would call for one 4-dollar bill and two 1-dollar bills, for a total of 3 bills.

As demonstrated in that example, the greedy solution will not always produce the right answer. One must think carefully to determine if the problem they are facing is one that can be solved using a greedy strategy. One of the most famous problems whose solution takes advantage of the greedy algorithm is the *Minimum Spanning Tree* problem, which we will discuss further in a later chapter. The major drawback of the greedy algorithm is its short-sightedness: imagine a chess player employing the greedy algorithm. The player will never sacrifice a piece, or move a piece backwards, as it makes the choice that is best specifically in that turn, and neither of those events are beneficial when viewed in the context of only that one turn. In a complicated problem, such as chess, it is often the case that taking a step backwards may result in a better end result.

In practice, the greedy algorithm is often used to find approximate solutions to a difficult problem. By taking a greedy approach, one can find an answer that is close to the actual answer, in order to provide a starting point for further optimization.

3.3 Recursion

Recursion is a strategy that we can use in order to help us solve complicated problems. The idea of recursion involves solving a larger problem by breaking it down into smaller, self-similar problems. Recursion is most useful when the problem one is trying to solve depends on subproblems that are equivalent to the original. For example:

Problem. Given N ($1 \leq N \leq 100$), determine the Nth Fibonnaci number, where the Nth Fibonnaci number is defined as the sum of the $(N-1)$th and $(N-2)$th Fibonnaci number.

We can employ the use of a recursive solution in order to solve this problem, outlined below.

Algorithm 3 Fibonnaci

```
function FIBONNACI($n$)
    if $n = 1$ or $n = 2$ then
        return 1
    else
        return FIBONNACI($n - 1$) + FIBONNACI($n - 2$)
```

All recursive solutions must have two parts: the base case and the recursion. The base case specifics where to stop - without a base case, the recursion will go on forever (or at least until the computer runs out of memory)! In this solution the base case is when $N = 1$ or $N = 2$. The recursive part is the bulk of the solution. Each call to the function calls the function again with a smaller argument, until it eventually becomes 1 or 2. Because the Fibonnaci problem can be broken down into smaller, self-similar pieces, recursion is a good choice to solve this problem.

3.3.1 Recursive search

We can also use recursion to generate and explore states. If from any given state in a problem, we can reach other states, then we can simply recur on the states that we generated to explore all possible states. This is known as a depth-first search, something that will be discussed later.

This idea can be represented by a search tree. The root node of the tree is the state from which we start. Then, its children are the other states that can be generated from it, and so on. We'll go over a few examples to demonstrate how we can use this idea to solve problems.

3.3.2 Permutations

Problem. Given an integer n, print out all possible permutations of the integers $1..n$.[2] For example, if $n = 3$, the permutations would be (1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1).

This is easy to solve for a fixed value of n, such as $n = 3$, just by writing n for-loops. But we can't exactly output n for-loops. Instead, we can build each permutation of length 1. We can then recur on those results to build permutations of length 2, and so on.

Algorithm 4 Generating permutations

```
$used \leftarrow \{false, false, \ldots, false\}$
function GENERATEPERMUTATIONS(depth, prefix)
    if $depth = n$ then
        PRINT(prefix)
        return
    for $i = 1..n$ do
        if not $used[i]$ then
            $used[i] \leftarrow true$
            GENERATEPERMUTATIONS($depth + 1$, $prefix \oplus i$)
            $used[i] \leftarrow false$
```

[2]In Python, you can use `itertools.product(range(n))`!

3.3.3 Recursion stack

When we perform recursion, we are implicitly using a stack: the call stack. This stores the arguments to functions and their local variables. We can transform any function that uses recursion to a function that does not, simply by keeping track of the relevant state ourselves rather than letting the programming language do it for us. One reason we might want to do this is that programming languages limit the maximum recursion depth. So one advantage of explicitly using a stack is that you can "recurse" more deeply. However, expressing programs in this way may be less clear than plain recursion.

When performing graph traversal, we can use an explicit stack to keep track of nodes to visit rather than using recursion. One of the downsides of this, however, is that performing an inorder or postorder traversal of a graph is not as simple. This is something we will need in our algorithm for topological sort below.

3.4 Searches

Searching is the act of finding an object, if it exists (and informing us if it does not) within a data structure, most commonly an array. We will discuss two kinds of searches, each with their own advantages and disadvantages. Searches are also useful in graph theory, and we will discuss the associated searches in a later chapter.

3.4.1 Linear Search

The linear search is the most straightforward way of searching an array. We iterate over each element of the array, and check to see if it is the element that we want. As we have to check every element of the array, linear search runs in $O(n)$. Although linear search is not the most efficient method, it will always work as each element is checked.

3.4.2 Binary Search

Binary search is a more efficient method of searching. Consider how one would find a word in a dictionary, for example, "cat". We would flip to the middle of the dictionary, and examine words beginning with "m". Because "cat" comes alphabetically before "m", we do not even have to search anywhere after "m", and can recursively repeat the process on the half of the dictionary that comes before "m". Note that this relies on a key assumption: the data must already be sorted. We can use the information that sorted data provides us to eliminate half the data at each step. The number of operations that binary search needs is equivalent to how many times we can cut the data in half, until there is only one object left. This is equivalent to the logarithm in base two of the size of the data, so the runtime of binary search is $O(\log n)$.

3.5 Sorts

Sorting is a fundamental problem in computer science: one that is easy to understand and solve, but not as easy to solve *efficiently*. Formally, the objective of a sorting algorithm is to rearrange the items in an array so that they are arranged in a well-defined order. Those items can be anything: numbers, strings, and even custom data types, so long as we have some way of determining how to order them.

In order to sort an array, we must first define how we intend to compare two objects in the array. It doesn't make any sense to ask if Apple or Orange is greater until we state what we mean by "greater". Different languages implement this differently: for example, in Java, we can determine the order in which to sort elements using the `Comparable` or `Comparator` interfaces. These interfaces force us to write a method which compares an object with another, so that any object in the array must be "greater than", "less than", or "equal to" another object, which we define within the method. The sorting algorithms we analyze in this section will be similar in that they can be performed using only comparison operations; thus, we refer to them as *comparison-based sorts*.

With only a comparison operation, we can verify that an array is sorted.

Algorithm 5

```
function CheckSort(A)
    for each $a_i$ in $A$ do
        if $i > 0$ and $a_{i-1} > a_i$ then
            return false
    return true
```

3.5.1 Selection sort

The first algorithm we will consider is quite simple. First, we find the smallest element in an array; then, we put it into the first position. Next, we find the smallest element among the remaining elements, and put it into the second position. We continue until we have sorted the entire array.

We need some way of quantifying the performance of this algorithm, so we will count the number of compares and exchanges performed. We make $n-1$ total passes through the array, each of which requires one exchange. However, we make $N-i-1$ compares for $i \in [0, n-1]$, so the number of compares is approximately $N^2/2$. Note that the number of operations remains constant, even if the input data is already sorted! The selection sort algorithm is $O(n^2)$.

3.5.2 Insertion sort

Next, we consider another algorithm that is closer to how you might sort in real life. We loop through each of the items in an array, inserting each of them into the correct position among the items before it that have already been sorted. Coding insertion sort does require a bit of care.

If the array is already sorted, we only need to perform $n-1$ comparisons and 0 exchanges total! However, in the worst case, we may need to perform i comparisons and exchanges for $i \in [0, n-1]$. In total, we will perform approximately $N^2/2$ compares and exchanges. Insertion sort is also $O(n^2)$.

Chapter 4

Graph Theory

In Chapter 2, we discussed how we can represent the search space of a recursive, brute-force algorithm as a tree of states. This means that all recursive, brute-force algorithms basically perform the same thing: visit each child state and recur. In other words, we learned one way in which we can attack a specific problem using a nonspecific approach. In fact, many seemingly specific problems can be approached by using a variant of a generic algorithm. For these reasons, it is often useful to abstract specific problems into more general, mathematical objects, such as graphs.

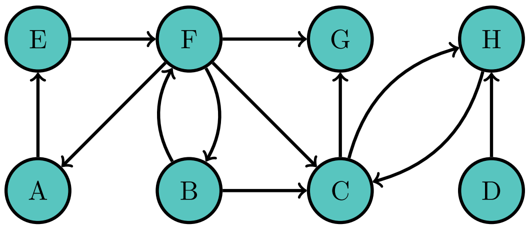
4.1 Definitions and Terminology

Formally, a *graph* set a set of *vertices*, or nodes, and a set of *edges*, where each edge connects two vertices. An *undirected* graph is one in which every edge is bidirectional. In other words, if an edge connects vertex A to vertex B, then it also connects B to A. In contrast, an *directed* graph is one in which every edge points in a single direction. Directed graphs are also commonly called *Digraphs.* Note, however, that it is possible for two directed edges to exist between two vertices. In a *weighted* graph, each edge is associated with some weight.

Graphs can take on many unusual structures. For example, an edge is allowed to point from a vertex to itself. Graphs may also contain multiple edges between the same pair of vertices. For the sake of simplicity, we will only deal with graphs that contain no self loops and multiple edges in this chapter. This is because these cases usually serve very little purpose in solving actual problems.

A *path* is a sequence of edges which connected a sequence of vertices. If the first and last vertices in the sequence are the same, then we refer to the path as a *cycle.* For example, in the digraph below, (E, F, C, H, C, G) is a valid path and (H, E, F, A) is a valid cycle. Two vertices are said to be *connected* if these exists a path between them. A connected graph is a graph in which all pairs of vertices are connectd.

A *tree* is a special kind of undirected graph that contains no cycles. Trees are particularly useful because we can choose a single node to be the *root* of the tree. Then all edges in the tree are given a natural direction, namely towards the root. We will discuss trees more in-depth in a later chapter.



4.2 Representations of Graph

Before discussing graph algorithms, it is important to understand how to represent a graph so that we have access information efficiently. In this section, we will go over the two main graph representations and their respective advantages.

For a graph with V vertices, an *adjacency matrix* is an V by V matrix A, where A_{ij} (the entry in row i and column j) represents the edge weight between vertices i and j. It no edge exists between i and j, then $A_{ij} = 0$. In an unweighted graph, we usually represent all of the edge weights as 1. Note that in an undirected graph, A would be symmetric about its main diagonal. Below is the adjacency matrix for the graph shown above.

$$\begin{array}{c|cccccccc}
 & \text{A} & \text{B} & \text{C} & \text{D} & \text{E} & \text{F} & \text{G} & \text{H} \\
\text{A} & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
\text{B} & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\
\text{C} & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\
\text{D} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\text{E} & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\text{F} & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
\text{G} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{H} & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0
\end{array}$$

An adjacency matrix allows us to determine the edge weight or existence of an edge between two vertices in $O(1)$. Furthermore, we can get all neighbors of any vertex in $O(V)$ by looping across a row of the adjacency matrix. However, The adjacency matrix requires $O(V^2)$ space, which may be undesirable if the number of vertices is large (say, around $V = 2000$). This is also true when the graph is *sparse*, or as relatively few edges.

We can also represent a graph as a list of sets, where each set contains the neighbors of an associated vertex. In an weighted graph, the sets contains pairs of vertices and their corresponding edge weights. Adjacency lists are less straightforward to implement than adjacency matrices; they are usually implemented as an array of dynamically allocated lists (which are usually provided in the standard library). Below is an adjacency list for the graph shown above.

A	E			
B	C	F		
C	G	H		
D	H			
E	F			
F	A	B	C	G
G				
H	C			

An adjacency list usually scales better for large N than an adjacency matrix because it requires $O(E)$ space complexity. In the worst case, we can also access all neighbors of any vertex in $O(V)$. The main disadvantage of an adjacency list, however, is that we cannot check the edge between any two vertices in $O(1)$. Fortunately, many algorithms do not requires constant time edge look-up, making adjacency lists an excellent choice for balancing both time and space complexity.

4.3 Graph Search Algorithms

In this section, we will apply what we've just discussed to learn a number of well-known graph algorithms.

4.3.1 Depth First Search

In section ??, we talked about using recursion to explore states, specifically in the case of generating permutations. We also represented those states as a search tree.

In what order did we traverse that tree? We started at the root node, or level 0. Then we chose a node at level 1, and then another at level 2, and so on until we reached the bottom of the tree. At that point, we've reached one node on each of the levels, but no more. Only after that will we start to check other nodes on the same levels.

Why does our recursive method behave like this? Let's say that every time we make a recursive call, we're visiting a node. So when we visit a permutation X, if we are able to we immediately generate a new permutation, we do so, and we visit the new permutation through a recursive call. Only after we get back to the original call do we generate other permutations. Thus, our use of recursion ensures that we examine the deeper elements first, which is why we call this *depth first search.*

A tree has no cycles. need to check for nodes that we've already visited

4.3.2 Breadth First Search

Another way to explore the tree is to go level by level. We start at the root node at level 0. Next, we examine its children on level 1. Then, we examine the children of the nodes on level 1, which are on level 2, and so on. How can we implement this in our code?

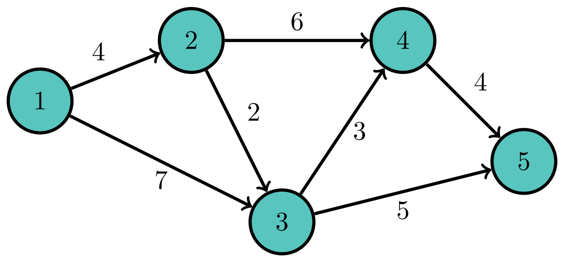
We could start by forming a list A_0 of nodes on level 0; i.e., only the root node. Then, we can find A_1 by adding all the children of the nodes in A_0, and so on, each time generating $A_(k+1)$ from A_k.

In fact, we can do this with only one array, $Q = A_0 + A_1 + ... + A_n$. While we're processing A_k, we can simply append the elements of A_k to the end of the list. What does this remind us of? That's right, a queue! For each node we remove from the front of the queue, we insert the children of that node into the back of the queue. This ensures that we process all nodes in the graph in breadth-first order.

It turns out that the only difference between a *graph* and a *tree* is that a tree is a graph without cycles. So if we want to extend our knowledge of tree traversal to graph traversal, all we have to do is avoid cycles.

4.4 Shortest Paths

The shortest-paths problem comes up a lot in competitive programming. Given two nodes u and v in a directed graph, we want to find a path between u and v such that the sum of the edge weights of the path is minimized.



4.4.1 Floyd-Warshall

The Floyd-Warshall algorithm solves the multi-source shortest paths problem; that is, it solves the shortest path problem for every pair of vertices. We use a matrix of distances *dist*, which stores the shortest distance we have found so far for each pair of vertices.

Then, for every vertex k, and every pair of vertices $i \to j$, we try to see if $dist(i, j)$ can be improved by going through k. In other words, if the current best distance from $i \to k \to j$ is shorter than the current best distance for $i \to j$ (which could be ∞). If it is, we update $dist(i, j)$, which we want to be as short as possible. This is similar to the *relax* operation that we will use for Dijkstra's algorithm, though it may not necessarily involve edges in the original graph.

Algorithm 6 Floyd-Warshall

```
$dist(i, j) \leftarrow \infty$ for vertices $i, j$
for all vertices $i$ do
    $dist(i, i) \leftarrow 0$
for all edges $(u, v)$ do
    $dist(u, v) \leftarrow weight(u, v)$
for all vertices $k$ do
    for all vertices $i$ do
        for all vertices $j$ do
            if $dist(i, j) > dist(i, k) + dist(k, j)$ then
                $dist(i, j) \leftarrow dist(i, k) + dist(k, j)$
```

One way to see why this works is to consider two vertices u and v for which we already know the shortest path from $u \to v$. Once we consider each of the vertices in that shortest path, $dist(u, v)$ will be the length of the shortest path.

4.4.2 Dijkstra's algorithm

Dijkstra's algorithm calculates the shortest path from a source vertex u to every other node in the graph. If we're only interested in the shortest path from $u \to v$, we can stop once we remove v from the priority queue.

But why does it work? Every time we remove a vertex u from pq, $dist(u)$ is monotonically increasing. This is because the priority queue always gives the minimum element, and for any u removed from the priority queue with neighbor v added to it, $dist(v) < dist(u)$.

Algorithm 7 Dijkstra's algorithm

```
for all vertices $v$ do
    $dist(v) \leftarrow \infty$
    $prev(v) \leftarrow -1$
    $visited(v) \leftarrow false$
$dist(src) \leftarrow 0$
$pq \leftarrow$ priority queue
add $src$ to $pq$ with key 0
while $pq$ is not empty do
    $u \leftarrow u$ in $pq$ with minimum $dist(u)$
    if $visited(v)$ then                                    ▷ only remove each node once
        continue
    $visited(v) \leftarrow true$
    for all neighbors $v$ of $u$ do                         ▷ relax edges
        $alt \leftarrow dist(u) + weight(u, v)$
        if not $visited(v)$ and $alt < dist(v)$ then
            $dist(v) \leftarrow alt$
            $prev(v) \leftarrow u$
            add $v$ to $pq$ with key $dist(v)$              ▷ add instead of update-key
```

An important implementation note: C++ implements a **max heap** rather than a min heap. You can get around this by negating the values of *dist*, or by using `std::greater` as the comparator.

4.4.3 Complexity

Suppose edge e connects vertices u and v. If $dist(v) > dist(u) + weight(e)$, then we define the process of *relaxing* an edge e as $dist(v) \leftarrow dist(u) + weight(e)$.

In the worst case, the algorithm will check every edge, and every edge will be relaxed. Each relaxation requires a priority queue insertion operation of complexity $O(\log V)$. Therefore, the complexity of Djikstra's Algorithm is $O(E \log V)$.

4.4.4 Bellman-Ford

If a graph contains negative edge weights, Dijkstra's algorithm cannot be used, because it is a greedy algorithm. Instead, we can use the **Bellman-Ford** algorithm, which simply relaxes all E edges, $V - 1$ times, which is the maximum path length.

If the graph contains a negative cycle, there is no "shortest" path. However, the Bellman-Ford algorithm is useful for detecting such negative cycles. To do this, we check if any distance needs to be updated after running the loop $V - 1$ times. If so, there is a path of length V, which is impossible without a negative cycle.

Algorithm 8 Bellman-Ford

```
for all vertices $v$ do
    $dist(v) \leftarrow \infty$
    $prev(v) \leftarrow -1$
$dist(src) \leftarrow 0$
for $i = 1, V - 1$ do
    for all edges $(u, v)$ do                                ▷ relax every edge
        if $dist(u) + weight(u, v) < dist(v)$ then
            $dist(v) \leftarrow dist(u) + weight(u, v)$
            $prev(v) \leftarrow u$
for all edges $(u, v)$ do                                    ▷ check for negative cycles
    if $dist(u) + weight(u, v) < dist(v)$ then
        negative cycle detected
```

The complexity of such an algorithm is obviously $O(EV)$. This can be improved by maintaining a queue of vertices whose distances were updated. Although the worst-case complexity remains the same, the average number of iterations decreases to E. However, checking for negative cycles will instead require us to look for cycles in the shortest-path tree[1] defined by *prev*.

[1]Well, if there is a cycle, it's not a tree...

Chapter 5

Dynamic Programming

Dynamic Programming is the misleading name given to a general method of solving certain optimization problems. Chances are if a problem asks you to optimize something and doesn't involve a graph, you'll want to use Dynamic Programming. As always, there are exceptions (for example, for problems with small upper bounds on N, brute force might actually be the way to go).

Dynamic Programming works on problems that can be represented as a series of sub-states. We can solve the smallest or base state first, then work up from there building up to the solution. Since we only calculate each sub-state once, the runtime of dynamic programming solutions is polynomial. A simple and overused example of dynamic programming is calculation of the Fibonacci numbers. Calculating the Fibonacci numbers recursively has an exponential time complexity $O(\varphi^N)$. But if we work from the bottom up calculating F_2, then F_3, etc, it's clear that this is $O(N)$.

0	1	1	2	3	5	8

Figure 5.1: DP Array for Calculation of the First Few Fibonacci Numbers

5.1 Knapsack Algorithm

One very common DP problem you'll see on many programming contests is the Knapsack Problem. There are various forms of the Knapsack problem, but the general idea is that you have a "knapsack" with some capacity C. You have to fill it with any number of N types of objects of some weight $W[i]$ and value $V[i]$. Given that the sum of the weights must not exceed C, find the maximum value that can be stored in the knapsack. This specific form, where you have an infinite number of discrete objects, is often known as the Integer Knapsack Problem. By way of example, consider the problem in Figure 5.2.

We know that each of the N objects is *in* the knapsack some number of times. We could iterate over each object over the number of times it could be in the knapsack, but this is clearly too slow - exponential in N. You might think that we can just greedily add the objects with the highest value to weight first. But consider

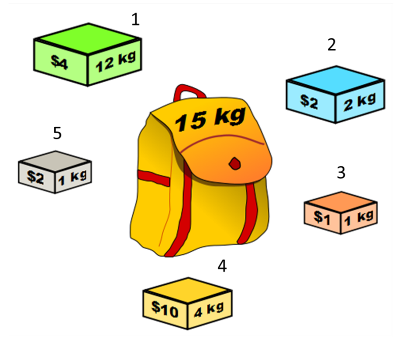


Figure 5.2: Example Knapsack Problem

the case $C = 10$ and we have two objects: $\{V_1 = 8, W_1 = 6\}$, $\{V_2 = 5, W_2 = 5\}$. We would greedily add object 1, yielding a value of 8. However, a cleverer person would add two object 2s, thus yielding a value of 10.

5.1.1 Dynamic Programming

Suppose instead of finding the maximum value obtainable with C as the capacity, we found the maximum value for some lower capacity limit. Finding the maximum for $C' = 0$, for example, is trivial. If we now take $C' = 1$, we can just find which objects have a weight of 1 and maximize over their values. In fact, as we increment C' we begin to notice a general pattern. Let's construct an array `dp[i]` that denotes the max value of a knapsack with capacity i. Then with a bit of thought, it's not hard to realize $dp[i] = \max_{\text{item } j} [dp[i - W[j]] + V[j]]$, assuming we initialize `dp[1...C]` to $-\infty$. Take a moment to understand why this works.

This function checks whether adding item j is beneficial or not. If you choose to add item j, the value will be $dp[i - W[j]] + V[j]$, since this is the sum of the value of a knapsack of capacity $C -$ Weight of item j, and the value of item j. Then, we simply want the max possible value over all possible item j's to add.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	2	4	6	10	12	13	16	20	22	24	26	30	32	34	36

Figure 5.3: DP Array - Top Row Represents Capacity, Bottom Row Represents Optimum Value

5.2 General Strategy

Now that we've seen an example problem and its dynamic programming solution, is there a general strategy for solving DP problems? In fact there is.

5.2.1 Definition of State Variables

In this step, we essentially need to figure out what the substates are and what variables will change for smaller or larger substates. We can then construct some array `dp` where accessing `dp[i][j][k]...` returns the desired answer for a given subproblem of those variables $i, j, k, \ldots$.

In the Knapsack example, our substates were the maximum values of the knapsack for smaller capacities. We could then base the substate on the capacity, creating a one-dimensional `dp` array. For some substate capacity i, `dp[i]` returned the maximum attainable value for that capacity.

5.2.2 Base Cases

Having figured out the state variables, we need to determine some base cases, usually at points where the state variables are very small.

In the Knapsack example, we might want to initialize `dp[0]` to 0 (since a knapsack of 0 capacity holds 0 value). We might also want to initialize the other elements of the array to $-\infty$ so that when we maximize, we make sure not to choose an unattainable capacity. If we use this method, we have to be careful when choosing the "maximum" that `dp[C]` is actually a positive number. Instead, we might choose the highest element of `dp` that is positive, since this will clearly be the maximum possible value of the knapsack of capacity C.

5.2.3 Transition Functions

Here's the part that usually requires lots of thought. We need to figure out the relationship between the elements in our array, and we need to be able to calculate them in a bottom-up approach so that our run time is polynomial.

In the Knapsack example, we chose the function $dp[i] = \max\limits_{\text{item } j} [dp[i - W[j]] + V[j]]$. From this, we start at `dp[1]` and continue to `dp[C]`, making sure to traverse in that order (bottom-up). What made us choose this particular function is that we know we want to consider all the possible ways to get `dp[i]`, and we want to maximize its value. So for each object j, we simply check if the value of that object added to a knapsack of capacity $i - W[j]$ is greater than `dp[i]`.

Appendix A

Input/Output

The following are some I/O examples in various languages to get you started. All programs read in a number N from the first line, proceed to read in N more numbers from the second line, and print their sum.

A.1 Java

Here we use a `BufferedReader` instead of a `Scanner` because it's faster and more reliable in the contest environment.

A.1.1 Standard I/O

```java
import java.util.*;
import java.io.*;

class sum {
    public static void main(String[] args) throws IOException {
        BufferedReader f = new BufferedReader(new InputStreamReader(System.in));
        int N = Integer.parseInt(f.readLine()); // read whole line
        int[] num = new int[N];
        StringTokenizer st = new StringTokenizer(f.readLine()); // split line by white space
        for(int k = 0; k < N; ++k) {
            num[k] = Integer.parseInt(st.nextToken());
        }
        int sum = 0;
        for(int k = 0; k < N; ++k) {
            sum += num[k];
        }
        System.out.println(sum);
        System.exit(0);
    }
}
```

A.1.2 File I/O

```java
import java.util.*;
import java.io.*;

class sum {
    public static void main(String[] args) throws IOException {
        BufferedReader f = new BufferedReader(new FileReader("sum.in"));
        PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter("sum.out")));
        int N = Integer.parseInt(f.readLine()); // read whole line
        int[] num = new int[N];
        StringTokenizer st = new StringTokenizer(f.readLine()); // split line by white space
        for(int k = 0; k < N; ++k) {
            num[k] = Integer.parseInt(st.nextToken());
        }
        int sum = 0;
        for(int k = 0; k < N; ++k) {
            sum += num[k];
        }
        out.println(sum);
        out.close(); // don't forget this!
        System.exit(0);
    }
}
```

A.2 C++

Here we use C++-style I/O. You may alternatively use C-style I/O.

A.2.1 Standard I/O

The first two lines are to speed up input. They are considered bad coding practice outside of the contest environment but are essential to get your times down if I/O is large. Note, however, that if you unlink with C-style I/O, you may not use `scanf()` and `printf()`, etc.

```
#include <iostream>
#include <fstream>

int num[100005];

int main() {
    std::ios_base::sync_with_stdio(0); // unlink C-style I/O
    std::cin.tie(0); // unlink std::cout
    std::cin >> N;
    for(int k = 0; k < N; ++k) {
        std::cin >> num[k];
    }
    int sum = 0;
    for(int k = 0; k < N; ++k) {
        sum += num[k];
    }
    cout << sum << "\n";
    return 0;
}
```

A.2.2 File I/O

```
#include <iostream>
#include <fstream>

int num[100005];

int main() {
    std::ifstream fin("palpath.in");
    std::ofstream fout("palpath.out");
    fin >> N;
    for(int k = 0; k < N; ++k) {
        fin >> num[k];
    }
    int sum = 0;
    for(int k = 0; k < N; ++k) {
        sum += num[k];
    }
    fout << sum << "\n";
    fin.close();
    fout.close(); // don't forget this!
    return 0;
}
```

A.3 Python

A.3.1 Standard I/O

```python
n = int( input() ) # input() grabs the whole line
nums = input().strip() # removes extra spaces at beginning and end, also \n
nums = nums.split() # splits at the spaces to turn it into an array of strings
nums = [int(stng) for stng in nums] #turn them into ints
print( sum( nums ) )
```

A.3.2 File I/O

```python
file = open('input.txt', 'r') # r for read
out = open('output.txt', 'w') # w for write

n = int(file.readline().strip()) # strip() isn't always necessary, but it's a good habit
nums = file.readline().strip()
nums = nums.split() # splits at the spaces to turn it into an array of strings
nums = [int(stng) for stng in nums] #turn them into ints
out.write( str( sum( nums ) ) + '\n' ) # str() and + are necessary because write() takes
    only a single string

file.close()
out.close()
```

Appendix B

Standard Libraries

B.1 Data structures

	Java	**C++**	**Python 3**
Resizable array	java.util.ArrayList	std::vector	list
Linked list	java.util.LinkedList	std::list	(none)
Array-based deque	java.util.ArrayDeque	std::deque	collections.deque
Priority queue	java.util.PriorityQueue	std::priority_queue	heapq[1]
Ordered set	java.util.TreeSet	std::set	(none)
Unordered set	java.util.HashSet	std::unordered_set	set
Ordered map	java.util.TreeMap	std::map	(none)
Unordered map	java.util.HashMap	std::unordered_map	dict

[1]Library of methods to use on a list.