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.

The majority of the material in this study guide is covered in the curriculum of AP Computer Science A, and we hope that the material will be useful as a review for those of you enrolled in that class. That being said, certain additional sections are not covered; namely, those related to trees, graph theory, and dynamic programming. As a result, these sections will account for **no more than $\frac{1}{4}$ of the points** in the competition.

We'd love to discuss how we can help you use this study guide effectively as a student, or as part of a computer science club. If you think there are errors or things we could improve, we'd certainly like to know about those too. Please don't hesitate to contact any of the officers individually, or send an email to tjioiofficers@gmail.com.

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

The TJ IOI Officers

Acknowledgments

We'd like to thank our faculty sponsor, Mr. Thomas Rudwick, 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 helping us with finances and contacting other schools. Furthermore, 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'd also like to thank the authors of the Crash Course Coding Companion, from which we took some inspiration and many diagrams: Samuel Hsiang, Alexander Wei, and Yang Liu. Their guide is available online at `https://github.com/alwayswimmin/cs_guide`.

Last but not least, we'd like to thank the officers and others who made TJ IOI possible in 2012 and 2013, whose work we are certainly building upon in this study guide as well as in other aspects. We would not be where we are without their efforts. A special shoutout goes to William Luo Qian, who essentially came up with the idea for TJ IOI from scratch, and who has kept in touch with us throughout this entire process. We hope to carry on their legacy in TJ IOI 2017.

Chapter 1

Fundamentals

1.1 Algorithms

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

Algorithm 1 Finding the maximum value in an array

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

For example, here is an example of the former: an algorithm to find the largest value in an array. We assign a value of 0 to M, and then **iterate** over each element of the array. While iterating 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 will have found the largest value in the array.

Note that the above algorithm was written in a form of *pseudocode*, rather than a formal programming language. This is because an algorithm is an idea, not lines of code. Before writing code to accomplish a task, you should first think through the problem and come up with a solution in words rather than in code. The programming language that you use is merely a 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++, Python, or any other programming language.

1.2 Code efficiency

Computers are very fast at running programs! You've probably written many programs that finish almost instantaneously. Yet push any program to its limits, and it will begin to run slowly. Adding two numbers

together takes mere nanoseconds. But what if we had to perform two billion additions? We'd quickly find out that computers aren't so instantaneous after all.

Another problem we might run into when writing programs is memory limitations. If we decided to store the result of each of those two billion additions, each of which was a 32-bit integer, we would take up 8 GB of memory! At that point, we could risk running out of memory, depending on what computer we're using.

In sum, we need some way to quantify how long a program takes to execute, or its **computational complexity**, and how much memory it uses, or its **space complexity**. We'll primarily discuss the former; however, we can use the same techniques to analyze the memory usage of programs.

1.2.1 Big-O notation

Big-O notation is one way we can describe the efficiency of an algorithm. Think back to our first algorithm (finding the maximum value 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.

1. Count the number of operations[1] the algorithm performs.
2. Take the fastest growing term.
3. Drop any constant factors.

As you've probably realized, we lose a lot of information about the algorithm's performance with big-O notation. However, its benefit is that it provides us with a simple way to broadly classify algorithms. Though there are other notations, none are nearly as widely used as big-O notation.

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.

We can also determine the Big-O efficiency of an algorithm mathematically. Given an algorithm, we can write some function $f(n)$ 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

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

[1] We define "operation" fairly loosely. Often, we count the number of array accesses or comparisons.

where M is a finite, positive value. In other words, $f(n)$ grows on the order of $g(n)$ where the exponent in the biggest term is the same for both functions. 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, which you may have used when taking the limit of a rational function in a calculus class. The term that grows fastest "dominates" the rest as the independent variable becomes large. In simpler terms, everything but the largest degree term is ignored and all coefficients are dropped.

Algorithm 2 Finding matching pairs

```
X ← 0
for all a_i in A do
    for all a_j in A where j > i do
        if a_i = a_j then
            X ← X + 1
```

As an example, consider the above algorithm. What is its complexity? 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 the algorithm would perform n^2 comparisons, and we would compare every pair twice: once for $i = j$ and another time for $j = i$.

In this case, however, the inner loop does not iterate over the entire array each time. Instead, we only check pairs for which $i < j$. Looping in the way this algorithm does ensures we will only check each pair once. This means that our program only needs half the number of operations!

However, as we have previously discussed, when using Big-O analysis, we discard constant factors, only caring about the order of the growth. Though this algorithm only needs $\frac{1}{2}n^2$ operations, the number of operations still grows as n^2. Thus, the complexity of 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 for 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. Given a problem, one useful trick is that we can infer how efficient our algorithm needs to be by looking at the input size. Figure 1.1 estimates the input size we can handle for each time complexity, if we assume that our program has one second of runtime.

Problem. Devon has N cookies ($1 \leq N \leq 6000$), each with between 0 and M chocolate chips ($1 \leq M \leq 1,000,000$). Devon would like to order these cookies by the number of chocolate chips they have, from most chocolate chips to least. Please help Devon do so!

For example, consider the above problem. We can see immediately that the number of cookies is bounded by 6000, which is approximately the limit for an $O(n^2)$ solution. As a result, without even solving the problem, we can guess that its intended solution will likely be $O(n^2)$. Note that the number of chocolate chips does not matter, as we are only using that number as a tool for comparison.

Name	**Big-O**	**Input size**
Factorial	$O(n!)$	10
Exponential	$O(2^n)$	25
Quartic	$O(n^4)$	50
Cubic	$O(n^3)$	500
Quadratic	$O(n^2)$	$5,000$
Linearithmic	$O(n \log n)$	$100,000$
Linear	$O(n)$	$1,000,000$
Logarithmic	$O(\log n)$	n/a[1]
Constant	$O(1)$	n/a[1]

Figure 1.1: Approximate input sizes for various complexities, given 1 s runtime.

Our estimated complexity will hint at how we should approach the problem. We can infer that we can afford to use a nested for-loop in our solution, in which we run n operations n times each, for a total of n^2 operations. However, we would not be able to afford a triply nested for-loop with a complexity of $O(n^3)$.

What if the number of cookies was bounded by $1 \leq N \leq 100,000$? In that case, we must find a faster solution — an $O(n^2)$ solution will no longer be sufficient! In this case, we would need an $O(n \log n)$ solution. The correct solution for this problem is to use a more efficient sorting algorithm, which we will discuss briefly in section 3.5.

[1]Don't forget that you'll generally need to read in input values, which takes linear time.

Chapter 2

Data Structures

Simply put, a **data structure** is a way of organizing data in a structured way so that it can be processed efficiently. In the context of programming contests, this data generally consists of integers, and in some cases, floating point numbers and strings. Good knowledge of data structures is essential to being able to solve certain problems efficiently.

In all likelihood, you won't ever have to code elementary data structures because the standard libraries of most programming languages already include implementations of them. However, it's important to understand how data structures work so you understand their performance characteristics and can decide which one will work best in a given situation. Understanding how simple data structures work will also help you when you need to write more complex data structures on your own.

2.1 Lists

A **list** represents a sequence of ordered elements. The elements are ordered in the sense that each is associated with an *index* that represents its position in the list. The main operations that we'd like to perform on a list are accessing the item at some position, and adding or removing an element at some position.

2.1.1 Arrays

Arrays are a fundamental type of data structure, implemented at the language level, that can be used as a list. Because they are created by requesting a block of memory from the operating system, one limitation of arrays is that they must have a *fixed size*. A good property of arrays, on the other hand, is that we can access any element of an array in $O(1)$ time, because of their low-level nature.

2.1.2 Dynamic arrays

A major caveat of arrays is their *fixed size*, which means that we need to know the maximum number of elements we'll have in an array before we create it. If we need to append an element beyond the capacity we've allocated, we're out of luck. The solution to this is pretty simple: we can simply create a larger array, and copy the elements over.

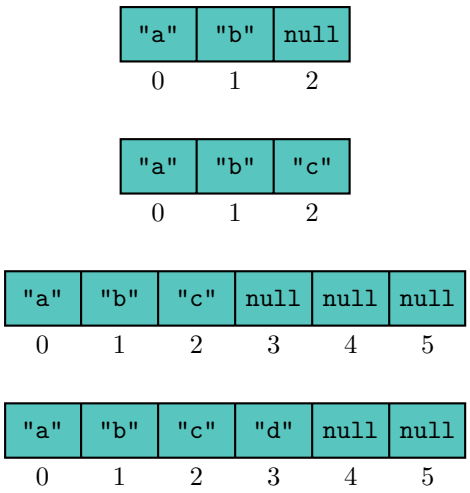


Figure 2.1: Inserting two elements into a dynamic array of size two.

A **dynamic array**, or a resizable array, provides an abstraction over this process. It maintains a *backing array* that stores the actual elements. When an insertion is impossible because there's not enough space, it "resizes" itself by creating a new backing array of a larger size, copying the elements over, then discarding the old backing array.

When we do need to resize our array, we'll typically create a backing array of *two times the size* of the previous one. This means that we'll take up more space than strictly necessary: half of the indexes in the new array will be null! However, the benefit of this is that we can add more elements without resizing the array again.

When we do have *excess capacity* in our backing array, an insertion operation at the end will only take $O(1)$. Otherwise, we'll need to perform an $O(n)$ resize. However, if we double the size of our backing array every time we resize it, the total complexity over k insertions will be approximately $O(k + \frac{k}{2} + \frac{k}{4} + \dots) = O(k)$. Thus, the *average* time complexity of an insertion operation is $O(1)$. Because the cost of resizing an array is spread out over k operations, we say that the **amortized** complexity of insertion is $O(1)$.

Because dynamic arrays are backed by arrays, we can access any element in $O(1)$. Inserting and deleting elements at the end is amortized $O(1)$. However, inserting and deleting elements at any position will take $O(N)$, since we'll need to shift all elements that come after the insertion or deletion point by one position.

2.1.3 Linked lists

A **linked list** data structure represents a list through a series of *nodes*. Each of these nodes is an object that stores the value of the item it represents, and a pointer to the next node. If there are no more items in the list, we typically represent this with a null pointer. We can imagine each node as a paper clip, as in figure 2.2: each paper clip is chained to the next.

How can we access items in this list? Since each node is only referenced by the previous node in the list, we need to access nodes $0 \dots k-1$ in order to get to item k. This is one of the large downsides of linked lists: accessing elements requires linear, not constant time. Continuing our paper clip analogy, we'd start out holding one end of the paper clip, and have to go through every paper clip to reach the other end.

Figure 2.2: A chain of paper clips.

A good thing about linked lists, however, is that they lend themselves well to recursion. As with a recursive function, there is a base case — when the pointer is null — and otherwise, a recursive case that splits the list into two parts: the current element, and everything else.

Doubly-linked lists

Above, we stated that it takes k memory accesses to access item k. So the maximum number of memory accesses in a linked list of size n would be n, to access a_{n-1}. You might notice, however, that if we could start from either end of a linked list, it would only take 1 step to access element $n-1$! This would halve the maximum number of accesses to $\frac{n}{2}$, to access element $a_{\frac{n}{2}}$.

So far, we've only talked about **singly-linked lists**, in which each node only has one pointer to the next node. To access nodes in either direction, we need to use **doubly-linked lists**, in which each node has a pointer to the next *and the previous* nodes. This uses slightly more memory, but is used more frequently because it allows us to easily modify both the front and the back of a linked list.

The easiest way to implement a doubly-linked list is using a cyclical list with a dummy node after the last item and before the first. This makes the code to implement linked lists simpler. Figure 2.3 illustrates such a linked list with elements "a", "b", and "c", with the dummy node on the left.

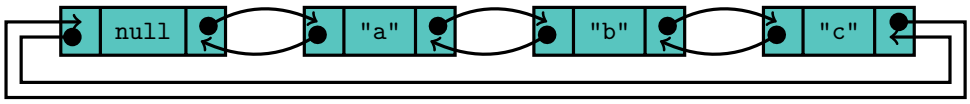


Figure 2.3: A doubly-linked list with a dummy node.

2.1.4 Efficiency

In general, dynamic arrays are more efficient than linked lists, as their values are stored in contiguous memory, and benefit from CPU caching. However, we might want to use linked lists in certain cases:

1. If we often need to insert or delete elements from the front of a list, this will take $O(N)$ time in a dynamic array. We'll address this in section 2.2.3 when we talk about deques.
2. Individual insertions at the end of a dynamic array can take up to $O(N)$ time, even if their amortized cost is $O(1)$. In some real-time applications, we'd prefer a linked list, to guarantee $O(1)$ time.
3. Furthermore, we can join together two linked lists in $O(1)$ time by changing the pointers at the end.

2.2 Stacks and Queues

2.2.1 Stacks

A **stack** is analogous to how a stack of books would work. Suppose we were creating a stack of books. We might start by putting down book A, and then book B on top of it. Then, if we tried to remove a book, we would end up removing book B, which is at the top of the stack. Because the the last element added will be the first one removed, stacks are said to operate in **LIFO** (Last in, First Out) order. We call adding an element a *push* operation, and removing an element a *pop* operation.

With a simple array, we can construct a fixed-size stack relatively simply, even in low-level code: all we have to do is maintain a counter of the number of elements in the stack. We increment the counter to add an element, and decrement it to remove an element. If we use a dynamic array, it can serve as a stack with no size restriction, with insertion and removal in amortized $O(1)$ time.

2.2.2 Queues

A **queue** operates much like a queue does in real life: imagine waiting in a checkout queue at a grocery store. The first person to get there will, no matter how many people line up behind up them, be the first to be served. In fact, the order of people who arrive in the queue will be the same as the order of people leaving the queue. Therefore, we say that the queue operates in **FIFO** (first in, first out) order. We call adding an element an *enqueue* operation, and removing an element a *dequeue* operation.

2.2.3 Deques

A **deque** (pronounced as *deck*) supports the same operations that a stack and a queue do. As a result, if we have an implementation of a deque, we can use it either as a stack or a queue, depending on whether we want elements in FIFO or LIFO order.

However, when combining these two interfaces, we run into a problem with our model: what should happen when we interlace stack operations with queue operations? Should a *push* add items at the same place as an *enqueue*, or at the opposite end? Instead, we revert to an abstract data type more similar to that of a list, with a *front* and a *back*. Our goal is to add and remove elements efficiently at either end.

Operation	**Dynamic array**	**Linked list**
Access front/back	$O(1)$	$O(1)$
Access position i	$O(1)$	$O(N)$
Add to front	$O(N)$	$O(1)$
Add to back	$O(1)$ amortized	$O(1)$
Add to position i	$O(N)$	$O(N)$

Figure 2.4: Complexity of various list operations.

The most straightforward way to implement a deque is to use a **doubly-linked list**. As we discussed above, this allows us to access, add, or remove elements at either end of a list in $O(1)$ time. Furthermore, although a linked list requires $O(n)$ time to access an arbitrary element, this doesn't matter for a deque.

2.3 Sets and Maps

2.3.1 Sets

A **set** is a collection of *unique* items. In other words, a set cannot have duplicate values — if we try to insert an item into a set and it is already in the set, our action will have no effect. The operations we'd like to perform on a set are to add an item, query whether the set contains an item, and remove an item.

One simple way to implement a set is to use a list, searching the list every time we insert an element to make sure it is not already in the list. However, this isn't particularly efficient; all operations take $O(n)$ time. We can do slightly better by sorting our list and using binary search, which reduces query time to $O(\log n)$.

Similarly, rather than maintaining an order that can be controlled by the user, set implementations typically maintain an internal order that allows them to efficiently search for items based on their value. An **ordered set** keeps the items in sorted order, which can be obtained by iterating over the items in the set. In contrast, an **unordered set** may use some other ordering not directly related to the values of the items.

2.3.2 Maps

A **map** (or a **dictionary**, a **symbol table**, or an **associative array**) allows us to associate arbitrary *keys* in the map with corresponding *values*. In some sense, this is a generalization of a list, which associates an integer index with a value. To query a map, we provide it with a key, and ask for the value associated with that key. As a result, the keys must be unique, so that we know what to answer to a query. Note, however, values do *not* have to be unique.

Maps are also closely related to sets; the items in a set are analogous to the keys in the map. The difference is that a map allows us to associate each of those items with a value. Because sets and maps have nearly identical implementations, we'll mainly consider implementations of sets instead.

2.3.3 Ordered sets

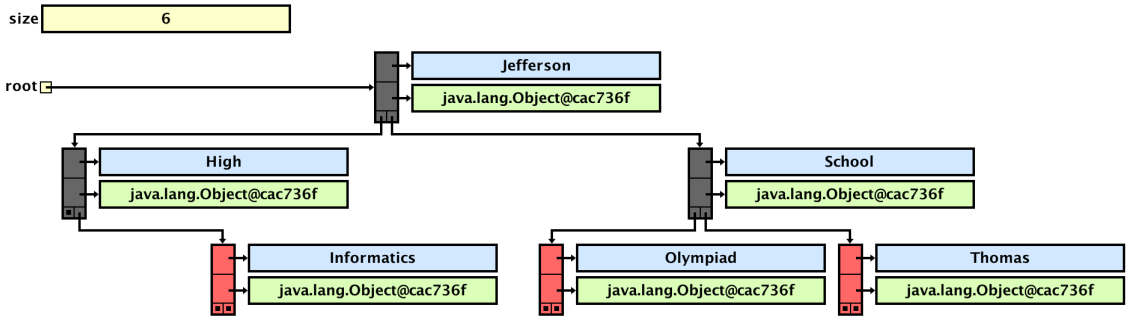


Figure 2.5: A representation of an ordered set using a BST in Java.

A common implementation for ordered sets uses a *binary search tree*, which maintains the order of its values. (We will discuss binary search trees in section 2.4.2.) Searching for, adding, and removing an item can be accomplished in $O(\log n)$, 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. Figure 2.5 is an example of an ordered set in Java being represented through a tree-based structure.

2.3.4 Unordered sets

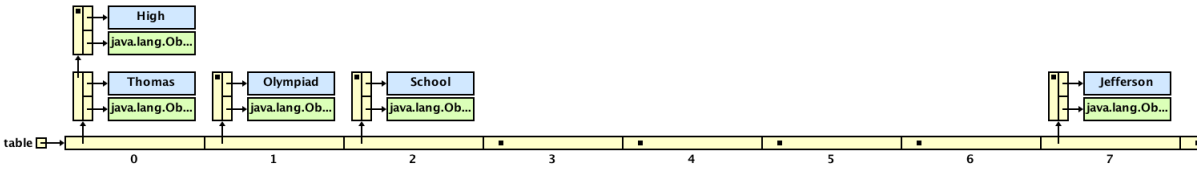


Figure 2.6: A representation of an unordered set using a hash table in Java.

In many cases, however, we may actually want to use an unordered set instead. Why? Because while $O(\log n)$ time is pretty good, a **hash table** can do even better, implementing the same operations in $O(1)$ amortized time!

Hash tables work by computing a *hash*, an integer value produced by a *hash function* based on the value of an item. This hash is then converted to an index in an array, where the item is stored. One major complication is dealing with *collisions*, in which two different items produce the same hash, though we won't discuss how to deal with those here.

Thanks to hashing, however, using an unordered set is nearly as fast as accessing elements in an array, which only takes $O(1)$. As with a dynamic array, however, we occasionally need to expand the size of a hash table, so the actual complexity is again amortized. In figure 2.6, you can see the hash table for this particular unordered set (notice the collision at index 0).

2.4 Trees

2.4.1 Definitions and terminology

A **tree** is a non-linear data structure consisting of a number of **nodes** which are each associated with a value. A node can have multiple **children**; if the node has no children, it is a **leaf node**. Correspondingly, each node in a tree also has a parent, except for the topmost node, which is the **root node**.

When a tree is drawn, the root is usually placed at the top, with arrows pointing to its child nodes below. (Mathematicians draw trees the other way around.) Figure 2.7 is an example of a tree. In this example, "a" is the *root node*, and its *children* are "b" and "c". The leaf nodes in the tree are "d", "h", "i", and "g".

The **height** of a tree (or its **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" or "j") 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.

An important property of trees is that each node can be thought of as the root of its own subtree. This lends itself to recursive algorithms for processing trees. We'll talk more about those in section 4.3.

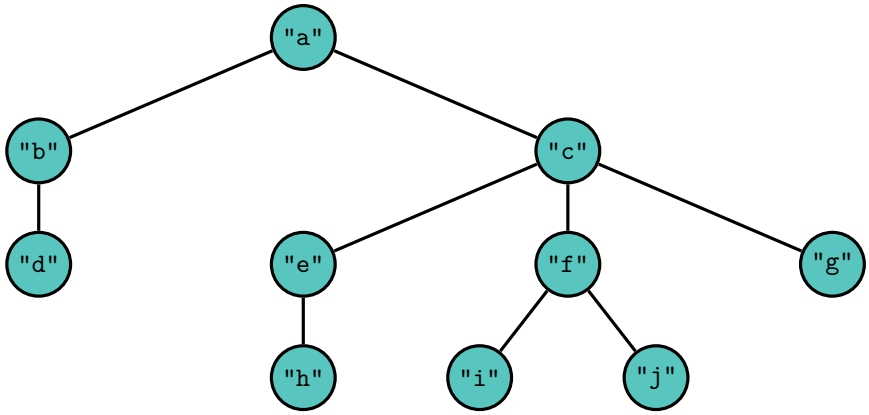


Figure 2.7: An example of a tree.

Generally, in a tree each node can have any number of children. A **binary tree** is a specific type of tree in which each node has at most two children; because of this, we often refer to the *left* and *right* child of a node. We can represent a binary tree with a class similar to that of a doubly-linked list; though instead of pointers to the previous and next node, we have pointers to the left and right children.

2.4.2 Binary search trees

A **binary search tree**, or BST, is a type of binary tree that has been structured in a specific way to make it amenable to searching. Suppose we have some node X with a value v_X. Then, for any of its children C, let its value of v_C. If C is in the left subtree of X, then $v_C < v_X$. Otherwise, if C is in the right subtree of X, then $v_C \geq v_X$. In other terms, everything in the left subtree of a given node has a lower value, and everything in the right subtree has a larger or equal value.

To use a BST, we need to impose some kind of ordering on the elements that we store. As we discussed with sorting in section 3.5, the ordering is generally implementation-defined based on the type of the elements, and can be overridden.

When searching for an element in the BST, we can employ a binary search. First, we compare the value we're looking for with the value of the root node. Then, we select either the left or the right subtree based on the result of the comparison, and we perform the same comparison. This continues until we either find the value we're looking for or reach a leaf node and conclude the value isn't present. If we assume that with every iteration we cut the number of elements in half — as with binary search — then we can find the element we desire in $O(\log n)$ time. This is better than the $O(n)$ time complexity of the list structures especially as the number of elements gets large.

Note that this complexity only applies for a *well-balanced* tree, or one that has an approximately equal number of nodes in both the left and right subtrees. An example of an unbalanced tree is one in which the tree's elements are strictly increasing. In this case, only right nodes are used, so the tree's structure will begin to approach that of a linked list, and the time complexity of searching will approach $O(n)$.

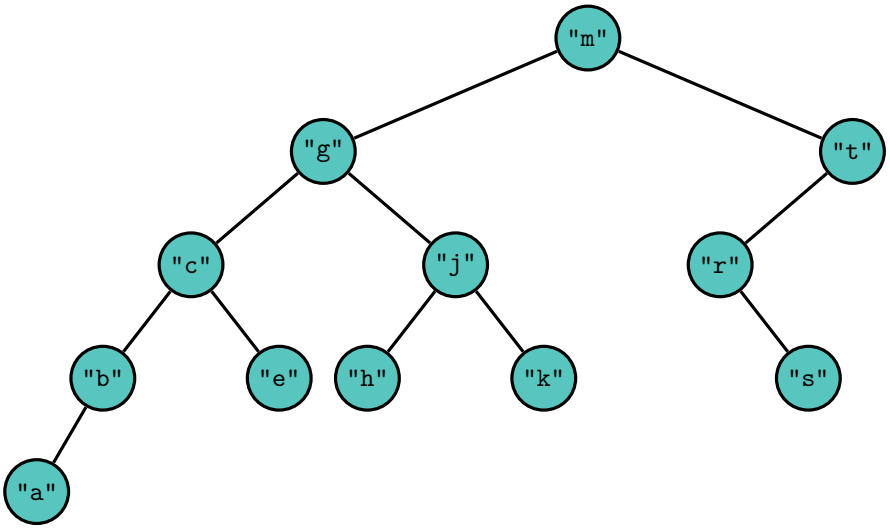


Figure 2.8: An example of a binary search tree.

Chapter 3

Basic Algorithms

Now that we've discussed some of the tools programmers use to write algorithms, let's 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

A **brute force** algorithm is one that tries every possible solution to a problem in hopes of finding one that works. Though this may seem inefficient, it may be the best option in some problems where the number of possibilities is sufficiently small. Another way of looking at it is that instead of generating solutions directly, we generate a larger set of possible solutions and filter out the ones that work.

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.

For example, consider the above problem.[1] As an initial approach to this problem, 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 search for more efficient algorithms. However, we can still 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 is not, a brute force approach may still be a starting point for further refinement. For example, coding a brute force solution may help us reveal repeated calculations, which we can take advantage of using **dynamic programming**, discussed later in section 5.

[1] You might recognize this as the *Traveling Salesman Problem*, which is a 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 algorithms

A **greedy algorithm** describes an algorithm that, in order to find the best solution, at each step, chooses the option that appears to best lead it 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 bills of \$1. 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 bill of \$4 and two bills of \$1. We'll discuss a variation of this problem in section 5.1.

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. 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, 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, we may be able to 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. More generally, a *recursive function* is a function that calls itself. We can apply this technique in a number of different ways, two of which we'll illustrate in this section.

3.3.1 Recursive functions

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 Fibonacci number, where the Nth Fibonacci number is defined as the sum of the $(N-1)$th and $(N-2)$th Fibonacci number.

We can employ the use of a recursive solution in order to solve this problem, following immediately from the mathematical definition of Fibonacci numbers.

Algorithm 3 Fibonacci

```
function FIBONACCI(n)
    if n = 1 or n = 2 then
        return 1
    else
        return FIBONACCI(n − 1) + FIBONACCI(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 until we hit a recursion limit! 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 Fibonacci problem can be broken down into smaller, self-similar pieces, recursion is a convenient way to solve this problem, though not a particularly efficient one. We'll discuss how to solve it with dynamic programming in section 5.

3.3.2 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**.

We can represent the states we traverse with a search tree (we discussed trees in section 2.4). 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. Here's an example to demonstrate how we can use this idea to solve problems.

Problem. Given an integer n, print out all possible permutations of the integers $1..n$. 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: for example, for $n = 3$, we can just write n for-loops. But given an arbitrary n, we can't just write n for-loops. Instead, we can use a recursive strategy. We'll start by building 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 ← {false, false, ..., false}
function GeneratePermutations(depth, prefix)
    if depth = n then
        Print(prefix)
        return
    for i = 1..n do
        if not used[i] then
            used[i] ← true
            GeneratePermutations(depth + 1, prefix ⊕ i)
            used[i] ← false
```

3.4 Searches

In this section, searching refers specifically to searching for an item within an array. We want to know whether the item exists, and if so, we want to know its position. We will discuss two kinds of searches, each with their own advantages and disadvantages.

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. If there is a match, then the value is returned. If there is not, it will continue to search to the end of the data collection. As in the worst case, we will have to check every element of the array, the time complexity of linear search is $O(n)$. 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*.

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 $O(n^2)$.

3.5.3 Other sorts

It turns out that there are sorts in only $O(n \log n)$ time. This time complexity is actually the lowest possible bound for comparison-based sorts! Among these sorts are *merge sort*, *quicksort*, and *heapsort*, which you may have heard of, though we won't cover the details of their implementations here.

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** consists of 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, we may sometimes allow edges to point from a vertex to itself, known as a *self-loop*. Graphs may also contain multiple edges between the same pair of vertices. For the sake of simplicity, we will not deal with graphs that contain self-loops and multiple edges in this chapter. This is because these cases usually serve 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 figure 4.1, (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 connected.

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.

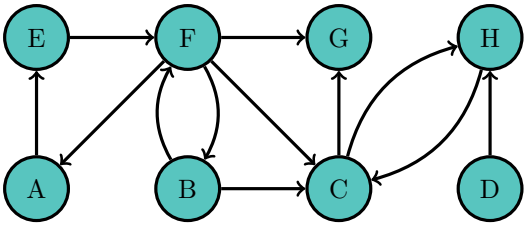


Figure 4.1: An example of a digraph.

4.2 Representations of graphs

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.

4.2.1 Adjacency matrices

For a graph with V vertices, an *adjacency matrix* is an $V \times V$ 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 in figure 4.1.

	A	B	C	D	E	F	G	H
A	0	0	0	0	1	0	0	0
B	0	0	1	0	0	1	0	0
C	0	0	0	0	0	0	1	1
D	0	0	0	0	0	0	0	1
E	0	0	0	0	0	1	0	0
F	0	1	1	0	0	0	1	0
G	0	0	0	0	0	0	0	0
H	0	0	1	0	0	0	0	0

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, an adjacency matrix requires $O(V^2)$ space, which may be undesirable if the number of vertices is large (say, around $V = 2000$). This is especially true when the graph is *sparse*, or as relatively few edges.

4.2.2 Adjacency lists

Alternatively, we can represent a graph as a list of sets, where each set contains the neighbors of an associated vertex. This is known as an **adjacency list**. 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 (see 2.1.2). 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 $O(V + E)$ 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 find the edge between any two vertices in $O(1)$. Fortunately, many algorithms do not require 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 Search trees

In section 3.3.2, 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*.

4.3.2 Depth-first search

We can use this same recursive strategy to traverse all nodes in a graph, by treating each node as a possible state and recurring on each of its neighbors.

One change that we must make, however, is that we must check whether we've already visited a node before recurring on it. This is because unlike trees, graphs can contain cycles; consequently, there may be multiple ways to reach any given node. If we don't do this, we might recurse indefinitely along a cycle.

The only difference between an *undirected graph* and a *tree* is that a tree contains no cycles. In other words, it is **acyclic**. We can express any acyclic undirected graph as a tree if we pick an arbitrary node to be the root, then treat its neighbors as its children, and so on.

We can apply depth-first search to a graph represented by an adjacency list, which stores the neighbors of each node. We can also apply this to a grid, in which the neighbors of a cell are the four adjacent cells. Interestingly, the edges that were traversed in the graph correspond to the search tree of the recursion. Assuming the graph is connected, the search tree is a spanning tree of the graph, which means that it includes all of the vertices in the graph.

Algorithm 5 Graph traversal with depth-first search

```
visited ← empty set
function DFS(v)
    add v to visited
    for w in v.neighbors do
        if w not in visited then
            DFS(w)
```

4.3.3 Breadth-first search

Another way to explore the tree is to go level by level, where level k of a tree represents all nodes with depth k. 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. Similarly, we can explore a graph using the same strategy, if we ignore repeated nodes, just as we did for depth first search. 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.

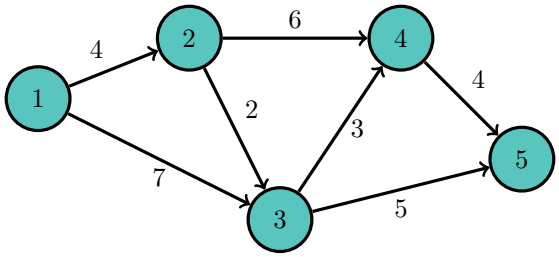
In fact, we can do this with only one array, $Q = A_0 + A_1 + ... + A_n$. While we're processing A_k, we can simply append the elements of A_k to the end of the list. What does this remind us of? That's right, a queue! For each node we remove from the front of the queue, we insert the children of that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

Algorithm 6 Graph traversal with breadth-first search

```
function BFS(v_0)
    visited ← empty set
    queue ← {v_0}
    while queue is not empty do
        remove v from queue
        for w in v.neighbors do
            if w not in visited then
                add w to queue
```

4.4 Shortest paths

The shortest-paths problem appears frequently 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 algorithm

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 7 Floyd-Warshall

```
dist(i, j) ← ∞ for vertices i, j
for all vertices i do
    dist(i, i) ← 0
for all edges (u, v) do
    dist(u, v) ← 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) ← 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. It functions using a greedy approach, continually removing the closest unvisited node to u, and then relaxing the edges connected to it. We can implement this by placing our nodes in a **priority queue**, which is a queue that allows us to efficiently remove the element with the minimum value. 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 8 Dijkstra's algorithm

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

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 $O(E \log V)$.

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 use dynamic programming to work from the bottom up calculating F_2, then F_3, etc, we can do this in only $O(N)$.

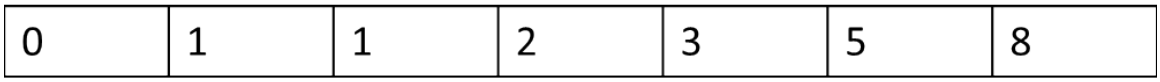


Figure 5.1: Calculation of Fibonacci numbers using DP

5.1 Knapsack problem

5.1.1 Introduction

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 of the knapsack problem in which you can have any integer number of objects is known as the **unbounded integer knapsack problem**.

By way of example, consider the problem in Figure 5.2. We know that each of the N types of objects is *in* the knapsack some number of times. We could go through each type of object and iterate over the number of times it could be in the knapsack, but this is clearly too slow—exponential in N.

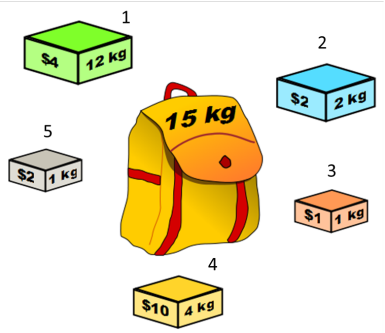


Figure 5.2: Illustration of a knapsack problem.

You might think that we could use a greedy approach (see Section 3.2), adding the objects with the highest value-to-weight ratio first. However, consider the case $C = 10$ with two types of objects: $\{V_1 = 8, W_1 = 6\}$, $\{V_2 = 5, W_2 = 5\}$. Because it has a higher value-to-weight ratio, we would add one of object 1, yielding a value of 8. The better solution would be to add two of object 2, yielding a value of 10.[1]

5.1.2 Dynamic programming approach

Suppose that instead of finding the maximum value obtainable with C as the capacity, we first found the maximum value for some lower capacity limit $c < C$. Finding the maximum value for $c = 0$, for example, is trivial: it would be 0, because we can't take any objects. If we now try $c = 1$, 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[c]` that denotes the max value of a knapsack with capacity c. Then with a bit of thought, it's not hard to realize $dp[c] = \max_{\text{item } j} [dp[c - W[j]] + V[j]]$, assuming we initialize `dp[1...C]` to $-\infty$. Take a moment to understand why this works.

This function loops through each item type j and determines which is the most beneficial when added to the bag. If you choose to add an item of type j, the remaining capacity of the knapsack will decrease to $c - W[j]$ while the knapsack's value will increase to $dp[c - W[j]] + V[j]$, which is the sum of item type j's value and the value of a knapsack with the remaining capacity $c - W[j]$. 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: An example DP array calculating the optimum value corresponding to a capacity.

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?

[1] If we were allowed to insert fractions of objects, then a greedy approach would suffice.

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 are the maximum values of the knapsack for smaller capacities. We can then base the substate on the capacity, creating a one-dimensional `dp` array. For some substate capacity i, `dp[i]` represents 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", so 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 Recurrence relation

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 $dp[i] = \max_{\text{item } j} [dp[i - W[j]] + V[j]]$. 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]`.

5.2.4 Summary

In summary, here are the steps to solving a DP problem:

1. **Identify state variables.** These are the variables that define subproblems, i.e., they are the arguments to the recursive function. In general, the more state variables there are, the slower the algorithm. For example, if we have state `dp[i][j][k]` where i ranges from 0 to A, j from 0 to B, and k from 0 to C, then the bottom-up approach must take at least $O(ABC)$ time.
2. **Determine base case(s).** Because DP solves recursive problems, there has to be a base case.
3. **Determine the recurrence relation.** This is the relationship between the overall problem and the subproblems. Also, these subproblems must overlap so that we can reuse previously calculated values.

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

```
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.

```cpp
#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

```cpp
#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

```
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

```
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.