Cr sh Course Coding Comp nion

Samuel Hsiang
Thomas Jefferson High School for Science and Technology
samuel.c.hsiang@gmail.com

lexander Wei
Phillips Exeter cademy

Yang Liu
Massachusetts Institute of Technology

December 31, 2016

https://www.dropbox.com/s/z2lur71042pjaet/guide.pdf?dl=0

https://github.com/alwayswimmin/cs_guide

cknowledgments

Contents

List of lgorithms

Preface

You might have heard of Evan Chen's Napkin, a resource for olympiad math people that serves as a jumping point into higher mathematics.[1] The Wikipedia articles on higher mathematics are just so dense in vocabulary and deter many smart young students from learning them before they are formally taught in a course in college. Evan's Napkin aims to provide that background necessary to leap right in.

I feel the same way about computer science. For most, the ease of the P Computer Science test means that the P coursework is often inadequate in teaching the simplest data structures, algorithms, and big ideas necessary to approach even silver US CO problems. On the other hand, even the best reference books, like Sedgewick, are too dense and unapproachable for someone who just wants to sit down and learn something interesting.[2] The road, for many, stalls here until college. Everyone should be able to learn the simplest data structures in Java or C++ standard libraries, and someone with problem-solving experience can easily jump right into understanding algorithms and more advanced data structures.

few important notes, before we begin.

- I'm assuming some fluency in C-style syntax. If this is your first time seeing code, please look somewhere else for now.
- It is essential that you understand the motivations and the complexities behind everything we cover. I feel that this is not stressed at all in P Computer Science and lost under the heavy details of rigorous published works. I'm avoiding what I call the heavy details because they don't focus on the math behind the computer science and lose the bigger picture. My goal is for every mathematician or programmer, after working through this, to be able to code short scripts to solve problems. Once you understand how things work, you can then move on to those details which are necessary for building larger projects. The heavy details become meaningless as languages develop or become phased out. The math and ideas behind the data structures and algorithms will last a lifetime.
- It is recommended actually code up each data structure with its most important functions or algorithm as you learn them. I truly believe the only way to build a solid

[1] In fact, I'm using Evan's template right now. Thanks Evan!

[2] Sedgewick, notably, is getting better. Check out his online companion to *lgorithms, 4th Edition*.

foundation is to code. Do not become reliant on using the standard library (`java.util`, for instance) without understanding how the tool you are using works.

Chapter 1

Fundamentals

1.1 Introduction

Welcome to competitive programming! If you know a bit about coding and you're curious about programming contests, then you're in the right place. We'll start these notes with some basic background: what happens during a programming contest, which skills they train, and how to practice and become better. In the second part of this section, we'll walk through an example contest problem together.

s you move forward, these notes will both guide you through key algorithmic and data structural ideas and provide interesting problems for you to think about. We'll try to challenge you, make you a better problem solver and programmer, and give you a sense for how the foundational concepts in computer science are intricately connected. The skills you'll pick up will be useful in not just other areas of computer science, but will also make you a stronger and more creative thinker. If you have questions or feedback about anything as you go along, feel free to contact us at `@gmail.com`. We hope you'll enjoy the world of competitive programming as much as we have. Have fun with it!

1.1.1 What is Competitive Programming?

programming contest typically consists of a collection of problems and a time limit within which they must be solved. To solve a problem, you'll have to code a program that reads some parameters as input and produces an output of the desired format. You'll then submit your code to a **judging server**, which is a server that compiles your code and executes it on a set of **test cases**. To pass, your program must solve each test case using only predetermined amounts of **time** and **memory**. That is, your program must be *efficient*. Furthermore, your program must be *correct*. To check correctness, the output for each test case is either compared to the output of a correct program written by the contest organizers or plugged into a verifier that checks that your output satisfies the desired conditions. So as a contestant, your job will be to find an efficient algorithm, write correct code, and submit.

This is just a very general description of programming contests. Each contest has its particularities in terms of scoring and the feedback you get on your submissions. Some contests will only mark your submission as correct if it correctly solves every test case, while others will give you partial credit for every test case you get correct. Some contests will also execute your submissions in real time, so you'll know if your code is correct within seconds, while others will only judge your final submissions after the contest is over. But the commonalities of these contests are in the skills they select for and train.

Problem solving is possibly the most important skill you can learn. This skill, and not whether you can crank out code quickly, is what interesting programming contests are about. You'll be given problems you'll have no idea how to solve, and you'll want to creatively reason about these problems and discover efficient solutions. Of course, being able to write clean, accurate code is also of high importance, since it is your code, not the solution in your head, that gets judged. For programming language, we recommend (and our notes will cover) coding in **Java** or **C++**. Finally, a solid understanding of algorithms and data structures will give you the tools you'll need to crack these problems. This combination of problem solving, coding, and algorithmic thinking will get you a long way in programming contests, and you'll definitely be able to apply them elsewhere too.

We'll do our best to teach you these skills by providing you with notes that show you the important algorithmic and data structural ideas, giving you pointers on how to write code, and presenting you with problems that will solidify your knowledge. I cannot emphasize how important solving the problems and writing the code is for your growth. Try to avoid reading solutions until you've given the problem a genuine shot and feel like you've stopped making progress. nd if you do read a solution, think about how you'll be able to solve a similar problem the next time it appears. fter you solve a problem, code it up, and don't give up trying to fix and optimize your code until you get all test cases accepted. Think of each bug you find as another bug you'll never see again.

But there is only so much we can do. The challenging part—persevering with difficult problems, spending long hours spent debugging, and taking time from your busy day to code—is all on you.

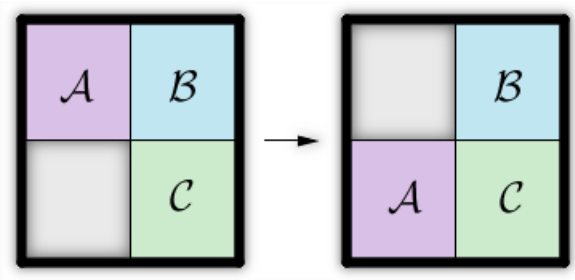
1.1.2 First Problem: Sliding Puzzles

We'll now work through an example problem, adapted from a recent Codeforces contest. On the next page is a typical problem statement, with the problem description followed by input and output specifications and sample cases. Try to come up with a solution (and possibly write some code) before reading our analysis. s in the rest of these notes, we'll follow our analysis with some discussion of how to implement the solution and provide links to C++ and/or Java code.

Sliding Puzzles

Time limit: 2s. Memory limit: 256MB.

Bessie and her best friend Elsie each own a sliding puzzle. Each of their sliding puzzles consists of a 2×2 grid and three tiles labeled $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$. The three tiles sit on top of the grid, with one grid cell left empty. To make a move, one slides a tile adjacent to the empty cell into the empty cell, as shown below:



Bessie and Elsie would like to know if there exists a sequence of moves that takes their puzzles to the same configuration. (Moves can be performed on both puzzles.) Two puzzles are considered to be in the same configuration if each tile is on top of the same grid cell in both puzzles. Since the tiles are labeled with letters, rotations and reflections are not allowed.

Input

The first two lines of the input consist of a 2×2 grid describing the initial configuration of Bessie's puzzle. The next two lines contain a 2×2 grid describing the initial configuration of Elsie's puzzle. The positions of the tiles are labeled , `B`, and `C`, while the empty cell is labeled `X`. It is guaranteed that the input contains two valid descriptions of puzzles.

Output

Print `YES` if the puzzles can reach the same configuration. Otherwise, print `NO`.

[dapted from Codeforces 645 .]

S mple Input

```
 B
XC
XB
 C
```

S mple Output

```
YES
```

S mple Input

```
 B
XC
 C
BX
```

S mple Output

```
NO
```

We present two solutions to this problem. The first is the more "obvious" one, while the second shows how solutions can often be simplified through some thinking and some clever observations.

Solution 1. One straightforward approach to this problem is to notice that there are only $4! = 24$ possible configurations that a puzzle could be in. Thus, for each puzzle in the input, it shouldn't be hard to find the list of configurations that the puzzle can reach. Once we have the lists of configurations, we can compare the two lists and check if they have an element in common. If they do, we output `YES`; otherwise, we output `NO`.

To find the list of possible configurations, we maintain a list containing all of the possible configurations we have found so far. (This list starts off as the puzzle itself.) For every configuration in our list, we check if a single move can take us to a configuration we haven't seen before. Once we find a new configuration, we append it to the list and repeat. If there exist no such new configurations, then our list contains all possible configurations for that puzzle. □

However, this solution may be somewhat difficult to implement—we have to figure out how to nicely represent each configuration as a string and make moves to reach new configurations. Writing the code to find the list of possible configurations is also a bit complex. Instead, a simple observation can reduce the trickiness of the code significantly:

Solution 2. Notice that two puzzles can reach the same configuration if and only if the $\mathcal{A}$, $\mathcal{B}$, and $\mathcal{C}$ tiles appear in the same orientation—clockwise or counterclockwise—in the two puzzles. Thus, it suffices to check if the two puzzles have the same orientation. We can do so by writing down the tiles of each puzzle in clockwise order and checking if one string is a cyclic shift of the other. □

To implement the first solution, you might want to make your list of possible configurations a **dynamic array**—that is, a Java `rrayList` or a C++ `vector`. This will allow you to easily append elements to the list.

For the second solution, once we have the tiles in clockwise order, we'll want to check if one ordering is a cyclic shift of the other. Given two strings s and t of the same length, a clever way to do this is to check if s is a substring of $t + t$. (Convince yourself that this is correct!) Take a look at a C++ implementation of Solution 2 using this trick.

1.2 More Problems!

Because problems are the most important thing in the world, below are a couple more for you to chew on. We'll cover solutions to these problems later in the chapter, but if you feel ready, try submitting your code on Codeforces first.

V sily nd C ndles

Time limit: 1s. Memory limit: 256MB.

Vasily the programmer loves romance, so he plans to write code while bathed in the warm glow of candlelight. He has a new candles, each of which burns for exactly one hour. Vasily is smart, so he can make a new candle from b burnt out candles. Vasily wonders, for how many hours can his candles light up his room if he acts optimally?

Input

The input contains two integers, a and b ($1 \leq a \leq 1000$ and $2 \leq b \leq 1000$).

Output

Print a single integer—the maximum number of hours for which Vasily's candles can keep his room lit.

[dapted from Codeforces 379 .]

S mple Input	S mple Output
4 2	7

S mple Input	S mple Output
6 3	8

Kef nd First Steps

Time limit: 2s. Memory limit: 256MB.

Kefa has started an Internet business n days ago. On the i-th day ($1 \leq i \leq n$), he made a profit of a_i dollars. Kefa loves progress, so he wants to know the length of the longest non-decreasing subsegment in her sequence of profits. (Here, a subsegment of a sequence denotes a contiguous subsequence $a_i, a_{i+1} \ldots, a_j$ ($i < j$).)

Input

The first line contains an integer n ($1 \leq 10^5$). The second line contains n integers $a_1, a_2, \ldots, a_n$ ($1 \leq a_i \leq 10^9$).

Output

Print a single integer–the length of the maximum non-decreasing sequence in Kefa's profits.

[dapted from Codeforces 580 .]

S mple Input	S mple Output
6 2 2 1 3 4 1	3

S mple Input	S mple Output
3 2 2 9	3

1.3 Input and Output

Now that you've familiarized yourself with what programming contest problems are like, let's get down to the details. The first part of solving any problem is reading the input correctly. s you may expect, doing so is very dependent on programming language. In this section, we'll cover input and output (I/O) in both Java and C++. Read our discussion for whichever one you plan to use.

1.3.1 Java

Here, we'll focus on Java I/O using `java.util.Scanner` and `java.io.PrintWriter`. There are two scenarios that you should be familiar with: standard I/O and file I/O. That is, interacting with `System.in/System.out` and files like `in.txt/out.txt`, respectively. You may have encountered standard I/O when you enter input and see output while running a program in the command line.

When using standard I/O, we can read from `System.in` using `java.util.Scanner` and output using `System.out.println`. To declare a new Scanner, simply call the constructor with `new Scanner(System.in)`. Here's a quick outline of Scanner methods:

Method	Description
`Scanner.next()`	Reads the next token in the input (i.e. up to a whitespace) and returns the token as a `string`.
`Scanner.nextLine()`	Reads the input up to a line break and returns the contents read as a `string`.
`Scanner.nextInt()`	Reads the next token in the input (i.e. up to a whitespace) and returns the token as an `int`.
`Scanner.nextLong()`	Reads the next token in the input (i.e. up to a whitespace) and returns the token as an `long`.
`Scanner.nextDouble()`	Reads the next token in the input (i.e. up to a whitespace) and returns the token as an `double`.

`System.out.println()` prints its argument and adds a newline at the end. (If you don't want the newline, you can use `System.out.print()`.) Here's an example of a `main` method that takes two integers and outputs their sum:

```java
public static void main(String args[]) {
  // hint: you should have "import java.util.*;" at the top of your code.
  Scanner sc = new Scanner(System.in);
  int x = sc.nextInt();
  int y = sc.nextInt();
  System.out.println(x + y);
}
```

File I/O is a touch more complicated. For our Scanner, we now have to call the constructor with a File object (e.g. with `new File("in.txt")`). We do the same with output for our PrintWriter (e.g. with `new File("out.txt")`). We can then use PrintWriter like we use `System.out`, by calling `pw.println()` and `pw.print()` for a PrintWriter `pw`.

However, PrintWriter also comes with a couple more usage notes. First, we should include `throws IOException` after our `main` method, since Java requires that we acknowledge the possibility of an `IOException` in the case that something goes wrong. fter we finish printing, we must also close the PrintWriter in order to ensure that everything gets written to the file. Here's a snippet showing how Scanner and PrintWriter work together with files:

```java
public static void main(String args[]) throws IOException {
  // hint: for file I/O, you should also have "import java.io.*;"
  Scanner sc = new Scanner(new File("in.txt"));
  int x = sc.nextInt();
  int y = sc.nextInt();
  PrintWriter pw = new PrintWriter(new File("out.txt"));
  pw.println(x + y);
  pw.close();
}
```

lthough more efficient methods of I/O exist, such as BufferedReader and BufferedWriter, what we've covered here should be sufficient for now. For example, it is possible to read 10^5 integers with Scanner in a fraction of a second.

1.3.2 C++

Here, we discuss I/O in C++ using the Standard Template Library's (STL) `iostream` and `fstream`. There are two scenarios that you should be familiar with: standard I/O and file I/O. You may have encountered standard I/O when you enter input and see output while running a program in the command line, whereas file I/O involves reading from and writing to files like `in.txt` or `out.txt`. In C++, standard I/O is done with the `cin` and `cout` objects in `iostream`, and file I/O is done with the `ofstream` and `ifstream` classes in `fstream`. We'll go through each of these below.

Using `cin` and `cout` is pretty straightforward. If you have a variable `x` that you want to read input into, you can do so by writing `cin >> x`. If `x` is an `int`, `double`, or `long long`, this will read the next such number in the input (up to a whitespace) into `x`. If `x` is a string, then `cin` will read similarly the input up to a whitespace into `x`. To output a variable `x` that

is of type `int`, `double`, `long long`, `string`, or `bool`, we simply write `cout << x`. To output a newline, you can write `cout << endl`. nd that's all!

Here's an example with `cin` and `cout` that outputs the sum of two integers:

```
int main() {
  // hint: you should have "#include <iostream>" at the top of your code.
  int x, y;
  cin >> x >> y;
  cout << x + y << endl;
}
```

Moving on to file I/O, suppose we want to read from `in.txt` and write to `out.txt`. We construct an `ifstream` and an `ofstream` on `in.txt` and `out.txt`, respectively. We can do so by writing `ifstream fin("in.txt")` and `ofstream fout("out.txt")`. Then `fin` and `fout` behave just like `cin` and `cout`. Here's an example:

```
int main() {
  // hint: you should have "#include <iostream>" at the top of your code.
  ifstream fin("in.txt");
  ofstream fout("in.txt");
  int x, y;
  fin >> x >> y;
  fout << x + y << endl;
}
```

lthough more efficient methods of I/O exist, such as `scanf` and `printf`, what we've covered here should be sufficient for now. For example, it is possible to read 10^5 integers with `cin` and `cout` in a fraction of a second.

1.4 Complexity

Before, we mentioned that contest problems test your ability to come up with efficient algorithms and to write accurate code. **Implementation problems** are problems that for the most part, assess the latter—that is, your ability to write code quickly and accurately. However, these problems are usually only common in easier contests, since they don't involve too much thinking or creativity; you just have to carefully implement what's written in the problem statement. Instead, most competitive programming problems ask you to come up with clever algorithms that are both fast and space-efficient.

To formally analyze the efficiency of algorithms, computer scientists use the notion of **complexity**. Complexity is roughly the number of steps an algorithm takes as a function of the input size. You can imagine algorithms that require $3n$, $n^4/3$ or even $2^n + n^2$ steps to halt for an input of size n. We categorize algorithms of different complexities using **big-O notation**: If an algorithm takes $f(n)$ steps to halt on an input of size n, we say that the algorithm is $O(f(n))$. However, this notation ignores any constant factors and lower-order

terms in the expression. For example, an algorithm that requires $100n^2 + 5$ steps is still $O(n^2)$.[1] I'll explain why in a moment—let's look at some examples for now.

Suppose we have three programs A, B, and C that require $3n$, $n^4/3 + 10$ and $2^n + n^2$ steps to finish, respectively. The complexity of the program A is $O(n)$ because we ignore the constant factor 3 on the $3n$. The complexity of the program B is $O(n^4)$. Here, we drop the constant $1/3$ and the lower-order term 10. For program C, we write its complexity as $O(2^n)$ because n^2 is a lower-order term relative to 2^n.

s to why we drop the constants and the lower order terms, consider programs A and B from above. When $n = 300$, the first program takes 900 steps, while the second program takes $2,700,000,010$ steps. The second program is much slower, despite a smaller constant factor. Meanwhile, if we had another program that runs in $5n$ steps, it would still only take $1,500$ steps to finish. Notice how the 10 after $n^4/3$ also looks irrelevant here. The takeaway is that constant factors and lower-order terms get dwarfed when comparing functions that grow at different rates.

Thus in programming contests, we usually want a program to have a sufficiently good complexity, without worrying about too much constant factors. Complexity will be the difference between whether a program gets **accepted** or **time limit exceeded**. s a rule of thumb, a modern processor can do around 10 computations each second. When you plug the maximum possible input into the complexity of your algorithm, it should never be much more than that.

We've focused on time and haven't talked much about memory so far, but memory can also be tested. However, in contests, memory limits are usually much more generous than time limits. The amount of memory a program uses as a function of n is called its **space complexity**, as opposed to the **time complexity** that we discussed earlier. If a program uses $2n^2$ bytes of memory on an input of length n, then it has a space complexity of $O(n^2)$.

1.5 More Solutions

Have you given the problems from 1.2 a shot yet and submitted some code? If you haven't, do so before reading the solutions below.

1.5.1 Vasily and Candles

Solution. Vasily and Candles is a straightforward implementation problem. Since a and b are small, we can simulate Vasily's candle burning process. We track the number of candles we have left, the number of burnt out candles, and the number of hours that have passed. Whenever we have more than b burnt out candles, we make a new candle. Once we run out of candles, we print the answer. In terms of implementation, a `while` loop is handy here. □

[1] ctually, this isn't entirely accurate. Saying an algorithm is $O(f(n))$ really means that there exists a c 0 such that the algorithm takes **t most** $c \cdot f(n)$ steps to halt.

1.5.2 Kefa and First Steps

Solution. Our first thought upon reading this problem might be to check each subsegment of the sequence a_i and see if that subsegment is non-decreasing. Unfortunately, this is too slow: There are approximately $n^2/2$ pairs of endpoints that we could choose, far too many when n can be up to 10^5. (Recall that the maximum number of computations we can do is ~10 .) Instead, we can solve this problem in $O(n)$ by executing a single `for` loop over the sequence. We maintain a counter representing Kefa's current "streak"—the number of days since her profit last decreased. If her profit decreases from day i to day $i + 1$, then we reset the counter to zero. The answer we report is the longest streak that we ever see. □

This problem is a clear example of how getting the right complexity is essential. Our initial idea, which could have been implemented in $O(n^3)$ or $O(n^2)$, was too slow. To make our program finish within time limit, we had to come up with a more efficient approach that runs in $O(n)$.

1.6 Even More Problems

For some more fun, check out the rest of the Introduction problem set that we put together on Codeforces. These problems will familiarize you with the essentials of contest programming—algorithmic thinking and writing accurate code. Enjoy!

Interlude

1.7 Sorting

To further explore the concept of complexity, we will use sorting algorithms as a case study. Sorting is just as it sounds—given a collection of objects, we want to sort them according to some ordering. For example, suppose we have a list of scores from a programming contest. In order to generate the final standings, we'll need to sort the contestants in descending order by score. Below, we present three classic sorting algorithms of varying complexity: insertion sort, merge sort, and quicksort. Insertion sort runs in $O(n^2)$, while merge sort and quicksort both run in $O(n \log n)$.

Don't worry too much about the details of these algorithms for now. You'll rarely need to implement them from scratch, since almost all modern programming languages come with built-in sorting algorithms. In our last subsection, we'll provide an example using these library functions in Java and C++ by working through a problem for which sorting is a subtask.

1.7.1 Insertion Sort

Insertion sort builds up a sorted list by inserting new elements one at a time. Inserting an element into a sorted list can be done in time proportional to the length of the list, so the runtime of this algorithm is $1 + 2 + 3 + \cdots + n = (n^2 + n)/2$, which is $O(n^2)$. Here's some pseudo code for insertion sort:

Notice that after the i-th iteration, we have a sorted list in the first i entries of the array. Insertion sort, despite being slower than merge sort and quicksort, is still useful because of its efficiency on small inputs. Many implementations of merge sort and quicksort actually use insertion sort once the problem size gets small.

Exercise 1.7.1. round how long is the longest list that you can sort with insertion sort in less than a second?

1.7.2 Merge Sort

The idea behind merge sort is the observation that given two sorted lists of length $n/2$, we only need n comparisons to merge them into a single sorted list of length n. We merge the two lists as follows: First, it is easy to find the smallest element among the two lists, since this element has to be the smallest element in one of the lists. To find the second-smallest element, we can delete the smallest element and do the same comparison again.

This method of merging lists allows us to sort using a divide and conquer approach. We split the array in half, sort each half recursively with merge sort, and then merge the two halves back together. Because our recursion goes $\log_2 n$ levels deep and requires $O(n)$ operations per level, this algorithm runs in $O(n \log n)$. (In fact, it is possible to prove that $O(n \log n)$ is optimal for comparison-based sorting algorithms, one of the few problems in computer science that has a non-trivial lower bound.)

Exercise 1.7.2. round how long is the longest list that you can sort with merge sort in less than a second?

1.7.3 Quicksort

Quicksort also uses a divide and conquer strategy to run in $O(n \log n)$ on average. We first choose a random element from the array and call it the **pivot**. We rearrange the array so that anything less than the pivot to the left of the pivot and anything greater than the pivot to the right of the pivot. This rearranging can be done in $O(n)$. Like merge sort, we can then recursively quicksort the two "halves" of the array defined by the pivot. Since we chose the pivot randomly, our problem size gets reduced down by a factor of $3/4$ most of the time, giving us $O(\log n)$ levels of recursion with $O(n)$ operations at each level. Thus quicksort runs in $O(n \log n)$ on average. We say "on average" because there do exist cases that make quicksort run in $O(n^2)$.

Exercise 1.7.3. What would happen if we chose the smallest element of the array as the pivot each time?

1.7.4 Sorting pplied

Ms. M n n 's Puzzles

Time limit: 1s. Memory limit: 256MB.

The end of the school year is near and Ms. Manana, the teacher, will soon have to say goodbye to yet another class. s a farewell present, she decides to give each of here n students a jigsaw puzzle. The shop asssistant tells Ms. Manana that there are m puzzles in the shop, but they differ in difficulty and size. Specifically, the i-th jigsaw puzzle consists of f_i pieces. Ms. Manana doesn't want to upset the children, so she wants the difference between the numbers of pieces in the largest puzzle and the smallest puzzle that she buys to be as small as possible. Help Ms. Manana compute this minimum difference.

Input

The first line contains space-separated integers n and m ($2 \leq n \leq m \leq 50$). The next line contains m space-separated integers $f_1, f_2, \ldots, f_m$ ($4 \leq f_i \leq 1000$).

Output

Print a single integer—the least possible difference that Ms. Manana can obtain.

[dapted from Codeforces 337 .]

S mple Input	**S mple Output**
4 6 10 12 10 7 5 22	5

Solution. We solve this problem by first sorting the sequence f_i. fter sorting, Ms. Manana will want to buy puzzles from a contiguous block of the sequence. (If she doesn't, then the difference between the largest and smallest puzzles will be greater than necessary.) Thus we can iterate through the sorted sequence to find the minimum difference between the endpoints of each subsegment of length n. □

Usually, when solving a sorting problem, we don't need to implement our own sorting function. Java users have `rrays.sort` in `java.utils. rrays` that does the magic for you. For those who code in C++, if you `#include <algorithm>`, you can then use `std::sort`. While coding up Ms. Manana's Puzzles, try to use the library sort function in your language.

Here are code snippets for sorting an array `arr` of length `n` in Java and C++, respectively:

```java
// hint: you should have "import java.util.*;" at the top of your code.
int[] arr = new int[n];
// do something to fill up the array.
 rrays.sort(arr);
```

```cpp
// hint: you should have "#include <algorithm>" at the top of your code.
int arr[n];
// do something to fill up the array.
std::sort(arr, arr + n);
```

Chapter 2

Big Ideas

In this chapter, we'll discuss some general problem solving ideas: brute force, depth-first search, and the greedy algorithm. We can think of these as the building blocks to more complex methods—each provides a very general approach to simplifying problems. In programming contests, they also appear frequently by themselves as the core ideas to solutions. Since the concepts we cover are independent of language, we will no longer present algorithms in concrete Java or C++ code, but rather in more abstract pseudocode.

2.1 Brute Force

Sometimes, the best way to approach a problem is to try everything. This idea of exhaustively searching all possibilities is called **brute force**. For example, if we want to unlock a friend's iPhone, we could try all of the 10^4 possible passcodes. s the name and this example suggest, brute force is often crude and inefficient. Usually we want to make some clever observations to make the problem more tractable. However, if the input size is small (check the number of operations against 10) or if we want to squeeze a few points out of a problem by solving only the small cases, brute force could be the way to go. nd if you're stuck on a problem, thinking about a brute force is not a bad way to start. Simpler, slower algorithms can often inspire faster ones. Through the following problems, we'll show you how to brutally apply the idea of brute force.

2.1.1 Square Root

`Given an integer` n, $1 \leq n \leq 10^{12}$, `find the greatest integer less than or equal to` $\sqrt{n}$ `without using any library functions. (This means you can't call functions like Math.sqrt or Math.log.)`

t first, it's not obvious how we can compute square roots. However, we can always go simple. Set $i = 1$, and while $(i+1)^2 \leq n$, increment i. That is, we increment i until increasing

it further will cause i to exceed $\sqrt{n}$. Since our answer i is at most $\sqrt{n} \leq 10^6$, our program runs in time. This is about the silliest approach we can use to calculate square roots, but hey, it works!

When implementing this algorithm, be careful about the size of n. The 32-bit `int` type in Java and C++ only holds values up to $2^{31} \quad 1 = 2,147,483,647$, which is exceeded by the maximum possible value of n. Thus we need to use a 64-bit integer type for our calculations: `long` in Java and `long long` in C++.

2.1.2 Combination Lock

`Farmer John purchases a combination lock to stop his cows from escaping their pasture and causing mischief! His lock has three circular dials, each with tick marks numbered 1 through` N `(` $1 \leq N \leq 100$`), with 1 and` N `adjacent. There are two combinations that open the lock: one combination set by Farmer John and one "master" combination set by the locksmith. The lock has a small tolerance for error, however, so it will open if the numbers on each of the dials are at most two positions away from that of a valid combination. Given Farmer John's combination and the master combination, determine the number of distinct settings for the dials that will open the lock.`

`(For example, if Farmer John's combination is` $(1, 2, 3)$ `and the master combination is` $(4, 5, 6)$`, the lock will open if its dials are set to` $(1, N, 5)$ `(since this is close to Farmer John's combination) or to` $(2, 4, 8)$ `(since this is close to the master combination). Note that` $(1, 5, 6)$ `would not open the lock, since it is not close enough to any single combination. Furthermore, order matters, so` $(1, 2, 3)$ `is distinct from` $(3, 2, 1)$`.) [ dapted from US CO 2013, Combination Lock.]`

gain, the simplest idea works. We can iterate over all possible settings of the lock, and for each setting, check if it matches either Farmer John's combination or the master combination. To do this, we can use three nested `for` loops. The first loop goes through the values for the first dial, the second loop through the values for the second dial, and the third loop through the values for the third dial. Since there are three dials, the lock has at most $N^3 \leq 10^6$ possible settings. We can check if each dial matches in $O(1)$ time, hence our algorithm runs in less than a second.

In terms of implementation, `Combination Lock` is a great example of how a problem can decompose into two easier components that we can think about separately. The first component is to use nested loops to iterate through the possible settings, which we've described above. (Nested `for` loops like this show up often!) The second component is to check if a given setting is close to either of the given combinations. If we implement a function `is_valid(a, b, c)` to do this, then the code becomes quite clean.

2.1.3 Ski Course Design

Farmer John has N hills on his farm ($1 \leq N \leq 1000$), each with an integer elevation in the range 0 to 100. In the winter, since there is abundant snow on these hills, he routinely operates a ski training camp. In order to evade taxes, Farmer John wants to add or subtract height from each of his hills so that the difference between the heights of his shortest and tallest hills is at most 17 before this year's camp.

Suppose it costs x^2 dollars for Farmer John to change the height of a hill by x units. Given the current heights of his hills, what is the minimum amount that Farmer John will need to pay? (Farmer John is only willing to change the height of each hill by an integer amount.) [dapted from US CO 2014, Ski Course Design.]

For Ski Course Design, we need to be a bit clever about how to implement our brute force. There are infinitely many ways we could change the heights of each hill, so it seems intractable to iterate over the possible heights for each hill separately. Instead, we look at the final range of the ski slope heights, which has length at most 17. This final range has to fall within the interval $[0, 100]$, hence there are less than 100 possibilities. (The possible ranges are $[0, 17]$, $[1, 18]$, $\cdots$, $[83, 100]$.) Once we fix a range, we can calculate in $O(N)$ the minimum cost to make the height of each hill fall within that range. Thus if we let M be the number of possible ranges ($M < 100$), we have an $O(MN)$ algorithm.

This problem shows that even when we brute force, we still have to think. Some approaches are better than others. In particular, we don't want to deal with cases that are irrelevant—for example, when the heights are not within a range of width 17 or when Farmer John has not used the cheapest set of changes. We also don't want our possibilities to explode out of control, which would have happened had we adjusted the height of each hill separately with nested `for` loops or recursion. By iterating over a more restricted set of possibilities, we have created a brute force solution that runs significantly faster.

2.1.4 Contest Practice

Here is a collection of problems solvable through brute force. Try working through them on your own and applying the ideas you've seen so far. (May the brute force be with you.)

2.2 Depth-First Search (DFS)

Depth-first search is a recursive search technique that provides us with another way to brute force. Instead of using nested loops to iterate through all possibilities, we can generate the possibilities using recursion, checking all possible choices in each level. Here's an example:
ll of your friends, upon realizing that it only takes four nested `for` loops to iterate through all possible 4-digit iPhone passcodes, have decided to make their passcodes n digits long.

Since you don't know n, nested `for` loops will no longer do the trick. Instead, we can use a DFS to recursively generate all n-digit passcodes.

Depth-first search works as follows: We check all passcodes starting with "0", then all passcodes starting with "1", then all passcodes starting with "2", and so on. To check all passcodes starting with "0", we check all passcodes starting with "00", then all passcodes starting with "01", then all passcodes starting with "02", and so on. To check all passcodes starting with "00", we have to check all passcodes starting with "000", then all passcodes starting with "001" and so on... (Think about why DFS is **depth-first**.)

In this way, we recursively generate all possible passcodes by extending the prefix character by character. We keep recursing until we have to check a passcode starting with a string of length n, in which case that string is the passcode itself. Thus the first passcode we check is "$00\cdots0$" and the last passcode we check is "$99\cdots9$". We implement this algorithm by writing a function that generates all passcodes given a prefix. Below is some pseudocode describing the algorithm. Make sure you understand how the function calls itself!

```
1: function GENERATEPASSCODES(depth, prefix)
2:     if depth = n then          ▷ If we've reached maximum depth, then print and return.
3:         PRINT(prefix)
4:         return
5:     for c from '0' to '9' do                    ▷ Iterates over all possible next digits.
6:         GENERATEPASSCODES(depth + 1, prefix + c)    ▷ Recurses with a longer prefix.
```

2.2.1 Permutations

`Given` n `(`$n \leq 8$`), print all permutations of the sequence` $\{1, 2, \cdots, n\}$ `in lexicographic (alphabetical) order. (For` $n = 3$`, this would be` $(1, 2, 3)$`,` $(1, 3, 2)$`,` $(2, 1, 3)$`,` $(2, 3, 1)$`,` $(3, 1, 2)$`, and` $(3, 2, 1)$`.)`

Like the passcode problem, we use DFS instead of nested `for` loops, since we don't know n. However, we have to be careful with implementation—we can use each number only once. long with our current prefix, we have to keep track of the set of numbers that we've already used. This is best done with a Boolean "used" array outside of the recursive function. Here's the pseudocode:

```
1: used ← {false, false, ··· , false}          ▷ Initialize used as an array of false values.
2: function GENERATEPERMUTATIONS(depth, prefix)
3:     if depth = n then
4:         PRINT(prefix)
5:         return
6:     for i = 1 to n do
7:         if not used[i] then
8:             used[i] ← true
9:             GENERATEPERMUTATIONS(depth + 1, prefix + i)
10:            used[i] ← false        ▷ We have to reset the used[i] variable once we're done.
```

To understand the order in which we visit the permutations, we can visualize this algorithm as traversing a tree-like structure. n animation of this algorithm for $n = 5$ is here.

2.2.2 Basketball

Two teams are competing in a game of basketball: the Exonians and the Smurfs. There are n players on the Exonian team and m players on the Smurf team, with $n + m \leq 17$. Each player has an integer skill level s between 1 and 10 . Define the strength of a set of players as the sum of their individual skill levels. In order to ensure a fair game, the Exonians and Smurfs plan on choosing two equally strong starting lineups. In how many ways can the two teams choose their lineups? (Two lineups are considered different if there exists a player who starts in one game, but not in the other.)

We use a DFS to recursively generate all possible starting lineups. Each starting lineup can be represented by a sequence of $n + m$ 0's and 1's, where a player starts if and only if he/she is assigned a 1. We do this the same way we generate all passcodes of length $n + m$. Once we have a starting lineup, it is straightforward to check for fairness. (Is it also possible to keep track of the strength of each team as we DFS? Hint: Keep an extra variable similar to "used" in `Permutations`.)

2.2.3 Problem Break

Before moving on, try to implement the DFS problems described above. You can test your code on the problem set here. Try to do some complexity analysis too. How does the runtime of these algorithms grow with respect to n?

2.2.4 Generalizing DFS

Thus far, all of the DFS solutions that we've seen have involved sequences. However, we can also use DFS in a much more general setting. In the same spirit as brute force, if we want to enumerate or construct something and we have a number of options at each step, we can recurse on each of the possible options and thereby check all possibilities. The examples below show the flexibility of depth-first search—a huge class of problems can be solved in a brute force manner like this.

2.2.5 Dungeon

Bessie is trying to escape from the dungeon of the meat packing plant! The dungeon is represented by an n-by-n grid ($2 \leq n \leq 6$) where each of the grid cells is trapped and can only be stepped on once. Some cells of the grid also

`contain obstacles that block her way. Bessie is currently located in the upper-left corner and wants to make her way to the exit in the lower-right corner. How many paths can Bessie take to escape, assuming that she avoids all obstacles and steps on no cell twice?`

We write a function `DFS(x, y)` thats runs a DFS from cell (x, y) and counts the number of paths to the lower-right corner given obstacles and previously visited squares. Upon arriving at (x, y), we mark that cell as visited. If (x, y) is the destination, we increment our answer by one. Otherwise, we try recursing in each direction—up, down, left, and right. Before recursing, we check that we don't go out of bounds and that we don't step on an obstacle or previously visited square. Once we're done counting paths from (x, y), we have to remember to mark this cell as unvisited again.

In terms of implementation, it is easiest to store the obstacles and visited cells in Boolean arrays outside of the recursive function. We can also define a function `ok(x, y)` to check if a cell (x, y) is safe to step on—if x and y are in bounds, (x, y) is not an obstacle, and (x, y) is unvisited. Finally, to avoid doing four cases, one for each direction, we can define two arrays `dx` and `dy` which contain `[1, 0, -1, 0]` and `[0, 1, 0, -1]`, respectively. Then `dx[i]` and `dy[i]` for $0 \leq i < 4$ represent the changes in x- and y-coordinates for each of Bessie's possible moves.

2.2.6 n Queens Puzzle

`Given` n $(4 \leq n \leq 8)$`, find an arrangement of` n `queens on an` n`-by-`n `chessboard so that no two queens attack each other.`

First, observe that each row must contain exactly one queen. Thus we can assume that the i-th queen is placed in the i-th row. Like before, we try putting each queen on each possible square in its row and recurse, keeping track of used columns and diagonals. When we add the i-th queen, we mark its column and diagonals (both of them) as attacked. s usual, once we're done checking all possibilities for the i-th queen, we unmark its column and diagonals. We terminate when we find an arrangement using all n queens.

2.3 Greedy lgorithms

greedy algorithm is an algorithm that makes the (locally) most desirable choice at each step. Instead of checking all possibilities as we do in a depth-first search, we choose the option that "seems" best at the time. This usually results in a fast and easy to implement algorithm. While this may not be a good way to live life—doing homework would always be relegated in favor of solving programming contest problems—there exist many computer science problems where being greedy produces optimal or almost optimal results. Keep in mind, however, that greedy algorithms usually don't work. Before coding any greedy solution, you should be able to convince yourself with a proof of correctness. Let's go through a few examples to get a better sense for what greedy algorithms are like.

2.3.1 Bessie the Polyglot

Bessie would like to learn how to code! There are n $(1 \leq n \leq 100)$ programming languages suitable for cows and the i-th of them takes a_i $(1 \leq a_i \leq 100)$ days to learn. Help Bessie determine the maximum number of languages she could know after k $(1 \leq k \leq 10^4)$ days. [dapted from Codeforces 507 .]

It's pretty easy to see that we want to learn programming languages in order of increasing a_i, starting from the smallest. Thus Bessie can obtain an optimal solution with a greedy approach. She first learns the language that requires the fewest days to learn, then the language that takes the next fewest days, and so on, until she can't learn another language without exceeding k days. To implement this, we sort a_i and find the longest prefix whose sum is at most k. Due to sorting, the complexity is $O(n \log n)$.

With greedy algorithms, we usually want to have an ordering or heuristic by which we decide what is the best choice at a given instance. In this case, our ordering was the most straightforward possible, just choosing languages in order of increasing learning time. However, figuring out how we want to greedily make decisions is not always obvious. Oftentimes, coming up with the correct heuristic is what makes greedy algorithms hard to find.

2.3.2 More Cowbell

Kevin Sun wants to move his precious collection of n $(1 \leq n \leq 10^5)$ cowbells from Naperthrill to Exeter, where there is actually grass instead of corn. Before moving, he must pack his cowbells into k boxes of a fixed size. In order to keep his collection safe during transportation, he will not place more than two cowbells into a single box. Since Kevin wishes to minimize expenses, he is curious about the smallest size box he can use to pack his entire collection.

Kevin is a meticulous cowbell collector and knows that the size of his i-th $(1 \leq i \leq n)$ cowbell is an integer s_i. In fact, he keeps his cowbells sorted by size, so $s_{i-1} \leq s_i$ for any $i > 1$. lso an expert packer, Kevin can fit one or two cowbells into a box of size s if and only if the sum of their sizes does not exceed s. Given this information, help Kevin determine the smallest s for which it is possible to put all of his cowbells into k $(n \leq 2k \leq 10^5)$ boxes of size s. [dapted from Codeforces 604B.]

Intuitively, we want to use as many boxes as we can and put the largest cowbells by themselves. Then, we want to pair the leftover cowbells so that the largest sum of a pair is minimized. This leads to the following greedy algorithm:

First, if $k \geq n$, then each cowbell can go into its own box, so our answer is $\max(s_1, s_2, \cdots, s_n)$. Otherwise, we can have at most $2k - n$ boxes that contain one cowbell. Thus we put the $2k - n$ largest cowbells into their own boxes. For the remaining $n - (2k - n) = 2(n - k)$ cowbells, we pair the ith largest cowbell with the $(2(n - k) - i + 1)$-th largest. In other words, we match the smallest remaining cowbell with the largest, the second smallest with

the second largest, and so on. Given these pairings, we can loop through them to find the largest box we'll need. The complexity of this algorithm is $O(n)$ since we need to iterate through all of the cowbells.

To prove that this greedy algorithm works, we first prove the case $n = 2k$, where each box must contain exactly two cowbells. Consider any optimal pairing of cowbells. If s_{2k} is not paired with s_1, then we can perform a swap so that they are paired without increasing the size of the largest box: Suppose we initially have the pairs (s_1, s_i) and (s_{2k}, s_j). If we rearrange them so that we have the pairs (s_1, s_{2k}) and (s_i, s_j), then $s_1 + s_{2k}, s_i + s_j \leq s_{2k} + s_j$. fter we've paired the largest cowbell with the smallest, we can apply the same logic to the second largest, third largest, and so on until we're done. Therefore, our construction is optimal if $n = 2k$. For $n < 2k$, we can imagine that we have $2k \quad n$ cowbells of size 0 and use the same argument.

This method of proving correctness for a greedy algorithm is rather common. We want to show that our greedily constructed solution is as good as an arbitrarily chosen optimal solution, so we compare where the optimal solution and our greedy solution differ. Once we find a difference, we try to transform one to the other without changing the value we're trying to optimize. In this case, since transforming the optimal solution to our greedily constructed solution doesn't make it worse, our solution must be optimal as well.

2.3.3 Farmer John and Boxes

`Farmer John has` n `(` $1 \leq n \leq 100$`) boxes in his barn. His boxes all have the same size and weight but have varying strengths--the` i`-th box is able to support at most` x_i `other boxes on top of it. If Farmer John wants to stack his boxes so that each box is directly on top of at most one other box, what is the minimum number of stacks that he'll need? [ dapted from Codeforces 388 .]`

We first observe that if a stronger box is on top of a weaker box, then we can swap the two boxes and still have a valid stacking. Therefore, an optimal solution exists where no box is stronger than any box below it. This allows us to sort the boxes in increasing order of strength and construct an optimal solution by inserting stronger boxes below piles of weaker boxes.

Initially, we start with a single empty stack. We then add boxes one-by-one as follows: If we can, we insert our new box below some stack that it can successfully support. Otherwise, if no such stack exists, we create a new stack containing only the new box. It's possible to check all existing stacks in $O(n)$; therefore this algorithm runs in $O(n^2)$. (In fact, binary searching allows us to do the checking step in $O(\log n)$.) To prove correctness, we can again compare our construction to an arbitrary optimal construction. Finishing the argument is left as an exercise for the reader.

The subtle part about this problem is the order in which we process the boxes. Processing boxes from weakest to strongest offers an elegant greedy solution, while processing from strongest to weakest offers no simple approach. (t least I haven't found one yet.) gain,

we see that ordering is important. Stepping back a bit, this problem also isn't one that immediately suggests a greedy approach. Since greedy solutions can often be unexpected, it's always worthwhile to take a moment to consider various orderings and "naïve" approaches to see if any work. Telltale hints of a greedy approach are observations about order and monotonicity—for example "we can always have stronger boxes below weaker ones."

2.3.4 Snack Time

Here's the link to the greedy problem set. Feast away!

Interlude B

2.4 Prefix Sums

Suppose we were given an array of numbers of length N and were asked to compute a series of Q queries each asking for the sum of the numbers in the contiguous subsequence of numbers indexed from i to j.

2	4	7	5	3	6	3	1	2	4	6	2	8	6	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

For example, the query on $[4, 14]$ would sum 11 numbers and return the answer 6. The naïve solution would simply resum the numbers from position i to position j each time:

```
int arr[M XN];
int query(int i, int j) {
    int sum = 0;
    for(int k = i; k <= j; ++k)
        sum += arr[k];
    return sum;
}
```

In the worst case, this for loop is $O(N)$, since each query could loop over the majority of the size N array. This solution is fine if we only have a constant number of queries. But what if Q is large? Notice that in case our naïve solution is doing a lot of repeated work. If we were to query $[3, 12]$, for example, we would sum $[4, 12]$ all over again, three fourths of the whole array!

How can we eliminate this excess work? Our intuition from this specific example would tell us to take our answer for $[4, 14]$, subtract out anything that is not in the overlap ($[13, 14]$), and add any part of the new query that we haven't yet included ($[3, 3]$). In this way we avoid summing over $[4, 12]$ twice. But we quickly see that this solution is unnecessarily complicated and does not always speed up our solution: the ranges $[1, 8]$ and $[9, 16]$ are both half the array size but have no overlap whatsoever, so each query could still be $O(N)$.

We find our answer using something called **prefix sums**. Prefix sums are an example of precomputation: performing some computation on our data before we read in any queries to

make the subsequent series of queries faster. nother example of precomputation you may have encountered before is sorting prior to a binary search, where we might be queried how many numbers in a list are greater than x for some number x. Naïvely checking each element in the list would result in a linear $O(N)$ time solution per query, while sorting beforehand costs $O(N \log N)$ but results in each query costing $O(\log N)$.

In this case, we notice that if we knew the answers to the queries $[1, j]$ and $[1, i \quad 1]$, we could simply subtract the two to get the answer to the query for $[i, j]$. Consider the following array, where the number stored in index i is the answer to the query $[1, i]$.

0	2	6	13	8	11	17	14	15	13	9	3	5	13	19	19	12
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Then the query for $[4, 14]$ would simply subtract 13 from 19 to get 6. In general, once we have constructed this array, each query takes $O(1)$, since we simply look up two numbers in the array and subtract them.

How might we construct this array in an efficient manner, though? We could use our naïve solution N times to get the N sums that we need. This would then result in an $O(N^2 + Q)$ overall solution. But the N^2 factor is very offputting. Here, we can make use of our observation that we are doing a lot of repeated work. If we know the sum for $[1, i]$, then calculating the sum for $[1, i + 1]$ is a simple matter of taking our previous answer and adding the $(i + 1)$th term in the array:

```
int N;
int arr[M XN];
int prefix[M XN];
void preprocess() {
    prefix[0] = 0;
    for(int k = 1; k <= N; ++k)
        prefix[k] = prefix[k - 1] + arr[k];
}
int query(int i, int j) {
    return prefix[j] - prefix[i - 1];
}
```

This solution is now $O(N + Q)$, much faster than the original $O(NQ)$ that we originally conceived!

So, at a high level, what happened here? We first noticed there was a lot of repetition in the work we were doing. We found a way to cut out the repetition by first finding the answer to a small (in this case, linear) number of nice cases. These nice cases have a certain structure that allow us to solve individual queries very quickly. Finally, we exploited the same structure to compute those nice cases in a quick, neat fashion.

2.5 Two Pointers

Chapter 3

Standard Library Data Structures

The purpose of this chapter is to provide an overview on how the most basic and useful data structures work. The implementations of most higher-level languages already coded these for us, but it is important to know how each data structure works rather than blindly use the standard library.

More technical explanations of all of these can be found in a language's PI. For Java, this is mostly under the package `java.util`, in the Java PI.

I strongly believe that Java is better than C++ for beginning programmers. It forces people into good coding habits, and though the lack of pointers initially frustrated me, it really does make learning general concepts liked LinkedLists much easier, as the intricacies of the C++ pointer no longer distract from the larger idea.

3.1 Generics

In general, a data structure can store any kind of data, ranging from integers to strings to other data structures. We therefore want to implement data structures that can hold any and all kinds of information. When we use a data structure, however, we might want our structure to store only one kind of information: only strings, for example, or only integers. We use *generics* to specify to an external structure that we only want it to store a particular kind of information.

```
1 rrayList<Integer> al = new  rrayList<Integer>();
```

This means that `al` is an rrayList of Integers. We can only add Integers into the rrayList, and anything removed from the rrayList is guaranteed to be an Integer. We can write `Integer i = al.get(0)` without any need to cast to an Integer.

I don't think the beginning programmer needs to know how to necessarily code a class that supports generics, since each language has its own complex set of rules governing generics. However, we use the standard library extensively in any coding environment, so it

is necessary to use a class that does support generics. I think standard classes are relatively straightforward to use but can be annoying to actually implement.

When examining Java PI or explanations of implemented functions in this chapter, the characters `E`, `V`, and `K` all can represent generics. For C++, generics are denoted by strings like `value_type`. For example, in Java, when we set `al = new rrayList<Integer>()`, `E` represents `Integer`. Otherwise, `E` simply means any object.

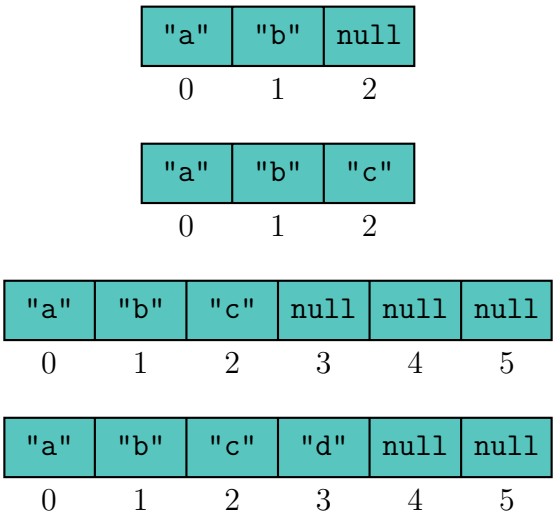
3.2 List

list is a collection of objects with an ordering. The objects are ordered in the sense that each element is associated with an index that represents its placement in the list. Users of a list have control over where in the list each object is and can access a specific element by its index, like in an array.

3.2.1 Dynamic rray

What is nice about an array? We can access or change any element we want in the array in $O(1)$ time. The problem is that an array has fixed length. It's not easy to append an element to the end of an array.

The fix to this is pretty simple. Why not just make a bigger array, and copy everything over to the new array? Then there's more room at the end to add a new element. If the backbone array runs out of space, we create a new array with double the size and keep going as if nothing happened. Therefore we now have an array of extendable size – a *dyn mic rr y*.



We see that there is still room in the array to add `"c"`, but to add more elements to the list, we must use a new array with double the length.

It's important to note that any given insertion to the structure is either $O(n)$ or $O(1)$, but there is only one $O(n)$ insertion for every $O(n)$ $O(1)$ insertions, so we still average out to constant time.

The Java implementation of a dynamic array is the `rrayList`. The C++ implementation is the `vector`.

For the following operations, think about how you would implement each and analyze its time complexity.

Function	**Java, `rrayList`**	**C++, `vector`**
dd an element to the end of the list	`boolean add(E e)`	`void push_back(const value_type& val)`[1]
insert an element to a particular index in the list, shifting all subsequent elements down one index	`void add(int index, E element)`	`iterator insert(iterator position, const value_type& val)`[2]
ccess the element stored at a particular index	`E get(int index)`	`reference operator[] (size_type n)`[3]
upd te the value of the element stored at a particular index to a new element	`E set(int index, E element)`	`reference operator[] (size_type n)`
se rch whether the list contains a particular element	`boolean contains(Object o)`	`iterator find (iterator first, iterator last, const value_type& val)`[4]
remove the element at a particular index from the list	`E remove(int index)`	`iterator erase (iterator position)`
search for and *remove* a given element from the list	`boolean remove(Object o)`	use iterators
return the *size* of the list[5]	`int size()`	`size_type size() const`

ccessing and updating elements at particular indices are very nice. They are easy to code and run in constant time. These are the bread and butter of any array. dding at the end of the list is nice as well. Checking whether some element is contained in the list is a pain, as it is $O(n)$, and adding to and removing from early in the list are more annoying.

3.2.2 Linked List

rrays are nice for accessing, say, the seventh element in the list. We extend this to the dynamic array to implement adding and removing elements to and from the end of the list

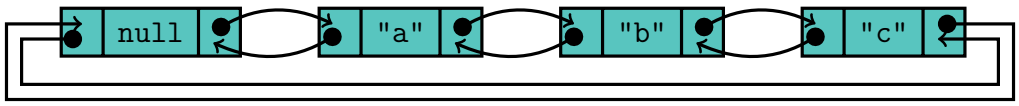
nicely. Removing elements from the beginning of the list, however, is cumbersome.

The *linked list* attempts to remedy this. It trades $O(1)$ access to any element in the list for an easier way to remove elements from either end of the list easily. Consider a chain of paper clips:

[6]

It's easy to add or remove more paper clips from either end of the chain, and from any given paper clip, it's easy to access the paper clip directly previous or next to it in the chain. If we needed the seventh paper clip in the chain, we'd need to manually count, an $O(n)$ operation. However, if we then needed to remove that paper clip from the chain, it wouldn't be that hard, assuming we kept a finger, or pointer, on the seventh paper clip.

The best way to think about and implement a linked list is through a cyclical doubly-linked list, with a dummy head. This means each element has its own node container, while the head of the list is simply a node without an element. Such a data structure looks something like this:



We see that each node maintains a pointer to its next neighbor and its previous neighbor, in addition to containing the String it stores. We can store this data in a class like the following:

```
class ListNode<E> {
    ListNode prev, next;
    E s;
}
```

[6] http://img.thrfun.com/img/078/156/paper_clip_chain_s1.jpg

If we were to insert an element after a ListNode `a`, it is necessary to update all pointers:

```
ListNode<String> b = new ListNode<String>();
b.prev = a;
b.next = a.next;
b.next.prev = b;
a.next = b;
```

Since the linked list is symmetric, inserting an element before a node is also easy. To add something to the end of the list, simply add it before the dummy head. From here it should not be too hard to implement all the important functions of a linked list.

The Java implementation of a linked list is `LinkedList`, and the C++ implementation is `list`. second C++ class that performs the same tasks but uses a backing array instead of a linked list structure is the `deque`.

Function	**Java, `LinkedList`**	**C++, `list`**	**C++, `deque`**
dd an element to the end	`boolean add(E e)`	`void push_back(const value_type& val)`	
insert[7]	`void add(int index, E element)`	`iterator insert(iterator position, const value_type& val)`	
ccess	`E get(int index)`	use iterators	`reference operator[] (size_type n)`
upd te	`E set(int index, E element)`	use iterators	`reference operator[] (size_type n)`
se rch	`boolean contains(Object o)`	`iterator find (iterator first, iterator last, const value_type& val)`	
remove the element at a particular index	`E remove(int index)`	`iterator erase (iterator position)`	
search for and *remove* a given element	`boolean remove(Object o)`	`void remove (const value_type& val)`	use iterators
size	`int size()`	`size_type size() const`	
end operations	`addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst`, `removeLast`	`push_front`, `push_back`, `pop_front`, `pop_back`	

With a linked list implemented, two other data structures immediately follow.

3.3 Stack

st ck gets its name from being exactly that: a stack. If we have a stack of papers, we can push things on the top and pop things off the top. Sometimes we peek at to access the element on top but don't actually remove anything. We never do anything with what's on the bottom. This is called *LIFO*: Last In, First Out.

Java implements the stack with `Stack`, C++ with `stack`.

Function	**Java, `Stack`**	**C++, `stack`**
push	`E push(E item)`	`void push (const value_type& val)`
pop	`E poll()`	`void pop()`
top	`E peek()`	`value_type& top()`
size	`int size()`	`size_type size() const`

Java implements `Stack` using an array-like structure. This works just as well, and is faster in practice, but I prefer the linked-list structure as a mathematical concept as it is more elegant in its relationship with the queue and more easily customizable.

3.4 Queue

queue is like a queue waiting in line for lunch. We push to the end and pop from the front. Sometimes we peek at the front but don't actually remove anything. The first person in line gets served first. This is called *FIFO*: First In, First Out.

In Java, `Queue` is an interface, and in C++, the implementation of the queue is `queue`.

Function	**Java, `Queue`**	**C++, `queue`**
push	`boolean offer(E e)`	`void push (const value_type& val)`
pop	`E poll()`	`void pop()`
top	`E peek()`	`value_type& front()`
size	`int size()`	`size_type size() const`

Since `Queue` is an interface in Java, we cannot instantiate a `Queue`, so the following statement is illegal.

```
Queue<String> q = new Queue<String>();
```

Instead, we must use `LinkedList`, so we do something like this:

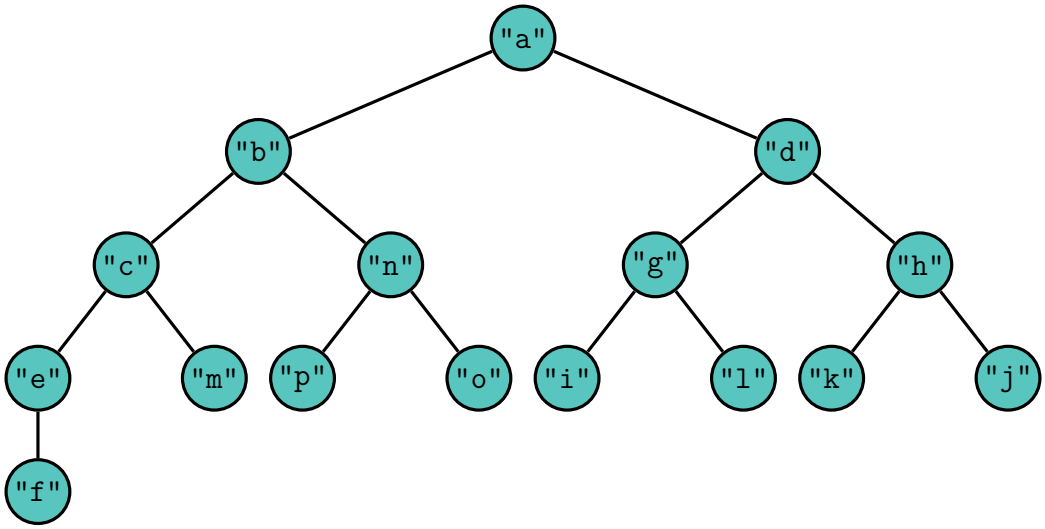
```
Queue<String> q = new LinkedList<String>();
```

This is legal because `LinkedList` implements `Queue`, making it the standard implementation of the FIFO queue.

3.5 Heap

Quite often a FIFO queue is not always desirable. For example, perhaps the string I want to remove at every given point is the one that is lexicographically least.

min heap is a tree such that every node is smaller than or equal to all of its children. max heap is a tree such that every node is larger than or equal to all of its children. Pictured is a complete binary min heap, which will be of use to us.



We see that the root of the tree will always be the smallest element. It is tempting to use a container class with a pointer to its left and its right child. However, we have a much nicer way to store *complete* binary trees with an array. Consider the following numbering of the nodes: