Code Quest Coach's Resource Packet

2022 Annual International Contest

Saturday, April 30, 2022

Table of Contents

Introduction

Thank you for supporting your students at this year's Code Quest event. We at Lockheed Martin are proud to be able to host this competition, but it would not be possible without the support of educators such as yourself. Your students represent the next generation of innovators and engineers, and you do our entire world a great service by encouraging them along this path. Particularly during the difficulties of the past two years, your willingness to go above and beyond to drive your students onward will mean more than you know.

In an effort to aid your efforts to continue this encouragement, we are providing you with this packet, which contains additional information about a selection of problems used in this year's contest. These problems were created by Lockheed Martin employees from around the world and cover a wide variety of topics, but the format of our event provides us with little opportunity to discuss those topics with the students. Teams will be given an opportunity to ask questions about the problems with our judging team following the contest, but we may not have time to address every team's concerns. This packet allows us, through you, to provide students with this extra information and further expand upon these topics.

This packet contains additional information about several of the problems presented in today's contest, including:

- An overview of the topics and disciplines covered by the problem
- A suggested approach for students to use in solving the problem
- Additional background information about the problem
- Ways to encourage students to think more about the problem or related topics

We do want to stress that the solutions provided in this packet are only suggestions; most problems can be solved in a variety of different ways, and your students may very well come up with a better solution than what we've provided here. These solutions are simply meant to help students understand the problems better, particularly if they struggled with the problem during the contest.

To that end, the walkthroughs are also tailored to the problem's difficulty level. Walkthroughs for easier problems will provide direct code snippets in each language; more difficult problems may provide sections of "pseudocode" to demonstrate the algorithms being described, if anything. These walkthroughs will aim to provide solutions that are practical for any of Code Quest's supported languages, however there are language-specific details are necessary; these will be explicitly called out in the walkthrough whenever applicable. Even if this is the case, all problems can be solved in any language that we support.

In addition to the information provided in this packet, we are able to provide solutions to all of the problems in this year's contest as written by Lockheed Martin employees, a selection of solutions submitted by students, and the official inputs and outputs used to judge student's submissions. If you

would like any of this information, please send an email to Brett Reynolds, Global Problem Packet Lead, at brett.w.reynolds@lmco.com.

Finally, we ask that you please do not post copies of this packet or any other contest materials online. All of the problems in today's contest will be made available soon on our new Code Quest Academy website at www.lmcodequestacademy.com, where students can attempt Code Quest problems at their leisure. By keeping this "answer key" private, you help to maintain the difficulty for anyone who wishes to challenge these problems again. Thank you for your cooperation, and once again for your outstanding support.

Problem 01: Null Cipher

Summary of Problem

Students must identify a hidden message by extracting specific characters from a text string.

Topics Covered

- String parsing
- Steganography

Suggested Approach

Our first task to solve this problem is to read the input string from the console. This can be done in exactly the same way as shown in the "Hello World" test problem. Rather than immediately printing it back out, however, you'll want to make sure it gets saved to a string variable, which we'll call 'ciphertext'.

Language	Input Reading Code

Java	`try(java.util.Scanner input = new java.util.Scanner(System.in)){`
	`  String ciphertext = input.nextLine();`
	`  // more to come...`
	`}`

Python	`ciphertext = sys.stdin.readline().rstrip();`
C++	`string ciphertext;` `getline(cin, ciphertext);`
C#	`string ciphertext = Console.ReadLine();`

In this problem, we need to scan through the entire ciphertext, and identify any characters that come immediately after a vowel. Those characters should be printed back out in order to reproduce the hidden message; everything else can be ignored. Fortunately, it's very easy to read through an entire text string character by character. A string is essentially an array of characters, and in fact, many languages treat strings and character arrays as the same thing, or at least very similar. Most important to us is that we can iterate over each character in a string just as we could iterate over elements in an array.

Before we do that, we need to set up a way for us to identify when we want to print out a character, and when we're not concerned about it. Since those are the only two possible outcomes for each character - print or don't - a Boolean value is ideal for keeping track of this. A Boolean value can be set to either 'true' or 'false' and can be used in logic statements (like if blocks or while loops) to control the behavior of your program. For this problem, we'll want to declare a variable called 'printLetter' and set it to 'false' by default. As we read through the ciphertext, we'll set this to 'true'

any time we see a vowel. Any time 'printLetter' is true, we'll want to print the next letter to the console, then set 'printLetter' back to 'false' until we find the next vowel.

Language	printLetter Initialization Code
Java	`boolean printLetter = false;`
Python	`printLetter = False`
C++	`bool printLetter = false;`
C#	`bool printLetter = false;`

Now we're ready to read the string. As mentioned before, we want to step through the string one character at a time. Just like arrays, each character in a string has an index number, starting with 0 for the first character. The last character in a string will be equal to the length of the string minus one. In most languages, we can use these index numbers to move through the string with a for loop and obtain each character from it in order.

Language	Ciphertext Reading Code

Java	`for(int n = 0; n < ciphertext.length(); n++){`
	`  char letter = ciphertext.charAt(n);`
	`  // processing code here (that's next!)`
	`}`

Python	`for n in range(len(ciphertext)):`
	`  letter = ciphertext[n]`
	`  # processing code here (that's next!)`

C++	`for(int n = 0; n < ciphertext.length(); n++){`
	`  char letter = ciphertext[n];`
	`  // processing code here (that's next!)`
	`}`

C#	`for(int n = 0; n < ciphertext.Length; n++){`
	`  char letter = ciphertext[n];`
	`  // processing code here (that's next!)`
	`}`

Now we're ready to write the processing code we discussed earlier. This will go inside the for loop, in place of the "processing code here" comment in the table above. First, we should check the value of our 'printLetter' variable. If it is 'true', we should print 'letter' to the console, then set 'printLetter' to 'false'. Otherwise, we should check the value of 'letter'. If it's an a, e, i, o, or u, then we should set 'printLetter' to 'true'.

Language	Letter Processing Code

Java	`if(printLetter){`
	`  System.out.print(letter);`
	`  printLetter = false;`
	`}`
	`else if(letter == 'a' || letter == 'e' || letter == 'i'`
	`|| letter == 'o' || letter == 'u'){`
	`  printLetter = true;`
	`}`

Python	`if printLetter:`
	`  print(letter, end='')`
	`  printLetter = False`
	`elif letter == 'a' or letter == 'e' or letter == 'i'`
	`or letter == 'o' or letter == 'u':`
	`  printLetter = True`

C++	`if(printLetter){`
	`  cout << letter;`
	`  printLetter = false;`
	`}`
	`else if(letter == 'a' || letter == 'e' || letter == 'i'`
	`|| letter == 'o' || letter == 'u'){`
	`  printLetter = true;`
	`}`

C#	`if(printLetter){`
	`  Console.Write(letter);`
	`  printLetter = false;`
	`}`
	`else if(letter == 'a' || letter == 'e' || letter == 'i'`
	`|| letter == 'o' || letter == 'u'){`
	`  printLetter = true;`
	`}`

With that, our solution is complete! Your program will step through every character in the ciphertext string, and print out only those letters that appear immediately after a vowel character. The last thing to do is print a newline character to the console, to ensure that each of our secret messages is properly separated from the others. Make sure to put this outside the for loop, so you don't print each letter on a separate line!

Language	Newline Code
Java	`System.out.println();`
Python	`print()`
C++	`cout << '\n';`
C#	`Console.WriteLine();`

Additional Background

Ever since mankind developed written communication, there have been things that people have wanted or needed to write down, but didn't necessarily want just anyone to be able to read. A military general, for example, might need to relay orders to his officers to prepare for a surprise attack against the enemy. However, if those orders were to be intercepted by the enemy, the surprise will be ruined, likely along with the general's army. Throughout the course of history, people have used three main methods to keep their written secrets secret: codes, ciphers, and steganography.

You've likely heard of codes and ciphers before; while they're often used interchangeably, they are slightly different. A code involves replacing entire words or phrases, whereas a cipher replaces the individual letters. For example, the phrase "the eagle has left the nest" could be a code meaning "the President has left the White House;" if that same sentence were encrypted with a cipher instead, it would likely be a string of random gibberish. Either way, the goal of codes and ciphers - collectively known as "cryptography" - is to obscure a message so that it cannot be read.

Steganography takes a different approach to secrecy. Rather than making the message illegible, it aims to make it appear as though there is no message at all (or at least that the message is completely innocent and has no secret meaning). In the earliest days of written communication, this may have been the only way to communicate secretly. Many early writing systems were pictographic rather than alphabetic, and even those that did use alphabets may not have their alphabets clearly defined. This would have made ciphers nearly impossible, and even codes would have been very difficult to understand. On the other hand, simply hiding a message is easy enough for anyone to do, provided that the intended recipient knows how to find it again.

The methods used to hide messages have varied widely throughout history, and some of them have been remarkably inventive:

- Messages tattooed on a messenger's skin could have been covered over with clothing or hair (although this method is largely considered apocryphal now, due to its wildly impractical nature)
- Before paper was invented, messages were often inscribed into wax tablets held in wooden frames; by melting the wax off, a message could be carved into the wooden backing, then hidden under a fresh layer of wax (which could hold an entirely different message).
- "Invisible inks" such as lemon juice would dry transparent but could be made visible by applying heat or other chemicals.
- A paper with punched holes could be laid over an otherwise innocent letter to reveal specific words that make up a secret message.
- By using a pattern, the sender of a hidden message can ensure their communication is not discovered, but remains secret. For example, every fifth word in that last sentence forms the phrase "the message is secret."

- Microdots are extremely small sections of transparent film that can contain printed messages shrunk to a microscopic scale; when glued on top of a period in a normal-sized document, they become nearly invisible.
- Digital images can have additional information encoded inside them, either in metadata or within the image itself.

With many of these methods, the message is plain to see if you know how to look for it, and some do provide clear hints. Microdots, for example, were often made slightly reflective so spies would be able to find and remove them; of course, this also meant that counter-intelligence agents could spot them as well. As a result, relying solely on steganography as a means of security is a poor choice; most often, steganography and cryptography are used together. By hiding the message, you reduce the risk of it being intercepted and decrypted; by encrypting the message, you ensure that it can't be immediately read without going through the effort of decrypting it.

Further Discussion Topics

- Can you think of some other ways to hide a message? How effective might they be in preventing your message from being intercepted?
- Write a program that can "encrypt" a message using the method shown in this problem. Make sure to add plenty of "fluff" characters before, after, and in between each letter of your message so it can't be easily read. When done, feed it back through your solution to see if it works!

Problem 03: The Good Ship Input

Summary of Problem

Students must read in a listing of systems, then identify all systems which appear in that list, but not in a second list.

Topics Covered

- Advanced data structures

Suggested Approach

To begin, we need to know how much data we're working with. The first line of input provides two integers which show, first, the number of systems we could be dealing with, then the number of systems being reported as functional by the ship. This will let us know when to stop reading data so we don't run into the next test case. In most languages, this can be done in one of three ways.

First, we can read in the entire line as a string, then split the string along the space separating the two numbers; these smaller strings can then be converted to integers. Most languages supported by Code Quest include a .split() function on strings, which returns an array containing substrings, divided along a given delimiter string or character. For example:

Language	Example Split() Code

Java	`String foo = scanner.nextLine();`
	`String[] fooParts = foo.split(" ");`
	`int databaseCount = Integer.parseInt(fooParts[0]);`
	`int shipCount = Integer.parseInt(fooParts[1]);`

Python	`foo = sys.stdin.readline().rstrip()`
	`fooParts = foo.split(" ")`
	`databaseCount = int(fooParts[0])`
	`shipCount = int(fooParts[1])`

C++	`// C++ doesn't provide a split() function`

C#	`string foo = Console.ReadLine();`
	`string[] fooParts = foo.Split(new Char[]{' '});`
	`int databaseCount = Convert.ToInt32(fooParts[0]);`
	`int shipCount = Convert.ToInt32(fooParts[1]);`

Second, you can find the index of the space (particularly since you know there will only be one space in that line) and create substrings around that index. This works in any language, although the way to get the substrings will vary:

Language	Example Substring Code

Java	`String foo = scanner.nextLine();`
	`int space = foo.indexOf(' ');`
	`int databaseCount = Integer.parseInt(foo.substring(0, space));`
	`int shipCount = Integer.parseInt(foo.substring(space + 1));`

Python	`foo = sys.stdin.readline().rstrip()`
	`space = foo.index(" ")`
	`databaseCount = int(foo[:space])`
	`shipCount = int(foo[space+1:])`

C++	`string foo;`
	`getline(cin, foo);`
	`size_t space = foo.find(' ');`
	`int databaseCount = stoi(foo.substr(0, space));`
	`int shipCount = stoi(foo.substr(space + 1));`

C#	`string foo = Console.ReadLine();`
	`int space = foo.IndexOf(' ');`
	`int databaseCount = Convert.ToInt32(space.substring(0, space));`
	`int shipCount = Convert.ToInt32(space.substring(space + 1));`

Finally, Java and C++ both provide a way to read the numbers and convert them to integers directly from the input. This generally requires reading in a "dummy" string to move the input's cursor beyond the newline character at the end of the line. This is the easiest approach for C++.

Language	Example Direct Read Code

Java	`int databaseCount = scanner.nextInt();`
	`int shipCount = scanner.nextInt();`
	`scanner.nextLine();`

C++	`int databaseCount, shipCount;`
	`cin >> databaseCount >> shipCount;`
	`string trailingNewline;`
	`getline(cin, trailingNewline);`

With the counts read in, we can start reading in the names of the systems in the database. We don't particularly need to do anything with these names, we just need to be able to hold them in memory so we can reference them later. In most languages, we could use an array to store these strings, however there's a more efficient data structure we can use. Furthermore, C++ doesn't support using a variable to define the size of an array; you'd have to directly allocate the memory yourself, which is risky and inconvenient. Instead, let's use a structure known as a List.

Lists are similar to arrays in that they can contain objects in a particular order, and usually allow retrieval of specific items by their index number (also known as 'random access'). We'll go into the benefits of lists in the next section, but for now the more relevant benefits are that they allow us to easily add and remove items, and not pay particular attention to how large the list is or needs to be.

After we declare our list, we should create a for loop to read in a number of lines equal to 'databaseCount'; each of these lines must then be added to the list.

Language	List Population Code

Java	`List<String> database = new ArrayList<>();`
	`for(int i = 0; i < databaseCount; i++){`
	`  database.add(scanner.nextLine());`
	`}`

Python	`database = []`
	`for i in range(databaseCount):`
	`  database.append(sys.stdin.readline())`

C++	`ArrayList database = new ArrayList();`
	`for(int i = 0; i < databaseCount; i++){`
	`  database.Add(Console.ReadLine());`
	`}`

C#	`list<string> database;`
	`for(int i = 0; i < databaseCount; i++){`
	`  string str;`
	`  getline(cin, str);`
	`  database.push_back(str);`
	`}`

With the database fully loaded, we can now start reading the list of systems reported by the ship as being in working order. We don't need to store these strings; in fact, these strings represent those we don't need to store in the database, either. We want to print out the list of systems that the ship *didn't* report; as a result, as we read in each string, we remove that string from the database. Any that remain when we finish reading lines must not have been reported by the ship, and must therefore require further inspection.

Language	List Subtraction Code

Java	`for(int i = 0; i < shipCount; i++){`
	`  database.remove(scanner.nextLine());`
	`}`

Python	`for i in range(shipCount):` `  database.remove(sys.stdin.readline())`

C++	`for(int i = 0; i < shipCount; i++){`
	`  database.Remove(Console.ReadLine());`
	`}`

C#	`for(int i = 0; i < shipCount; i++){`
	`  string str;`
	`  getline(cin, str);`
	`  database.remove(str);`
	`}`

Again, once the loop is completed, 'database' will contain only the names of systems which appeared in the original database list, but did not appear in the second list provided by a ship. These are the systems we need to print out to the console to complete our task. Loop over each element within the 'database' list and print it to the console.

Language	Output Code

Java	`for(String system : database){`
	`  System.out.println(system);`
	`}`

Python	`for system in database:` `  print(system)`

C++	`for(int i = 0; i < database.Count; i++){`
	`  Console.WriteLine(database[i]);`
	`}`

C#	`for(auto it = database.begin(); it != database.end(); ++it){`
	`  cout << *it << '\n';`
	`}`

Additional Background

In this problem, we used lists, one of a number of types of advanced data structures supported by most programming languages. While most of these data structures are supported "behind the scenes" by primitive arrays and memory pointers, they are built in such a way to provide additional functionality, convenience, and flexibility to programmers. Knowing what data structures are available in your language and how they work will be a critical part of solving more advanced Code Quest problems. Let's look into some of the more common types now.

Here, we used a list. As mentioned previously, lists are very similar to arrays, in that they hold a number of data objects in a particular order. Each item is associated with an index number, typically starting at zero, and ending at the size of the list minus one. Unlike an array, however - and like all of the other data structures we'll discuss here - lists usually do not have a fixed length and can grow or shrink as needed to contain more or less data. New items are usually added to the end of a list by default - that is, they're assigned the next available index number - but generally they can also be placed at a specific index number, either replacing an existing value or forcing it and any higher-indexed items to be shifted over to make room.

Queues and stacks are often treated as separate data structures, but they are effectively a specialized version of a list; in fact, any list can be used in a way that simulates either a queue or a stack. Unlike a list, queues and stacks generally don't permit "random access" of their contents; that is, you can't simply retrieve the Nth item in the structure, nor add a new item at a particular position. Instead, queues and stacks both impose a specific ordering when adding and retrieving items. Queues impose a first-in, first-out (FIFO) ordering. Items are added to the back (end) of the queue, and are retrieved from the front (start) of the queue. This way, whichever item was in the queue the longest will always

be the next one to be retrieved. This works similar to a queue of people waiting their turn to buy tickets at a movie theater; each must wait for the person in front to be seen before they can buy their own tickets. Stacks work similar to a stack of plates. Items are both added and removed from the front of the stack, imposing a first-in, last-out (FILO) order. In order to avoid damaging plates in a stack, you can only place new plates on top, and you can only remove the top-most plate from the stack. You may often see the terms "push" and "pop" used in conjunction with stacks, in place of the "add" and "remove" terms respectively used with other data structures.

A set is another type of data structure you may commonly see used. Unlike lists, queues, and stacks, sets usually don't impose any sort of ordering on their contents; you can't retrieve specific items by their index number, and you're not guaranteed to get them in any particular order when iterating over the full set. What makes a set unique is that it enforces uniqueness; a set is guaranteed to contain only unique items. Attempting to add a duplicate item to a set - an item that is equivalent to one already existing in the set - will fail. Usually this is a "silent" failure, meaning that the program won't throw an error when you attempt to add a duplicate, but instead will simply ignore your request. In this case, the 'add' function used to attempt to add the duplicate will usually return 'false' or some other special value to indicate that it had no effect.

The last kind of common data structure you're likely to come across and use is a map (called a dictionary in Python). Unlike the other structures, maps associate one data value (known as a 'key') with a different data value (the 'value'). As in a set, keys must be unique, as they present the only means by which to retrieve the associated values, however values can be duplicated within a map. When adding an item to a map, it must be associated with a key. If that key already exists within the map, the new item will typically overwrite the existing value (which, depending on the language, may then be returned by the 'add' function).

Depending on the programming language you use, you may have access to some of all of these data structures, however they may go by different names than those presented here. Some languages, particularly Java, may even have different implementations of these types of structures. Each will meet the basic definitions of a list or set or whatever, but may provide additional functionality or exhibit slightly different behavior. Before you use a data structure, be sure to review your language's documentation so you know what to expect from it - and more importantly, what *not* to expect.

Further Discussion Topics

- Review your programming language's documentation to identify which kinds of data structures are supported. In what situations might you use each type of structure?
- Consider programs you've previously written. How could you use these data structures to improve those programs?

Problem 04: Special Treatment

Summary of Problem

Students must read a string of text and reprint the string without any special - non-alphanumeric - characters.

Topics Covered

- Special character handling
- SQL/Code Injection

Suggested Approach

As with many problems, there are multiple ways to approach a solution to this particular problem. Most languages provide utility methods that can identify characters that represent letters or numeric digits, or provide more advanced features such as regular expressions that can identify and remove any special characters in one fell swoop. These approaches are just as valid as the one presented here; however, the exact means of using those approaches will vary from language to language. The solution shown here aims to avoid any language-specific details.

To begin, students should read the string of input being provided for the test case. Since each test case consists of a single line of text, we only need to declare one variable to hold that information, and need not worry about looping over multiple lines of text. This portion of the solution is very similar to the provided "Hello World" solutions; rather than printing the input immediately, however, we're saving it to a string variable.

As it turns out, this one variable is the only one we actually need to declare. We need to print out the same string without any special characters; however, we don't actually need to store that information in order to print it. Instead, we can check each character in the string in turn; if it's a letter, number, or space, we print it immediately; otherwise, we ignore it and move on to the next character.

This is possible because strings are actually an array of individual characters. All languages provide a way to access these characters directly by their index, just as you would in an array. Java, for example, provides a `.charAt(x)` method; x is an integer value representing an index point in the string. Python, C++, and C# are even more straightforward, with syntax identical to that used for accessing members of an array; in each of these languages, given a string called 'foo', `foo[2]` will return the third character in that string (the one at index 2; remember that in most languages, indices start at 0).

How do we identify a character as a letter, numeric digit, or space, however? On a computer, everything eventually gets stored as a number. In fact, a character is itself a small integer. Characters can be added, subtracted, multiplied, and divided, and your computer will treat them the same as any other numeric value. Several standards - known as encodings - exist for converting text characters to

numbers, but the one most commonly used for characters you can type off a standard US keyboard is ASCII (American Standard Code for Information Interchange). The reference information provided for Code Quest includes a table showing the numeric values of different characters; for example, a space is equal to the integer 32. This mapping is what we'll use here.

With all that explained, let's begin. We have our string of text, which may include special characters. For this solution, we'll declare a for loop that will iterate over each character in the string (example pseudocode below):

```
for (int i = 0; i < text.length(); i++)
```

During each iteration of this loop, we'll want to grab the character at the current index (the 'i' counter in our for loop). We'll then compare the number against several ranges of numbers:

- If the character's numeric value is 32, it's a space.
- If the character's numeric value is between 48 and 57 inclusive, it's a numeric digit (0-9).
- If the character's numeric value is between 65 and 90 inclusive, it's an uppercase letter (A-Z).
- If the character's numeric value is between 97 and 122 inclusive, it's a lowercase letter (a-z).

If the character's numeric value falls within any of those ranges, we print the character. Otherwise, it's some sort of punctuation - a special character. In that case, we do nothing, simply moving on to the next character in the next iteration of the for loop. Again, example pseudocode is below; you'll need to translate this into your programming language:

```
char chr = text[i];
if(chr == 32 ||
     (chr >= 48 && chr <= 57) ||
     (chr >= 65 && chr <= 90) ||
     (chr >= 97 && chr <= 122)) {
   print(chr);
}
else{
   // ignore; special character
}
```

Once the for loop ends, we need to make sure to print a newline character to keep each test case's output on a separate line.

Additional Background

We often refer to punctuation as "special characters" in programming scenarios because they often serve a special function. A backslash (\) typically means that the next character should be handled differently; \n, for example, usually represents a line feed character. A \" within a double-quoted string - "like \" this \" one" - means that the quotes do not end the string, but is a part of it. When writing software, characters such as / or # are often used to mark comments, which causes the compiler to ignore all remaining text on that line.

Unfortunately, specifically because these special characters have special behavior, they can be especially dangerous if a software developer doesn't pay special attention to them when they're received as input. Consider an application which provides a form users can complete in order to register for an event. Information entered into this form is stored in a database, so that the event organizers can access and manage the information at their leisure. Many databases for this and similar purposes use a specialized language called SQL - Structured Query Language - in order to access and modify information in the database. For example, one of the event organizers might run a query such as this to get the names of everyone registered for the event who provided an email address:

```
SELECT firstname, lastname
FROM attendees
WHERE email IS NOT NULL;
```

This query accesses information in the 'attendees' table, filters out entries in that table which do not have a value defined for 'email', and returns the stored values for 'firstname' and 'lastname' for all remaining records.

SQL can also be used to modify data in the database, or the structure of data. In this case, the event organizers have designed their application to build an INSERT command based on the information their users have entered. Their Java code looks like this:

```
String query = "INSERT INTO attendees ";
query += "(firstname, lastname, email)\nVALUES ('";
query += firstName;
query += "', '";
query += lastName;
query += "', '";
query += email;
query += "');";
```

The event organizers expect that this should produce a String that looks like this:

```
INSERT INTO attendees (firstname, lastname, email)
VALUES ('John', 'Doe', 'johndoe@gmail.com');
```

In most cases, that's exactly what it will do. However, one of their users is up to no good. Rather than entering their actual email address, they enter this instead:

```
foo@foo.com');\nDROP TABLE attendees; --
```

This produces the following string:

```
INSERT INTO attendees (firstname, lastname, email)
VALUES ('John', 'Doe', 'foo@foo.com');
DROP TABLE attendees; -- ');
```

The event organizer's application dutifully runs this as a command on the database... and suddenly the application starts to crash. When the organizers look at the database to figure out what's happened, they find that the attendees table - along with all of the records it held - has been deleted!

This is what's known as a SQL injection attack. By adding a few extra characters to close the intended command earlier than actually intended, the attacker was able to add a second command of their own. In this case, a DROP TABLE command instructs the database to delete the identified table and all of its data. The two dashes at the end of the attacker's input turn any remaining characters into a comment, ensuring that everything - including the malicious command - executes without any difficulty.

A similar vulnerability in a library called Log4J was recently discovered and caused a widespread panic across the internet. Log4J is a Java library that provides logging utilities for major applications. These logs allow those maintaining an application to monitor what's happening at any given point in time and ensure that the system remains secure. Unfortunately, Log4J contained a major security flaw that permitted attackers to run virtually any command on the machines that were running these applications. If an application logged input received from users, and had a certain Log4J setting enabled (which was enabled by default), an attacker could enter a specially formatted string containing commands as input. Log4J would recognize the format of the input string and would attempt to execute it as though it was trusted code. This vulnerability has been used to install ransomware on application servers, and can be used to install and distribute malware to users. While the issue was quickly patched, it affected millions of servers, including some operated by major corporations like Google and Microsoft.

These sort of attacks - SQL injection and code injection - can largely be prevented by sanitizing inputs. If there is no reason for special characters to be entered, they should be removed from the input before it is saved, or rejected before it enters the application proper. Even when special characters are allowed, applications should validate input strings against an expected format; for example, an email address should always contain a single @ symbol somewhere in its middle, and end with a website domain (such as gmail.com or mail.co.uk). By taking care to treat input - and special characters in particular - with a certain degree of suspicion, applications can be easily guarded against a large range of possible cyberattacks.

Further Discussion Topics

- Research the Log4J vulnerability - known as Log4Shell or CVE-2021-44228 - and see how it works, how it was resolved, and what impact it had on affected systems. There are many videos on the internet demonstrating this attack vector.
- Review some of your past programs and see how unexpected inputs could affect them. How could you modify your program to resolve these issues?

Problem 06: Find the Missing Sensor

Summary of Problem

Given a mostly contiguous but unsorted list of integers, students must identify the one integer missing from the list.

Topics Covered

- Sorting
- Unit testing

Suggested Approach

In this problem, we must identify a single missing number from an otherwise complete sequence of numbers. Since the list of numbers we're given is unsorted, we can't simply walk through the list and identify the missing number by finding the point at which the next number increases by two instead of by one. While there are several approaches that could be taken to overcome this issue, most of them boil down to one of two approaches: sorting the list in order to do what we just mentioned, or crossing expected numbers off a list until only one remains. We'll outline both approaches here.

To start with, we need to read in our input. Each test case provides first the expected number of sensors. Once this is read and converted to an integer, we need to read in the following line, which contains the sensor IDs. This list will contain one fewer ID than the expected number given in the first line of input, so if using a for loop to iterate over these values, remember to subtract one from the expected count in order to avoid array indexing errors or segmentation faults.

Once the list of IDs is stored in an array or list, we're ready to begin. Again, we'll be describing two approaches to this problem, so use whichever method seems easier for you to implement.

Sorting elements of a list is a common task in many computer algorithms. Knowing how to sort items is therefore an important skill, but most programming languages already have a relatively efficient sorting function built into their basic libraries. Unless there's a particular reason to use a particular algorithm (such as for better time or memory performance), you should generally use one of these functions whenever you need to sort data.

- Java: The class java.util.Collections provides a number of utility methods for working with Collections, or more broadly, Lists, Sets, and Maps. Among these is .sort(), which accepts a List of items as an argument. The function will re-order items within the List according to their natural order. You can also provide a Comparator as a second argument to impose a custom sorting order.

- Python: Lists provide a .sort() function which will re-order items within the list according to their natural order. This ordering can be reversed by passing in the "reverse=True" argument, or customized by passing in a function that returns a numeric value for the "key" argument.
- C++: The sort() function, within the <algorithm> library, accepts two iterators as arguments. Data elements between the two iterators, including the element pointed to by the first iterator but not the one pointed to by the last iterator, will be swapped within memory according to their natural order. A function can be provided as an optional third argument to customize this ordering.
- C#: Lists and Arrays provide a .Sort() method which will sort items within the Array or List in their natural order. A static Sort() method is also provided within the System namespace for sorting Arrays or Lists. In either case, an implementation of the IComparer interface can be provided as an additional argument to customize the sort order.

When we refer to the "natural order" of items, this is a default ordering method for primitive data types. Numbers will typically be sorted in ascending order, and Strings will be sorted in ascending order according to the numeric value of their first character(s). Certain other data types may have natural orderings of their own, which will be explained in the relevant documentation.

Once our list of IDs is sorted, we can iterate over the list to identify the missing number. The first number in the list should be 1; each subsequent number should be one more than the previous one. When we find a number that's different from the one we were expecting, we can stop, as the number we expected is the missing one. If we reach the end of the list without finding a mismatch, the missing number must be equal to N, the expected number of sensors. This entire approach is outlined in the pseudocode below.

```
Sort(ListOfIds);
Let Expected = 1;
For Each Id In ListOfIds{
  If(Id != Expected){
    Break;
  }
  Expected = Expected + 1;
}
Print Expected;
```

The second approach avoids sorting the original list. Instead, we can populate an array of boolean values, equal in length to the expected number of sensors, with 'false' values (or an integer array with 0 values). We'll call this the 'tracking array.' We then iterate over the provided list of sensor IDs. For each ID, subtract one, then set the value of that index within the tracking array to 'true.' Since each sensor ID is unique, by the time you reach the end of the list, all but one value within the tracking array should be 'true'. Iterate through the tracking array until you find the 'false' value; that index, plus one, is the ID of the missing sensor. This is outlined in the below pseudocode:

```
Let Tracking = Boolean[N];
For I From 0 To N - 1{
  Tracking[I] = False;
```

```
}
For Each Id In ListOfIds{
  Tracking[Id - 1] = True;
}
For I From 0 To N - 1{
  If(Not Tracking[I]){
    Print I + 1;
    Break;
  }
}
```

Additional Background

The sort of diagnostic testing described in the problem description is very important in many professions, including software engineering. As stated in the problem, the earlier a problem is identified, the easier (and often more importantly, cheaper) it is to fix. It's easy to see how this might be the case for a jet fighter like the F-35; replacing or repairing a faulty part in a completed aircraft would require removing any paneling or other parts blocking access to the faulty part. Any damage caused by the faulty part would need to be repaired as well. If the faulty part caused a problem mid-flight, it's possible that the pilot could have been injured or killed, and the aircraft itself destroyed. As a result, Lockheed Martin conducts rigorous testing on all of its aircraft before they're delivered to the military for service, and trains military personnel in how to conduct regular maintenance inspections to identify any damage caused by wear-and-tear before it can present any safety problems.

But how does this apply to software engineering? Updating a buggy program shouldn't be as difficult as repairing an aircraft when there aren't any nuts and bolts to unscrew. While that may usually be the case, errors in software can still prove to be costly. Once a software program is released, it could be downloaded by thousands of people around the world in a matter of days. If a bug is discovered after that release, an update could be released, and all of those users warned to install the update. However, not all of them will do so, and any problems caused by the bug in the meantime may negatively impact the developer's reputation and make it harder to sell future software. Secure applications like those developed by Lockheed Martin could be installed on isolated systems, such as on aircraft carriers or submarines, or embedded into flight control modules in aircraft. In these cases, there may be actual physical nuts and bolts that have to be unscrewed in order to access a USB port to upload the patched software, and simply delivering a hard drive with the patch can be a major headache on its own when a warship might not return to a naval base for weeks at a time.

It's therefore important, in software development as in everything else, to identify and address problems as soon as possible, ideally before they actually become problems. This is often handled in software development through the use of unit testing. A unit test is a programmed function that can be configured to run and test a small portion of a software application on its own, usually as part of a build process. Whenever a developer saves a change to the application's code, a server will compile the application, then run a series of unit tests (known as a "test suite") against the new version. The

results of the tests can identify if there are any new problems (called "regressions") introduced by the developer's change.

Unit tests typically focus on small details of a larger process; this can help to identify the exact location of a bug and how best to fix it. For example, when you use a web browser to access a website, all you do is enter the website's URL and press Enter; however, the web browser goes through a series of steps:

- It builds an HTTP request to ask the website for information
- If the website is secured using HTTPS, it will encrypt the request
- The browser then sends the request to the website and waits for a reply
- It may be redirected to another address (for example, a bit.ly address will usually point to another website)
- It may encounter an error, and be required to display a standard error screen
- If the website responds correctly, the browser needs to display the website to the user

Each of these bullet points are smaller tasks within that process that could be the subject of a unit test. In each test, the test framework would call the function in the application code responsible for performing that task, then examine its results to ensure they happened as expected.

Let's look at a simple case; a calculator application written in Java. A common unit testing framework used with Java is JUnit; you can read more about it at their website, https://junit.org. JUnit uses a series of annotations and static functions to allow developers to create and configure unit tests. In this scenario, we'll create a few unit tests to assess our calculator's exponent() function, shown below:

```
/**
 * Raises the given base to the given exponential power
 * @param base the base value, a double
 * @param power the exponential power, an integer greater than or equal to 1
 * @return the result of base^power
 */
public double exponent(double base, int power){
  double result = base;
  for(int i = 1; i < power; i++){
    result *= base;
  }
  return result;
}
```

Given a base value and an exponent greater than or equal to 1, the function is expected to return a value equal to the base raised to the power of the given exponent. Let's write a JUnit test to confirm that it behaves as expected:

```
@Test
public void testExponentPowerOfTwo(){
  assertEquals(100.0, exponent(10.0, 2));
  assertEquals(4.0, exponent(2, 2));
}
```

The @Test annotation is from JUnit; JUnit will scan through all classes in your classpath in search of methods marked with this annotation, then run them as individual tests. The assertEquals() method is also from JUnit; it confirms that the expected value (the first argument) equals the test value (the second argument). In this case, we're testing the exponent() function twice; once to calculate 10.0 squared (which we expect to return 100.0) and again to calculate 2 squared (which we expect to return 4.0). If exponent returns a different value for either case, the test will stop, and JUnit will report that the test failed. In this case, the test will pass; exponent() will return the expected value in both cases. However, our exponent() function may not always work as expected.

The test above is an example of a "happy path" test. We're providing our code with expected arguments that we know should work. This will help us to identify bugs that may occur in everyday use of the function, but doesn't prove that the function works correctly - or at least, fails gracefully - with *unexpected* arguments. The documentation says that we expect the 'power' argument to be greater than or equal to 1, but what happens if we pass in 0 for that argument? exponent() should return 1.0... shouldn't it?

```
@Test
public void testExponentPowerOfZero(){
   assertEquals(1.0, exponent(10.0, 0));
   assertEquals(1.0, exponent(2, 0));
}
```

```
java.lang.AssertionError: expected <1.0> but was <10.0>
```

Oh no! Our test failed. JUnit saw that exponent() returned a value of 10.0 for the first call, and immediately threw an exception and failed the test. Our second call, using 2 as the first argument, never got executed. Why did this occur? If you look back at our function, we did state in the documentation that 'power' should be greater than or equal to 1, but we don't have anything in the code to enforce that restriction or properly calculate other exponents. Since 'result' is initially set to 'base', and we only increase its value from there, 'base' will always be the minimum value returned. Let's try to address this:

```
/**
 * Raises the given base to the given exponential power
 * @param base the base value, a double
 * @param power the exponential power, an integer greater than or equal to 1
 * @return the result of base^power
 * @throws IllegalArgumentException if power is 0 or less
 */
public double exponent(double base, int power) throws IllegalArgumentException{
   if(power < 1){
      throw new IllegalArgumentException("Power must be >= 1");
   }
   double result = base;
   for(int i = 1; i < power; i++){
      result *= base;
   }
   return result;
```

```
}
```

Now exponent() will throw an exception if we give it an unexpected value for 'power.' This will require that we change our test as well:

```
@Test
public void testExponentPowerOfZero(){
  try{
    exponent(10.0, 0);
    fail("No exception thrown!");
  }
  catch(IllegalArgumentException e){
    assertEquals("Power must be >= 1", e.getMessage());
  }
  catch(Throwable e){
    fail("Wrong type of exception thrown: " + e.getClass());
  }
}
```

Now, JUnit will attempt to call exponent(), with the illegal '0' for its second argument, without bothering to check its return value; however, if it does successfully return, our test will fail. JUnit's fail() method always throws an AssertionError exception, with any given string as its message. If an exception is thrown, we'll catch it. If the exception is an IllegalArgumentException - as we expect - JUnit checks the exception's message to ensure it is correct. Any other exception will be caught by the "catch(Throwable e)" block, and will result in another test failure. This test helps us to confirm that our function fails properly; if a user gives a bad argument, it doesn't give them a bad result in response, but will throw an error that explains the problem. Properly handling bad input is just as important as properly handling good input, and unit testing is important to verify both scenarios.

Further Discussion Topics

- Identify a unit testing framework for your preferred programming language and review its documentation.
- Write some unit tests for other programs you've written, either for class or for Code Quest. What is the "happy path" for your code? How should your program handle an "unhappy path?"

Problem 08: DNA Storage

Summary of Problem

Students must read a string representing the base pairs in a DNA molecule, interpret that information as a binary sequence, then convert it to a text string.

Topics Covered

- ASCII-binary conversions
- Moore's Law

Suggested Approach

The table shown in the problem description provides a quick visual representation of how to manage this problem. This walkthrough will break the problem down into four main steps:

1. Divide the input string into seven-character segments
2. Convert each segment into binary
3. Convert the binary number into a decimal number
4. Convert the decimal number into a character and print it

The number of characters in the resulting string can be determined by finding the length of the input string and dividing by seven. The length of the input is guaranteed to be evenly divisible by seven, and so this will always return an accurate result. We'll use this number to declare an array or list to contain a series of strings, which we'll then populate by breaking the input string into seven-character-long substrings.

```
let outputLength = input.length() / 7;
let substringArray = new String[outputLength];
for(let x = 0; x < outputLength; x++){
  substringArray[x] = input.substring(x * 7, (x + 1) * 7);
}
```

In the above pseudocode, we're using `x * 7` and `(x + 1) * 7` as the arguments for substring; these represent the start index (inclusive) and the end index (exclusive) of the substring we want to extract from `input`. As a result, our first substring will cover from index 0 up to 7, the second from 7 up to 14, and so on. The substring functions for some languages work differently (for example, some accept a start index and a length); make sure to double-check the documentation for your language and ensure that you're passing in the correct arguments.

With our substrings obtained, we can now work on each one in turn to convert it into a single character. We'll do this in three stages, as outlined above. First, the DNA sequence must be converted into a binary number. Each character in the substring should be read in turn and used to build a new

string containing only 1's and 0's. Any A or T characters represent a 0; any C or G characters represent a 1.

```
for(let x = 0; x < outputLength; x++){
  let substring = substringArray[x];
  let binaryString = “”;

  for(let n = 0; n < 7; n++){
    if(substring[n] == ‘A’ || substring[n] == ‘T’){
      binaryString += ‘0’;
    }
    else{
      binaryString += ‘1’;
    }
  }

  // ...
```

Now we need to convert this into a decimal number that we can actually process. Many languages provide functions that can convert a binary string into a decimal number; for example, in Java, `Integer.parseInt(binaryString, 2)` would work for this purpose. The '2' argument tells Java to interpret the string as a base-2 (binary) number. If you're not familiar with such a function, the conversion can be done manually as well.

Declare an integer, initialized to 0, that will receive the converted value; we'll call it `result`. Then declare a second integer, which we'll call `powerOfTwo`, that should be initialized to 2^6, or 64. (Where's the 6 come from? The length of our binary string, 7, minus 1.) We'll then use a for loop to step over each bit in the binary string, from left to right. If a bit is 1, we add the current value of `powerOfTwo` to `result`. If the bit is a 0, we leave `result` unchanged. Either way, we then divide `powerOfTwo` by two. When the for loop is complete, `result` will contain the decimal equivalent of the binary number.

```
  // ...
  let result = 0;
  let powerOfTwo = 64; // 2^(n-1)
  for(let n = 0; n < 7; n++){
    if(binaryString[n] == ‘1’){
      result += powerOfTwo;
    }
    powerOfTwo /= 2;
  }

  // ...
```

With that done, we can finally convert the number to a character. This is the easiest part of all; since all characters are numbers, we simply need to cast the number we just calculated to a 'char' value. We can then print the character directly to the console, and continue with processing the next substring. Once all substrings are done, make sure to print a newline character to the console so the output from each test case remains separate.

```
  // ...
  let printable = (char) result;
  print(printable);
} // end for x
print('\n');
```

Additional Background

The capabilities of computers have improved dramatically since the invention of the integrated circuit (or microchip). Former IBM CEO Gordon Moore observed as early as 1965 that the amount of computing power available in the average microchip was doubling about every year; a decade later, he revised his prediction to doubling every two years, but that rate of progress has largely held since then. This observation has since become known as Moore's Law, and has served as a model for the computer engineering industry to manage expectations and goals.

This can be best illustrated by looking at the devices we've used to store data over the years. As computers become better at processing more data more quickly, we need to be able to store all that data somewhere. As we've generated more and more data and created more powerful computers, we've needed more efficient ways to store that data, both in terms of how quickly it can be read off the storage device, but also in terms of how much data can be stored in a given amount of physical space.

To be able to really make sense of this, though, let's look at how we measure data. We referenced 'bits' in the program walkthrough before. A bit is the smallest unit of data possible; a single binary digit, either 1 or 0. A Boolean value can be stored in a single bit (with 1 representing true and 0 representing false). All other types of data require one or more bytes. A byte is eight bits in length and represents a number between 0 and 256 (or between -128 and 127, if it's a signed integer). Obviously, this isn't very much, so - as with other units of measurement - we typically use SI prefixes to define larger units, as shown in the table below. You might have previously seen these units defined using a power of 2 - for example, 1,024 bytes to a kilobyte - but recently these have been redefined to use powers of 10 to align with the actual SI definitions.

Unit	Equal To	What can be stored with this much data?
Byte (B)	8 bits	A single text character
Kilobyte (KB)	1000 bytes = 8000 bits	Most Code Quest input files are less than 10 kilobytes
Megabyte (MB)	1000 kilobytes = 8000000 bits	High-quality photos are usually several megabytes
Gigabyte (GB)	1000 megabytes = 8 billion bits	Software applications, especially video games, are usually measured in gigabytes.
Terabyte (TB)	1000 gigabytes = 8 trillion bits	The largest commercially available hard drives are capable of holding a few terabytes.

Unit	Equal To	What can be stored with this much data?
Petabyte (PB)	1000 terabytes = 8 quadrillion bits	All imagery available in Google Earth - satellite photos, Street View images, etc. - is roughly equal to 20 petabytes
Exabyte (EB)	1000 petabytes = $8 * 10^{18}$ bits	The total amount of digital data stored worldwide in 2007 was estimated to be around 281 exabytes.

Now that we have a foundation from which to judge the relative size of different amounts of data, let's look at how we've stored this data over the years. In 1928, German engineers developed magnetic tape. Originally used for storing and replaying sound recordings, it was first used to store computer data in 1951 with the UNIVAC I, the first digital computer intended for business use. The tape is a long strip of plastic, which is coated with a magnetic material on one side. By being exposed to a magnetic field, a section of the tape could be magnetized to represent a 1 bit; unmagnetized portions represented 0 bits. The amount of data that can be stored on a magnetic tape is dependent on how precise the device reading the tape is; in fact, some tape drives are still used today, and can store dozens of terabytes of data. However, accessing data from a tape can be difficult, as data can only be read sequentially. In order to read data stored at the very beginning or very end of the tape, you have to (un)wind the entire length of the tape to reach that point. Larger-capacity tape drives are also extremely heavy and highly subject to decay. The plastic used as the basis for the tape will often break down after a decade or two, and exposing a tape to a magnet could destroy all of the data it contains.

Despite these drawbacks, magnetic storage systems continue to be used today; however, because of those drawbacks, magnetic tapes are typically only used for archival purposes. For everyday use, other storage systems have been used throughout the years. "Floppy disks," first developed in the 1960's, utilized a disc of magnetic film rather than a tape; this allowed for rapid access to data stored anywhere on the disk, as the drive could spin the disk to the relevant portion and read it directly. The smaller size also made them more portable and easier to use than magnetic tapes, however the flimsy material made them somewhat susceptible to damage (hence the term "floppy"). By the end of the 1980's, floppy disks had shrunk in size from 8 inches wide to just 3.5 inches and gained a hard casing, which helped with this. Unfortunately, the capacity of a floppy disk was severely limited; most floppy disks could only hold a few megabytes of data; many modern digital photographs would be unable to fit on a single disk. Today, floppy disks are no longer supported by most off-the-shelf computer systems, and they live on only as the icon for "Save" buttons in many applications.

CD-ROMs (compact disk, read-only memory) were the clear successor to floppy disks. Rather than using magnetism to store data, CDs used an optical system; lasers would be fired into the reflective surface of the disk. One layer of the disk would be pitted with a series of small grooves; the groove would allow the laser beam to penetrate further into the disk, preventing it from being reflected properly. The drive could then interpret the pattern of intensity of the reflected beam's light as binary data. As with a floppy disk, a CD allowed random access, as the drive could read any portion of the disk on demand as it spun. Despite not being much larger than a floppy disk, the narrow laser beam

was able to read data with a far greater degree of precision than a magnetic sensor, and so could contain much more data. A high-capacity CD-ROM can store over 900 MB of data. As the technology was developed further, different lasers were used to increase the data density even further; a DVD (digital video disk) could hold up to 8.5 GB on a single side; Blu-ray disks can hold up to 128 GB. While still in use, optical disks did have a major drawback; in most cases, once data was written to a disk, it could not be *un*written (hence the ROM in CD-ROM). Rewritable optical disks were created, using a special layer that could be re-melted to replace data, but these required very precise hardware to both read and write, and had limited capacity compared to other technologies.

Solid-state drives have been used for decades in portable USB drives, and are now commonly found as the primary hard drive in high-end desktop computers and laptop computers. These drives use transistors, of the same sort found in microchips, to store data using only an electrical charge. Since any portion of a solid state drive can be accessed directly, without having to wait for a disk or tape to move into position, these drives provide read/write speeds that are orders of magnitude faster than magnetic or optical drives. The lack of moving parts also makes these drives somewhat less prone to mechanical failure and makes them much lighter in weight. Unfortunately, they are more difficult to produce and thus far more expensive than traditional drives, and their capacities are limited by the model presented by Moore's Law. However, that capacity is still exceptional compared to other methods of data storage; some formats of MicroSD card are able to store terabytes of data, and measure just 1.65 square centimeters (about a quarter of a square inch) in size.

So why is DNA being investigated as a new storage medium? DNA represents one of the most dense data storage mediums known to mankind. In 2019, researchers were able to successfully encode 16 GB of data - the entire text of the English Wikipedia - into a single DNA molecule. Two years prior to that, another team reported they had achieved a data density of 215 petabytes of data per gram of DNA. By comparison, a MicroSDXC card offers the highest commercially available data density, about 1 terabyte of data per gram; 215,000 times less than that of DNA. Unfortunately, the reason your computer isn't using a biological drive right now is because it's very difficult and expensive to read and write that much data from DNA. In a few more decades, however, we may be using DNA-driven hard drives, and Moore's Law could become a thing of the past.

Further Discussion Topics

- How much data do you think you process every day? You can get estimates from the settings menu in a cell phone, checking logs of your internet modem or router, and by estimating the amount of TV you watch, books and other text you read, and music you hear.
- Visualize the effects of Moore's Law by finding older data storage devices, such as cassette tapes, floppy disks, CDs, or older hard drives. Estimate how many of the older devices would be required to have the same storage capacity as your current computer's hard drive.

Problem 10: Emergency Update

Summary of Problem

Students must read the contents of two lists of emergency contact information, then identify the items which were added, deleted, or updated within the second list.

Topics Covered

- Data storage strategies
- Version control

Suggested Approach

In order to be able to compare the contents of the first list against the second, we must be able to store the contents of at least the first list. There are several viable strategies to doing this; we'll cover a few options shortly. Once the first list is fully stored, we can begin reading in the contents of the second list. With each new record, we'll want to split apart the individual data fields in the new record for comparison, then search for a corresponding record in the stored (old) data. If no match is found, the new record must have been created; we create a string to be printed later indicating this fact. If a match is found, the individual fields can be compared for differences; any differences are likewise reported. When the comparison is complete, the old record should be marked as having been fully processed in some way. Once all of the new records have been processed, any remaining old records that haven't been so marked must have been deleted. We add them to the list to be printed as output.

Finally, the list of output messages should be sorted alphabetically; since each output message starts with the person's name, it makes more sense to sort the output messages - which theoretically should be shorter than either list of data - than attempt to sort the entire list of data itself. The output messages can then be printed.

The largest portion of this problem, then, is storing the data. Students may be tempted to immediately break the input lines into their separate data fields. While this isn't by any means a bad idea, they must use caution in storing this data, to ensure that each field remains associated with the same person. As we mentioned above, there are several ways to do this.

First, the simplest approach would be to store the lines as they are given, without breaking apart the individual fields first. This simply requires declaring an array or other collection of strings with sufficient capacity to hold the entire list (as indicated by the first line of input for each test case) and filling that array with the input strings. When searching for records for comparisons, we can identify the target records by iterating through the array until we find a string that starts with the name given in the new record. If such a match is found, we can break apart the data fields at that time. Processed records can be marked by simply replacing them with null or empty strings within the array.

This works primarily because we know that all of the data we receive will be following the same format, will be valid, and doesn't contain many fields. Such an approach wouldn't be advisable in an actual software application, for many reasons. Among these is that you may not necessarily *want* a full data record. Say our data also included work locations; an HR representative may want to identify how many employees work at each location. In this case, they don't need to access the full record for each employee; they only need the location.

A more robust approach would be to break apart each record into the individual data fields, store those fields in an object designed for the purpose, then store the object in an array. This requires a bit more work up-front - the structure of the object must be declared, and functions created to allow easy retrieval of that data - but it's a better design that can easily be expanded upon for future updates. For example, rather than deleting objects that have been found, a boolean flag can be added to an object to indicate if it has been identified in the list of new records. This approach is similar to how relational database systems work; by separating the data fields, you gain additional control over how (or if!) they can be accessed and modified.

Of course, for this particular scenario, this level of design is unnecessary. The pseudocode below outlines the general algorithm for the first approach, using a simple array for the old data and processing it only as needed:

```
let counts = input.readLine().split(“ “);
let oldCount = counts[0] as int;
let newCount = counts[1] as int;
let oldData = new string[oldCount];
for(let i = 0 to oldCount){
  oldData[i] = input.readLine();
}
let outputMessages = new string[newCount + oldCount];
let outputIndex = 0;
for(let i = 0 to newCount){
  let newRecord = input.readLine().split(“,”);
  let oldRecord = null;
  for(let j = 0 to oldCount){
    if(oldData [j].startsWith(newRecord[0])){
      oldRecord = oldData[j].split(“,”);
      oldData[j] = null;
}
    if(oldRecord){
      if(oldRecord[1] != newRecord[1] && oldRecord[2] != oldRecord[2]){
        outputMessages[outputIndex++] = newRecord[i] + “ UPDATED BOTH\n”;
      }
      else if(oldRecord[1] != newRecord[1]){
        outputMessages[outputIndex++] =
             newRecord[i] + “ UPDATED PHONE NUMBER\n”;
      }
      else if(oldRecord[1] != newRecord[1]){
        outputMessages[outputIndex++] =
             newRecord[i] + “ UPDATED ADDRESS\n”;
```

```
        }
      }
      else{
        outputMessages[outputIndex++] = newRecord[i] + " CREATED\n";
      }
    }
}
for(let i = 0 to oldCount){
  if(oldData[i]){
    outputMessages[outputIndex++] = oldData[i].split(",")[0] + " DELETED\n";
  }
}
sort(outputMessages);
for(let i = 0 to outputIndex){
  print outputMessages[i];
}
```

Additional Background

Identifying how data changes over time is a critical task in many fields. Climate scientists examine records of global temperatures to track how the world around us is changing. Athletes take note of changes in their own performance to better improve their training regimens. In software engineering, tracking how our software changes over time is known as "version control," and is vitally important to maintaining secure applications that perform as desired.

Professional software isn't written in one fell swoop; it takes months of work by a team of people to produce even a relatively simple application. In order to be able to work together - and to avoid accidentally losing any code they're already written - members of the development team will regularly "check in" their code to a central, shared repository. This repository allows each member of the team to access and review each other's code. It also manages how to combine changes made by different people at different times.

These version control repositories typically manage multiple copies of an application's code, called "branches." A "master" branch contains the current version of the application. Many teams will go to great lengths to ensure that the code held within the master branch is fully functional and relatively free of errors; most commonly, this is done by forbidding team members from directly modifying any code on the master branch. Instead, when a team member wants to make a change to the application, they create a new branch by copying all of the code on the master branch. They can then work on this new branch with some degree of freedom; if mistakes are made, they won't break the application as a whole, or negatively impact other team members (who will be working on their own branches).

Once a team member's work is complete, teams will usually review their colleague's work by reviewing the changes made to the branch. How do they know what changes have been made, however? Applications can contain thousands of lines of code, and picking out individual changes would be like looking for a needle in a haystack. Once again, the version control repository can help here. By employing a process very similar to what we just did in solving this problem, the repository

will compare each file in both the team member's branch and the master branch, creating a list of any and all changes known as a "diff" or "patch."

A diff file shows all lines that have been added or removed from the base revision, as well as lines that have changed. It also provides a small amount of context to help with identifying where in a file these changes have taken place. For example, take a look at this diff file:

```
diff -r /home/user/ProbA.java /home/user/HelloWorld.java
17c17
< public class ProbA {
---
> public class HelloWorld {
38,41c38,40
<    // First, connect to the standard input channel
<    // This "try-with-resources" block will automatically close
<    // the Scanner when we're done using it (even if your program
<    // throws an Exception).
---
>    // First, connect to the standard input channel
>    // This "try-with-resources" block will automatically close
>    // the Scanner when we're done using it.
52c51,52
<         System.out.println(input.nextLine());
---
>         String line = input.nextLine();
>         System.out.println(line);
```

This shows a series of changes made to an example Java solution to our practice "Hello World" problem. The line at the top of the file, starting with 'diff -r', shows the Linux command used to generate the file. The first file listed should always be the original file; the second file is the newer version. Typically with version control repositories, this would be used to compare two branches (or at least two versions of the same file, from different branches), but here it seems someone made a copy of ProbA.java and renamed it to HelloWorld.java.

Each change in the file is preceded by a set of numbers. These correspond to line numbers in the two files; the first change shows '17c17', indicating that line 17 in the original file has been updated and corresponds to line 17 in the new file. The second change shows a range of lines on each side; lines 38-41 in the original file, and 38-40 in the new file. Since the second range is shorter, this indicates that a line has been deleted. We can see that another line was added in the third change; line 52 in the original file now corresponds to lines 51 and 52 in the new version.

Following these line numbers are details of the actual change. Lines starting with a < character indicate the contents of the given lines in the original file. This is followed by a line with three dashes, and a set of lines starting with a > character. In order to apply the changes, the first set of lines (as indicated by the line numbers and confirmed by the provided content) should be replaced - in entirety - with the second set of lines. Including the copy of the original file's lines serves two purposes; it allows us to look at the diff and easily see what changed, and it also serves as a form of validation.

If, when attempting to apply these changes, the lines in the file being modified do *not* match the ones in the diff, that indicates that someone else has edited the same section of the file. This creates an "edit conflict" which must be resolved first. Whoever is attempting to make these changes must stop what they're doing and generate a new diff and compare their changes against those made by another person. This prevents someone from accidentally undoing someone else's work. Version control repositories can handle some conflicts automatically, but will almost always ask for confirmation that what they're doing is correct.

Further Discussion Topics

- What are some other advantages of having a version control system? Why else might it be useful to know what changes have been made at what times?
- Create an account at Github and start a new project for your Code Quest solutions. Github uses git, a version control system, to manage files. Experiment there to see what features it offers and how version control works in practice.

Problem 12: Morse Code

Summary of Problem

Given a customized version of Morse Code, students must encode a plaintext message and decode an encoded one.

Topics Covered

- Substitution ciphers
- Huffman coding

Suggested Approach

To begin, students should read in the implementation of Morse Code provided for the test case. Since each test case will provide a different version of the encoding, this must be processed each time. Programming languages that support maps or dictionaries can store the entire code in one of those structures. For example, in Python:

```
englishToMorse = {}
morseToEnglish = {}
for idx in range(26):
  mapping = sys.stdin.readline().split(“ “)
  englishToMorse[mapping[0]] = mapping[1]
  morseToEnglish[mapping[1]] = mapping[0]
```

As each line is read in, it is split at the space separating the English letter from the Morse encoding. These are then placed within the dictionaries; for ease with both encoding and decoding, here we create two dictionaries, one for each direction.

Since the code is guaranteed to be presented in alphabetical order, however, an advanced data structure like this is not necessary. By declaring an array of strings capable of holding 26 items, the encodings can be stored in alphabetical order. For example, in C++:

```
std::string encodings[26];
for (int i = 0; i < 26; i++){
  std::string line;
  std::string encoding;
  getline(cin, line);
  line.copy(encoding, line.length() - 2, 2); // remove the letter and space
  encodings[i] = encoding;
}
```

Once populated, the array will contain all of the encodings in order from A through Z.

To perform the encoding, the characters in the plaintext string should be processed one at a time. If the character is not the first character being read in the line, three spaces should be printed first; this creates the required spacing between letters. Once that is done, if the character is a space, four more

spaces should be printed to the output, thereby creating the larger gap of seven spaces between words. Otherwise, the character must be a letter that requires encoding. If a dictionary or map was used to hold the encodings, then we simply need to look up that letter within the data structure and print the associated value.

If an array was used, we have to determine which index to retrieve and print. Since individual characters are stored as numbers, we can perform numeric operations on them. In this case, we know that the array contains the encodings for the letters A through Z, such that the index for A's encoding is 0, B's encoding is 1, and so on. As a result, we can simply calculate the index by subtracting 'A' from the character we're encoding:

- 'A' - 'A' = 0
- 'B' - 'A' = 1

  ...
- 'Z' - 'A' = 25

With this index, we can quickly retrieve the appropriate encoding from the array and print it to the output.

The decoding can be conducted in much the same way. Split the encoded string at each substring of three spaces to separate individual letters. When this is done, individual decoded spaces will be represented as a string containing four spaces; letters will be any string which starts with a period or dash. If using dictionaries, the corresponding letter can be easily retrieved. If using arrays, we will have to search through the array until a matching string is found. We can then add the index number of that string to 'A' to calculate the corresponding letter; the inverse of the calculation performed above. Each letter (or space) should be printed to the output as it is identified.

Additional Background

As mentioned in the problem description, Morse Code was initially developed for use with the telegraph. Unlike many codes, it was never intended to obscure or hide information; it was developed simply to enable communication by using a simple electrical current. Early telegraphs worked by sending pulses of electric current along a wire. A receiver on the opposite end contained an electromagnet, which would generate a magnetic field in response to a current pulse. This, in turn, would cause a visual or audio effect that would be noticed by the receiving person. Since this system was binary - the only signals it could send were "on" and "off" - both sender and receiver had to agree on a system to convert patterns of "on" and "off" to meaningful text.

Artist Samuel Morse created the first standardized system for telegraphs, hence the name "Morse Code." However, there have been several different versions of Morse Code throughout history. Morse's own original design provided only numerals; telegraphers would have to consult with a code book to determine which word was represented by the number that had been transmitted. Obviously this system was very limiting, and so engineer Alfred Vail (who had originally worked alongside Morse and

physicist Joseph Henry to develop the electrical telegraph system) expanded the code. He assigned each letter - and a series of special characters - its own series of dots and dashes. This made the code much more flexible and useful.

Vail's assignments we're simply random, however; he chose them quite deliberately. One of the keys to ensuring a new system gets widely adopted is convenience. Transmitting messages over a telegraph took time; there had to be clear separation between each dot and dash, to avoid confusing a sequence of dots with a single long dash. As a result, letters with lots of dots and especially lots of dashes took longer to send, and thus were less convenient.

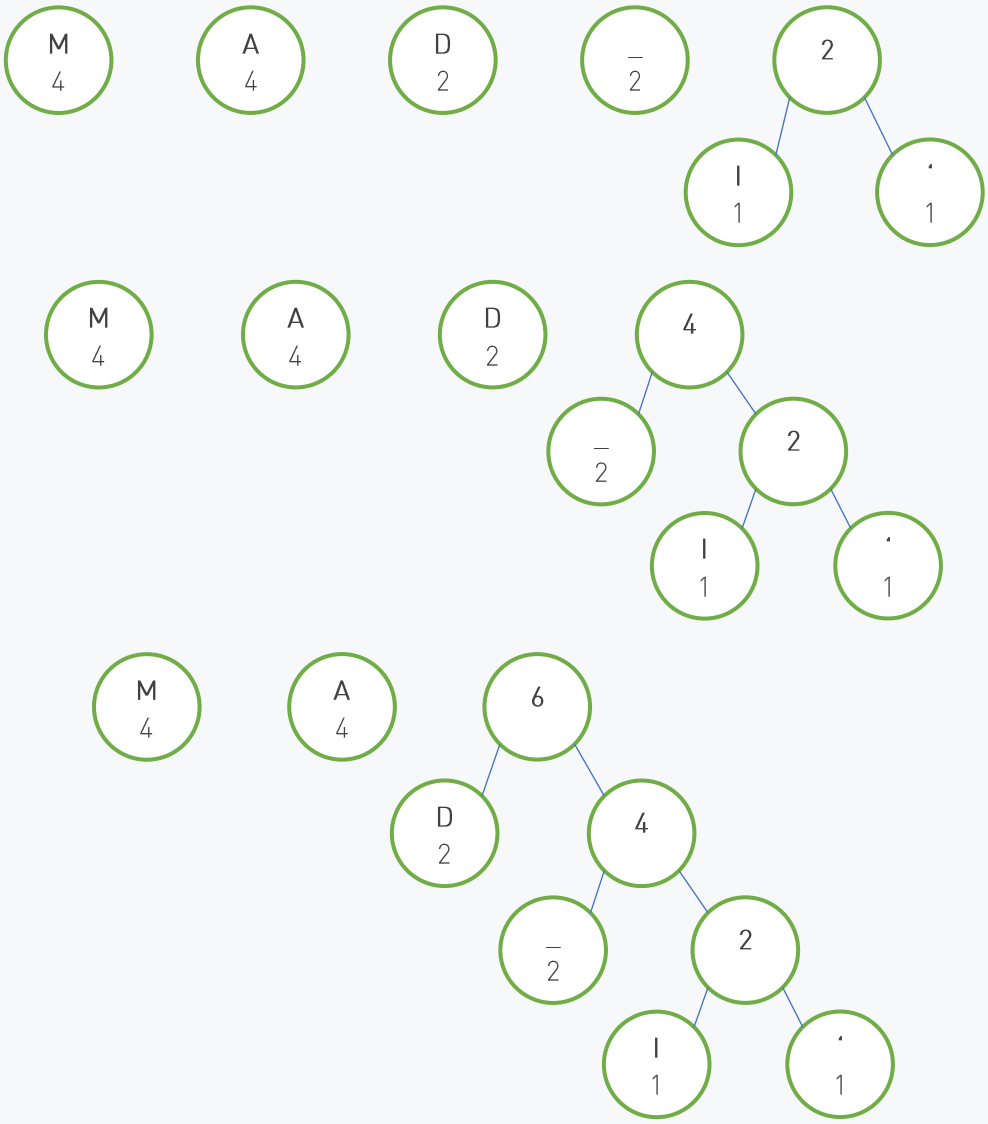
To address this, Vail visited a local newspaper office and made a count of the number of dies for each letter in the office's movable type printing press. He assumed that the more frequently a letter was used in the English language, the more the newspaper would need to print that letter, and the more dies it would need in order to do so. He used this frequency information to inform his assignments of Morse encodings to letters; more common letters received shorter assignments; E, the most common letter, was represented with a single dot, and A, I, and O each had two signal encodings. Uncommon letters received longer encodings; J required a total of two dashes and two dots, requiring over ten times longer to transmit than a single E.

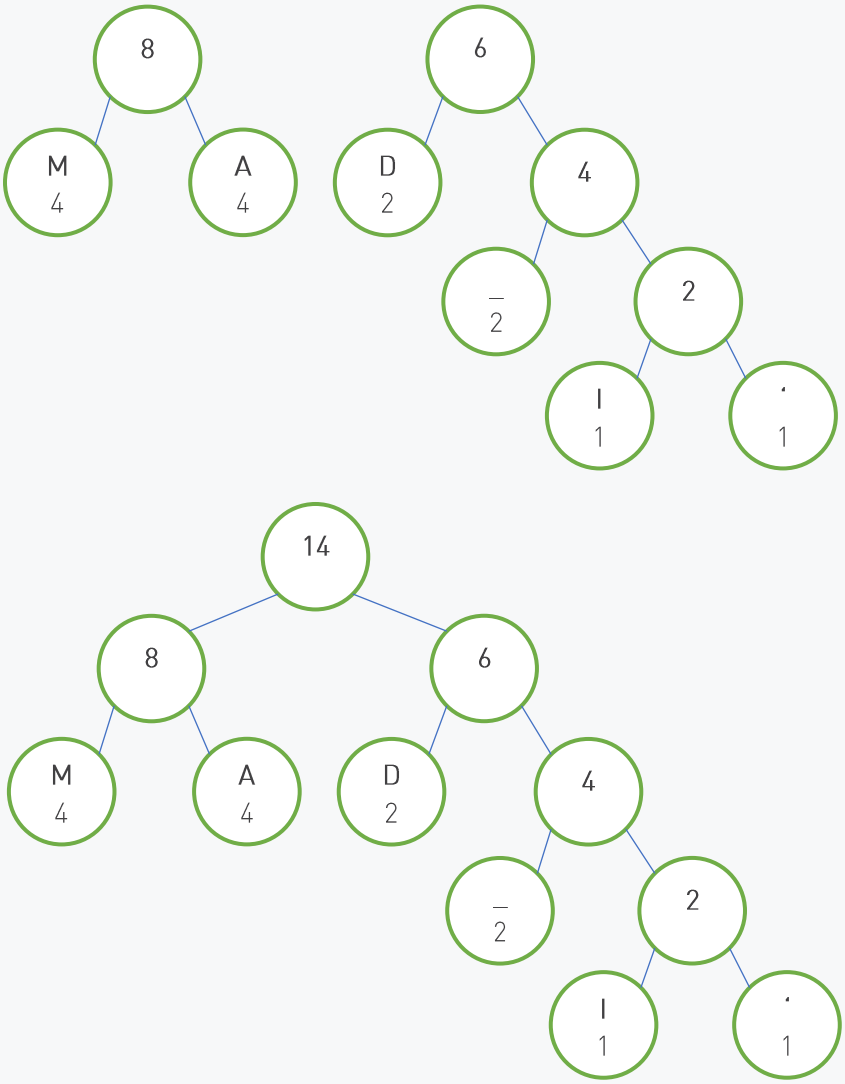
What Vail had essentially done was invented a way of compressing information. Consider a scenario in which each letter required the same amount of time to transmit; let's say an equivalent of 10 dots. Sending a simple greeting such as 'hello' would then take a total of 50 dots to transmit (not counting the 3-dot pauses between letters). However, 'hello' is comprised of very common English letters. With Vail's encodings, 'h' could be sent in the time of seven dots (four dots, plus a dot-long pause between each). As noted above, 'e' is a single dot, and 'o' was a dot-dash. 'l' required a single dash to transmit. With these encodings, 'hello' could be sent within 18 dots (again not counting pauses between letters) - a significant improvement.

This manner of compressing information would go largely ignored until the advent of the computer; however, in 1952, David Huffman published a paper that presented an algorithm for lossless data compression. Huffman's algorithm essentially presented a way of automatically doing what Vail had done by hand over a century earlier. By constructing a binary tree based upon the frequency of individual symbols in a message, each of those symbols can be assigned a binary code that identifies that symbol. More frequently used symbols will appear higher in the tree and thus will have shorter binary codes.

Consider the phrase "MADAM_I'M_ADAM" - the phrase contains 14 characters. Using the binary ASCII values provided in the Code Quest reference materials, this could be represented with 98 bits of data. Let's see if we can improve upon that. Huffman's algorithm associates each symbol with its frequency (or probability of frequency). For our sample phrase, the frequency of each symbol is as shown:

Huffman's algorithm runs recursively until all symbols have been added to a tree. In each round, the nodes with the smallest frequencies are added as children of a new parent node. The parent node's frequency is the sum of those of its children. Any ties that occur when identifying the smallest frequency can be broken in any consistent manner. So, our tree builds itself from the nodes above like so:





Now that our tree is complete, we can use it to determine the encodings for each character in our original phrase. A left branch in the tree represents a 0 bit, and the right branch a 1 bit; the code used for each character is created by following the path from the root node to that character and noting the bits as you go. As a result, 'M' is encoded as 00, 'A' as 01, 'I' as 1110, and so on. This allows us to encode the message in just 34 bits; just over a third of what was required using normal ASCII encodings. As long as we ensure that our recipient knows the same encodings, they can use that information to restore the original message.

Further Discussion Topics

- Investigate other compression algorithms and compare them to Huffman's. What are the advantages and disadvantages of each?
- Use Huffman's algorithm to create an encoding table for the English alphabet, using the relative frequencies of each letter shown on the next page. Use this information to create your own optimized version of Morse Code, exchanging 0's and 1's for dots and dashes.

Letter	Frequency	Letter	Frequency	Letter	Frequency
A	8.2%	J	0.15%	S	6.3%
B	1.5%	K	0.77%	T	9.1%
C	2.8%	L	4.0%	U	2.8%
D	4.3%	M	2.4%	V	0.98%
E	13%	N	6.7%	W	2.4%
F	2.2%	O	7.5%	X	0.15%
G	2.0%	P	1.9%	Y	2.0%
H	6.1%	Q	0.095%	Z	0.074%
I	7.0%	R	6.0%		

Problem 15: Codebreaker Returned

Summary of Problem

Students must perform a digraph and trigraph frequency analysis on a given section of text.

Topics Covered

- Frequency Analysis
- Cryptanalysis of simple ciphers

Suggested Approach

If you've attempted 2021's problem "Codebreaker," the solution for this problem will be very similar; that problem also conducted a frequency analysis, but only on individual letters. If not, we'll point out the points during this walkthrough that you would need to modify to solve that problem.

As always, our first step is to process the input. After reading in the number of lines and converting that value to an integer, we'll want to declare a number of other variables to hold the information we'll need during the course of the analysis. We'll use the names indicated below to refer to these variables throughout the walkthrough, but of course you may use whatever names seem best to you.

- `numDigraphs`, an integer tracking the total number of digraphs identified, initialized to 0
- `numTrigraphs`, an integer tracking the total number of trigraphs identified, initialized to 0
- `digraphCounts`, a map of strings (digraphs) to integers, tracking how many times each digraph appeared
- `trigraphCounts`, a map of strings (trigraphs) to integers, tracking how many times each trigraph appeared
- `prevLetter`, a character, to hold the previously read character, which should be initialized to a special value such as a space
- `earlierLetter`, a character, to hold the character read before that one, also initialized with the same special value

We have to declare several more variables than we did in "Codebreaker" due to the fact that we're tracking groups of letters, rather than individuals. In "Codebreaker," you would only need a single map (characters to integers) for tracking counts of individual letters, and a separate integer to track the total number.

With our tracking variables declared and initialized, we're ready to begin the analysis. The lines of text can be read in all at once and concatenated into a single massive string, or processed one at a time; since words (and therefore digraphs and trigraphs) are broken across newlines as well as spaces, it doesn't make much difference either way. To analyze the text, you'll want to iterate over every character in the string in turn and handle is according to the steps below.

If the character is a letter, it should be converted to uppercase (if needed). Then, if `prevLetter` is also a letter (that is, not the special value to which it was initialized), it should be prepended to the current letter to make a two-character string. The corresponding number in `digraphCounts` should be incremented by one (or set to 1, if this is the first occurrence of the digraph). Next, the value of `numDigraphs` should be incremented by one. If `earlierLetter` is also a letter, it should be prepended to the digraph string to make a trigraph, and the process repeated with `trigraphCounts` and `numTrigraphs`. Finally, `earlierLetter` should be set equal to `prevLetter`'s value, then `prevLetter` should be set to the current character.

If the character is a space, the values of `prevLetter` and `earlierLetter` should both be reset to their initial values (such as a space). Digraphs and trigraphs are broken across word divisions, so we don't want to accidentally record a combination of letters that doesn't actually exist. If reading lines one at a time, this process will also need to be repeated between each line.

For all other characters (that is, those that are neither letters nor spaces), simply skip over the character as though it does not exist. If convenient for your language, you may wish to remove these characters entirely before processing a string; for example, the Java code below uses a regular expression to replace all characters that are not letters nor spaces with an empty string, effectively deleting them:

```
text = text.replaceAll("[^A-Za-z ]", "");
```

Once all lines of the text have been processed in this manner, your analysis is complete and must simply be reported. Extract the keys from the `digraphCounts` map and sort them in alphabetical order; then iterate over each of these digraphs and print the relative frequency, using the formula given in the problem description:

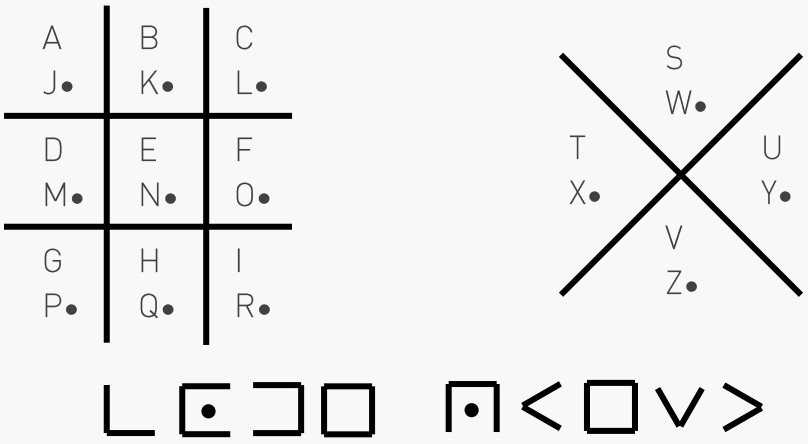
```
Double frequency = (digraphCounts.get(digraph) / numDigraphs) * 100.0;
```

Repeat this process for the `trigraphCounts` map, and the test case is complete.

Additional Background

As mentioned in the background sections for both "Codebreaker" and "Codebreaker Returned," frequency analysis is an important tool in cryptanalysis, and has been used to break virtually all forms of substitution ciphers. In order to understand the value of frequency analysis towards breaking these ciphers, it's necessary to understand how those ciphers work.

A substitution cipher is what many people think of when they think of codes and ciphers. A substitution cipher simply replaces (or substitutes) letters, groups of letters, or entire words or phrases with other symbols. These symbols are typically letters, but could be any distinguishable symbol; for example, the "pigpen cipher" is often learned by children and is a favorite in escape room adventures. This draws a number of basic shapes and fills the spaces it creates with the letters of the alphabet:



Each letter is replaced by a symbol drawn using the lines surrounding its space. Those letters with dots shown next to them have a dot (a "pig") shown in the middle of that symbol. The symbols shown below the diagram, for example, spell "CODE QUEST." Such a cipher is easy enough to read by those who know how the cipher works; but to those who don't, it's a pile of gibberish.

Cryptanalysis is the practice of breaking a cipher when the method of encipherment - or at least, the encryption key - is unknown. Frequency analysis is one such method of breaking ciphers, and substitution ciphers are particularly vulnerable to it. As seen in this problem, this involves analyzing a section of text to determine the relative frequencies of certain letters or groups of letters. In any language, certain letters will appear much more frequently than others. Vowels, in particular, tend to be the most common letters, seeing as how they perform a critical role in making words pronounceable. In English, the most common letters in general text are (in decreasing order) E, T, A, O, I, N, S, and R. In contrast, the letters J, Q, X, and Z hardly see any use by comparison. Likewise, certain pairs and triplets of letters appear more commonly than others. In English, "the" is the most common word and trigraph, which causes the digraphs "th" and "he" to be very common as well. Certain other quirks - such as how the letter 'q' is almost always immediately followed by 'u' - can help to break down a substitution cipher one letter at a time.

Consider the encrypted message below. We know that this was encrypted using a form of substitution cipher and that the original message was in English, but we lack any further information on how the encryption was accomplished.

YWLZLKIYWFSNTLUUNILIALNYZCBTHNVYUWFJLSSLVYBELYWLPULFSSALXPLKVCNKNZCUBUJBZZMLJWLKN
YYLTHYBKIYFMALNRYWLVBHWLAPULGYFLKVACHYBYYWLZFKILAYWLTLUUNILYWLMLYYLAYWLALUPZYU

Let's see if we can use frequency analysis to break this cipher. To do so, we need to count the number of times each letter appears in the ciphertext, and compare that with the overall frequencies of those letters in English. This comparison is shown in the table on the next page.

Right away, it is obvious that the highest and lowest frequencies in each data set do not align with one another nor follow any particularly clear pattern. If this cipher were a Caesar cipher, in which each letter is replaced with another letter a certain number of steps further down the alphabet, we would see some correlation between these high and low points. Since this does not exist, we can assume

that the cipher simply replaced each letter with a random different letter in the alphabet. As a result, we can use this data to make some guesses about the letters in our message.

The most common letters in the ciphertext - by a considerable margin - are 'L', at 19%, and 'Y', at 13%. We can start by guessing that these letters represent the most common English letters, 'e' and 't' respectively. Let's begin by replacing all L's in the message with e's and all Y's with t's; following common conventions, we'll leave the unidentified ciphertext letters in uppercase, and write (suspected) plaintext letters in lowercase:

Letter	English Frequency	Cipher Frequency
A	8.2%	5.0%
B	1.5%	4.4%
C	2.8%	2.5%
D	4.3%	0%
E	13%	0.63%
F	2.2%	3.8%
G	2.0%	0.63%
H	6.1%	2.5%
I	7.0%	3.8%
J	0.15%	1.9%
K	0.77%	4.4%
L	4.0%	19%
M	2.5%	1.9%
N	6.7%	5.7%
O	7.5%	0%
P	1.9%	2.5%
Q	0.095%	0%
R	6.0%	0.63%
S	6.3%	3.1%
T	9.1%	2.5%
U	2.8%	6.9%
V	0.98%	3.1%
W	2.4%	6.9%
X	0.15%	0.63%
Y	2.0%	13%
Z	0.074%	4.4%

tWeZeKItWFSNTeUUNIeIAeNtZCBTHNVtUWFJeSSeVtBEetWe
PUeFSSAeXPeKVCNKNZCUBUJBZZMeJWeKNtteTHtBKItFMAeN
RtWeVBHWeAPUeGtFeKVACHtBttWeZFKIeAtWeTeUUNIetWeM
etteAtWeAeUPZtU

This method of breaking ciphers involves a great deal of trial and error, but you can usually find ways to confirm your guesses. As discussed before, 'the' is the most common English word, and we should expect it to appear at least once even in a message of this length. In fact, we see seven places in which the pattern 'tWe' occurs; they are all highlighted above. If these are all the word 'the', then 'W' must represent the letter 'h'. This is supported by the similarity in the relative frequencies of these letters in the message ('W': 6.9%) and English ('h': 6.1%). Let's make the replacements:

theZeKIthFSNTeUUNIeIAeNtZCBTHNVtUhFJeSSeVtBEethe
PUeFSSAeXPeKVCNKNZCUBUJBZZMeJheKNtteTHtBKItFMAeN
RtheVBHheAPUeGtFeKVACHtBttheZFKIeAtheTeUUNIethеM
etteAtheAeUPZtU

Having identified the most common word, we can now try to identify other common letters, which should gradually allow us to fill in the rest of the alphabet. We do expect to see some variance between the frequency of letters in this message and in English; not all letters will be as easy to identify as 't,' 'h,' and 'e' were. Depending on the subject matter, these frequencies can be heavily skewed; for example, an article about zebras in zoos in Zimbabwe will have a much higher concentration of Z's than would typically be found elsewhere.

In this case, the next most common letter in the message is 'U,' at 6.9%. This could match to the third most common English letter, 'a,' however we can see a few problems with this. 'UU' appears more than once in the message, and the digraph 'aa' is very uncommon in English. While this could be the end of one word and the start of another, 'a' also doesn't typically appear at the end of words, and the

last letter in the message is also a 'U.' This same logic also rules out the next most common letters, 'o' and 'i.' The next most common letters are 'n' and 's;' Both letters can appear as doubles and at the ends of words, but 's' does both more frequently. Let's guess that 'U' is actually 's:'

theZeKIthFSNTessNIeIAeNtZCBTHNVtshFJeSSeVtBEethePseFSSAeXPeKVCNKNZCsBsJBZZMeJheKN tteTHtBKItFMAeNRtheVBHheAPseGtFeKVACHtBttheZFKIeAtheTessNIetheMetteAtheAesPZts

Already we're starting to see sections of words become clearer. In fact, most of the last line has been translated already; the only missing letters are represented by 'M,' 'A,' 'P,' and 'Z.' By making some educated guesses to fill in the blanks and comparing against the frequencies of those letters to possible English counterparts, we can deduce that these represent 'b,' 'r,' 'u,' and 'l' respectively, completing the phrase "the better the results." By making those replacements, our message is now:

theleKIthFSNTessNIeIreNtlCBTHNVtshFJeSSeVtBEetheuseFSSreXueKVCNKNlCsBsJBllbeJheKN tteTHtBKItFbreNRtheVBHheruseGtFeKVrCHtBtthelFKIertheTessNIethebettertheresults

Now that we've eliminated some letters, let's see if we can get some more vowels in place. The next most common letter in the message is 'N' at 5.7%; we can guess that represents 'a,' the next most common remaining English letter.

theleKIthFSaTessaIeIreatlCBTHaVtshFJeSSeVtBEetheuseFSSreXueKVCaKalCsBsJBllbeJheKa tteTHtBKItFbreaRtheVBHheruseGtFeKVrCHtBtthelFKIertheTessaIethebettertheresults

The most common English letters remaining are 'i,' 'o,' and 'n;' we can guess these match with some of the next most common cipher letters, 'B,' 'K,' 'F,' and 'G.' After more trial and error, the most likely combination appears to be 'B' = 'i', 'K' = 'n', and 'F' = 'o', as this allows us to distinguish more words:

thelenIthoSaTessaIeIreatlCiTHaVtshoJeSSeVtiEetheuseoSSreXuenVCanalCsisJillbeJhena tteTHtinItobreaRtheViHheruseGtoenVrCHtitthelonIertheTessaIethebettertheresults

At this point, the cipher is largely broken; the message reads more as a fill-in-the-blank puzzle than a line of nonsense. The message appears to begin with the words "the length of", which means 'l' replaced 'g' and 'S' replaced 'f'; this allows us to identify two instances of the word "message", identifying 'T' as 'm'; and so on.

thelengthofamessagegreatlCimHaVtshoJeffeVtiEetheuseoffreXuenVCanalCsisJillbeJhena ttemHtingtobreaRtheViHheruseGtoenVrCHtitthelongerthemessagethebettertheresults

With only a few letters remaining the subject of the message begins to come clear. Once the final letters are identified, we can add spaces and punctuation where logical, and fully restore the original message:

the length of a message greatly impacts how effective the use of frequency analysis will be when attempting to break the cipher used to encrypt it. the longer the message, the better the results.

Further Discussion Topics

- Have students develop their own substitution ciphers and attempt to break them using frequency analysis.
- How could you adapt a substitution cipher to foil frequency analysis (or at least make it less useful)?

Problem 16: Synaptic Server

Summary of Problem

Students must create a neural network to identify a street sign from a low-resolution image.

Topics Covered

- Machine learning
- Neural networks

Suggested Approach

This problem provides a relatively detailed explanation of the work that must be done to solve it; however the exact details can still prove to be difficult. To begin, we should declare data structures to contain the information needed by each neuron to perform its calculations:

- A list or array of integers containing the number of neurons contained in each layer in the network
- A list of lists of decimal values to contain the bias values for each neuron in each layer
- A list of lists of lists of decimal values to contain the input weights for each input for each neuron in each layer.

In each structure, the outermost List represents the entire neural network, and will contain L members, each representing one of the layers in the network. For the latter two structures, those child lists will in turn contain a number of members equal to the number of neurons in that layer. In the last structure, the innermost lists will contain a number of values equal to the number of neurons in the *previous* layer (or 64 members, in the case of the first layer).

Populating the first list, containing the layer sizes, is relatively simple. After reading in the number of layers (L), the next line contains that number of values representing these sizes. They simply need to be read in, split into individual values, and added to the List in sequence. Processing the next several lines will be a bit more complicated, so see the Java code below to review everything we've done so far:

```
// 'input' is a Scanner bound to System.in, the standard input channel
int numLayers = Integer.parseInt(input.nextLine());
List<Integer> layerSizes = new ArrayList<>();
List<List<Double>> networkBiases = new ArrayList<>();
List<List<List<Double>>> networkInputWeights = new ArrayList<>();

String[] layerSizeStrings = input.nextLine().split(" ");
for(int i = 0; i < numLayers; i++){
  layerSizes.add(Integer.parseInt(layerSizeStrings[i]));
}
```

First, we should declare an integer value called 'previous,' which will contain the number of neurons in the previous layer, which is equal to the number of inputs to be received by each neuron in the current layer. This should be initialized to 64; even through there *is* no previous layer for the first layer, it still receives input: one value for each pixel in the image. We're now ready to populate the more complicated nested data structures.

Each line corresponds to a single layer in the neural network, and contains all information needed to configure all neurons in that layer. The line should be read in and split along spaces into individual values. The first several values in this list - specifically, a number of them equal to 'previous' - should be converted to decimal values and added in sequence to a list. These represent the input weights for the first neuron in the layer. This List should be added to another List, which will in turn be added to the triple-nested List structure mentioned before. The very next value also pertains to this first neuron; it represents the bias value to be added to the neuron's calculation. This gets added to yet another List, which will eventually be added to the double-nested structure. This entire process repeats for each neuron in the current layer. The value of 'previous' is then updated to the size of the current layer, and we can move to the next line of input - that is, the next layer in the network. See the Java code below for this processing algorithm:

```
int previous = 64;

// each line of input corresponds to one layer
for(int layer = 0; layer < numLayers; layer++){
  String[] layerInformation = input.nextLine().split(“ “);
  // create layer-level data structures
  List<Double> layerBiases = new ArrayList<>();
  List<List<Double>> layerInputWeights = new ArrayList<>();

  // keep track of our position within the input data
  int infoIndex = 0;
  // process each neuron in the layer
  for(int neuron = 0; neuron < layerSizes.get(layer); neuron++){
    // create list to hold neuron weights
    List<Double> neuronInputWeights = new ArrayList<>();


    // read in input weights
    for(int i = 0; i < previous; i++){
      Double inputWeight = Double.valueOf(layerInformation[infoIndex++]);
      neuronInputWeights.add(inputWeight);
    }

    // read in bias value
    Double neuronBias = Double.valueOf(layerInformation[infoIndex++]);

    // store neuron data within the layer
    layerInputWeights.add(neuronInputWeights);
    layerBiases.add(neuronBias);
  } // end of ‘for neuron’ loop
```

```
  // store layer data within the network
  networkBiases.add(layerBiases);
  networkInputWeights.add(layerInputWeights);
  previous = layerSize;
} // end of ‘for layer’ loop
```

At this point, we've finally reached the final line of input. This contains 64 decimals representing the grayscale value of each pixel in the image. These will serve as the inputs to each neuron in the first layer. As before, we read these in and separate them along spaces; now we can begin our actual calculation.

Our algorithm will iterate through each layer of neurons. Within each layer, we will process each neuron in turn. Each neuron will receive the output generated by the previous layer (or our pixel values, for the first layer) and perform the calculation given in the problem description. This value will be added to a list, which will serve as the input for the neurons in the next layer of the network. Again, in Java this is done like so:

```
// to be updated following the processing of each layer
List<Double> previousOutput = new ArrayList<>();

// read initial inputs
String[] pixelValues = input.nextLine().split(“ “);
for(int i = 0; i < 64; i++){
  previousOutput.add(Double.valueOf(pixelValues[i]));
}

// process each layer in turn
for(int layer = 0; layer < numLayers; layer++){
  List<Double> currentOutput = new ArrayList<>();

  // process each neuron in turn
  for(int neuron = 0; neuron < layerSizes.get(layer); neuron++){
    List<Double> weights = networkInputWeights.get(layer).get(neuron);
    double output = networkBiases.get(layer).get(neuron);
    // add to the bias the product of each input and its weight
    for(int i = 0; i < previousOutput.size(); i++){
      output += weights.get(i) * previousOutput.get(i);
    }

    // this handles the max() function
    if(output < 0.0){
      output = 0.0;
    }

    currentOutput.add(output);
  } // end of ‘for neuron’ loop

  // overwrite previous output with what was just generated
  previousOutput = currentOutput;
} // end of ‘for layer’ loop
```

Once the layer loop is finished, the 'previousOutput' list should contain only five values; the last layer is guaranteed to contain five neurons. To determine our result, we simply need to identify which of these values is the largest and print the corresponding street sign's name.

Additional Background

A neural network is designed to mirror how actual brains work in humans and animals. As seen in this problem, a neural network consists of a massive number of individual "neurons," representing nerve cells in a biological brain, which work together to solve complex problems. When initially created, the developer creating the network likely has no idea how to configure the neurons to produce the desired results, however. As with real brains, a neural network must be "taught" certain behaviors. This training can take a number of different forms, generally suited to the particular task the neural network is expected to perform.

In this problem, we looked at a network that would be assigned classification duties; examining data to group it into one of several known categories. In these situations, developers will typically use "supervised learning" to train the network. This involves providing a network with a number of known inputs, paired with the expected results. Again using this problem as an example, a developer would provide the network with a series of images of street signs, then tell the network outright whether each one is a stop sign or speed limit or another variety. An algorithm would repeatedly run the network against these inputs, adjusting the configuration of each neuron until all of the inputs produced the expected result, or at least until the number of misidentifications are reduced to an acceptable level. This style of learning is ideal for situations when a human can quickly and easily identify the corrected answer, but needs an artificial system to handle future unknown inputs without that manual involvement.

Neural networks can also be applied to estimation problems; these are generally focused on unstructured data, which a researcher wants to analyze for any patterns that may be present. Such networks are simply provided with a large amount of data as input and a few basic assumptions about the data. For example, you may have seen news articles about artificial intelligence systems that produce their own writing within a certain genre; often these genres are very specific, like "obituaries" or "Doctor Who fan-fiction." These AI systems have been (or at least employed) neural networks, which were provided with a large number of existing, human-written texts within the same genre. The text these systems produce is often hilariously nonsensical, but nonetheless has clear similarities to the human-written versions. Within a specific genre, certain terms and phrases will be very common: in obituaries, you may often see phrases like "was survived by her 4 grandchildren" or "in lieu of flowers, please donate." Fan-fiction of any variety will contain frequent references to the main characters and recurring themes within that particular literary universe. Given enough data, the network will be able to recognize these common occurrences and generate something that incorporates them. The nonsense factor occurs because the network is unable to derive any real meaning from these terms or any knowledge of how they fit together; regardless, this sort of pattern identification can prove to be very useful in the course of research.

You've likely heard of Nobel laureate Ivan Pavlov, or at the very least his dogs. Pavlov conducted research into the digestive systems of animals (dogs in particular). He noticed that his dogs would begin to salivate even before they received their food, upon seeing the assistant who normally fed them. He conducted experiments in which he would cause a specific sound (a bell or a metronome) to be heard whenever his dogs were fed. After a period of time, the dogs began to associate that sound with food, and would begin to salivate upon hearing it - even though the sound had no direct relationship to being fed. This same concept is used in the last training method we'll discuss, "reinforcement learning." When a network produces an output for a given input, that output is assigned a score. A higher score is more desirable, and the network will adjust its algorithms to more frequently produce outcomes with higher scores and avoid those outcomes with lower scores. This sort of behavior is best used for situations where there is no single "correct" outcome, but rather an "ideal" one. For example, an AI trained to play chess may employ a neural network; it doesn't need to aim for a particular board layout, but instead needs to make moves that best position it to eventually win the game.

Further Discussion Topics

- What are some other situations that could benefit from the use of a neural network?
- For each of those situations, what would be the ideal means of training the network to function as expected?

Problem 17: Get Out the Vote

Summary of Problem

Students must implement a ranked-choice voting system that is able to determine the victor from a series of ballots.

Topics Covered

- Ranked-choice voting
- Cybersecurity

Suggested Approach

As with many problems, we first need to begin with reading in a line indicating the amount of data we're dealing with. Upon reading in the line, it should be split at the space; the two resulting strings should then be converted into integers.

We should then prepare the other data structures we need. First, we'll need to declare an array or list of strings capable of holding each of the ballots. Each ballot will be represented as a string with a length equal to the number of candidates; due to how the system works, every ballot must include a vote for each candidate, even the voter's last choice. We'll also need a map to associate each candidate with the number of votes. This could also simply be an array of integers; since each candidate is identified by a letter, in alphabetical order, the first integer in that array would correspond to the votes for candidate A; the second integer shows the votes for B, and so on. Next, we'll need something to contain the list of eliminated candidates; this could be a list, but could also be an array or even a string, as we'll demonstrate later. Finally, we need to know how many rounds of counting we've done (starting at 0), and how many votes a candidate must receive in order to win; this is equal to the number of ballots, divided by two, then plus one. See the pseudocode below for everything we've created so far:

```
let countStrings = input.readLine().split(“ “);
let ballotCount = countStrings[0] as int;
let candidateCount = countStrings[1] as int;
let ballots = new string[ballotCount];
let voteCounts = new int[candidateCount];
let eliminated = “”;
let rounds = 0;
let targetCount = (ballotCount / 2) + 1;
let winner = null;
```

Reading in the ballots is next; each ballot should be added to the 'ballots' array or list, but shouldn't be processed just yet.

Actually counting the ballots will likely require multiple rounds of counting. We'll need to declare a loop, set to terminate once a winner is identified. Alternatively, you could declare this as an infinite

loop (such as `while(true)`), but you must be sure to `break` out of the loop once a winner is identified. Within the loop, we'll want to first increment our 'rounds' counter, then count the number of votes for each candidate. Since each ballot (string) lists candidates (characters) in order of preference - favorite candidate first - we simply need to check the first character in each string to identify the desired vote. We then add one to the corresponding candidate's count, and move on to the next ballot.

```
while(winner == null){
  rounds++;

  for(let i = 0; i < ballotCount; i++){
    let vote = ballots[i][0];   // get first character
    let voteIndex = vote - 'A'; // characters are numbers too!
    voteCounts[voteIndex]++;    // find and increment count
  }
  // ...
```

Once all of the votes are tallied, we check to see if there is a winner. A winner is declared once one of the candidates reaches the target vote count of 50%+1. If any candidate has reached this threshold, save their letter to the 'winner' variable so we can report who won. At the same time, we should also check the counts for which candidate had the *lowest* vote count. Some care has to be taken here; any candidate we've already eliminated will have a vote count of zero (you'll see why in a moment), but we also have to consider the possibility that a non-eliminated candidate could receive zero votes in the first few rounds. See the pseudocode below:

```
  let lowestCount = ballotCount;
  let loser = null;
  for(let candidate = 'A'; candidate < 'A' + candidateCount; candidate++){
    if(voteCounts[candidate - 'A'] >= targetCount){
      winner = candidate;
    }
    if(voteCounts[candidate - 'A'] < lowestCount &&
        eliminated.indexOf(candidate) < 0){
      lowestCount = voteCounts[candidate - 'A'];
      loser = candidate;
    }
  }
```

If we found a winner, we should break out of the loop here and stop processing ballots. Otherwise, we need to eliminate the candidate we identified as the loser in that round. We should add their letter to the 'eliminated' list/array/string, then remove that character from each of the ballots. Any ballots for which that candidate was the first choice will now have the second choice as the first character. In any ballots where that candidate was a second or subsequent choice, they'll be removed anyway, allowing us to move directly to the next choice in line one other preferences are eliminated. Finally, we should reset all of the vote counts to zero to allow us to prepare for the next round of tallies. See below:

```
  if(winner == null){
    eliminated += loser;
    for(let i = 0; i < ballotCount; i++){
```

```
        ballots[i] = ballots[i].replace(loser, “”);
      }
      for(let i = 0; i < candidateCount; i++){
        voteCounts[i] = 0;
      }
    }
} // end of while(winner == null) loop
```

Once out of the while loop, we simply need to print out the results of the election. Calculate the percentage of votes the winner received by dividing their final vote count by the total number of ballots, and print the results.

Additional Background

Election security has been a hotly contested topic in the United States over the past few years. Subject to particular scrutiny is the use of electronic voting systems, which are becoming increasingly common as a way for governments to reduce the use of environmentally-unfriendly paper ballots and increase the speed of tallying results. Detractors of these systems claim that they threaten the security of elections because they can be easily hacked or otherwise manipulated into incorrectly skewing the results towards a particular candidate. In almost every case, however, these claims have been proven false.

Security is a major concern with many software applications, and as noted above, it is of particular concern with something that has major real-life impacts such as an election. But what is security, exactly? With any sort of computing system, security (or “cybersecurity”) is concerned primarily with three main points:

- Confidentiality - The knowledge that your data is kept secret and hidden from those not authorized to view it
- Integrity - The knowledge that your data has not been tampered with or altered in any way from its intended form
- Availability - The knowledge that you (and other authorized people) can access your data whenever it may be needed

Let’s look into some ways these concepts could be applied to an electronic voting system. Confidentiality is a major point in a fair election. People casting their votes need to be confident that their votes are secret; that nobody knows specifically how they voted. Voting machines, as a result, often aren’t given any information about who is voting. If they do display information, it’s simply to confirm that the person voting is the person it expects to be voting; that data can then be discarded, and would not be associated with the actual vote. Most jurisdictions do rely on some sort of voter registration system to confirm that those voting are allowed to do so, but this is handled by human officials, not the voting machines, and again isn’t associated with the final ballot.

Another point in confidentiality is keeping the overall vote counts secure. While press agencies often conduct “exit polls” to get an idea of how the election is going as voting is taking place, reporters can’t

interview everyone, and not everyone will respond (or choose to do so honestly). At the end of the day, the official count is what really matters. Unauthorized access to that data could skew an election, however. If one candidate were to become aware that they were in the lead halfway through the election, they could prematurely claim victory on the news and social media outlets. This may cause supporters of their opponent to think it's not worth voting and skip casting their ballot... even if there would have been enough of those supporters to take the lead. So, how do we prevent these leaks from occurring?

Encrypting data is one of the best ways to ensure that data can only be read by those who should access it. Without the key to decrypt the data, it's gibberish and therefore useless. As a result, any election related data can be encrypted by the machines, and will remain encrypted until it's read by authorized election officials. Encryption can be broken, however, and so it can't be the only means by which to rely on securing data. Fortunately, the simplest solutions are often some of the best. If there is no way to connect to the voting machines, there's no way for anyone to access the data they hold. For this reason, voting machines are almost never connected to the internet or any other network. This means that the only way to access the data is by physically removing the memory chip that holds that data; which is itself likely locked up. Once the election is over, an election official can unlock the compartment, remove the data chip, and personally deliver it to a government office for counting. Anyone else attempting to access the chip would be very obvious and could be stopped before they were able to gain access to the chip, let alone the data it holds.

This lack of network access also supports our data integrity and availability. Even if a hacker can't read encrypted data, they can still corrupt it or make it unusable by overwriting it with different information. Again, however, this would require direct physical access to the voting machine's memory chip since it contains no network connections. Even so, election officials are often required to conduct audits of the elections they oversee, even if no foul play was suspected. This helps to confirm that counts are accurate and supports the "integrity" part of the security mission. To support this, voting machines can use a number of methods to confirm that their counts are accurate. "Checksums" can be generated by hashing ballot data together; even a minor change to any part of any ballot will result in a wildly different checksum value, indicating that something is wrong. Blockchains, similar to those used for cryptocurrencies, can be used to reconstruct ballots in the order they were cast. Failing all else, a log simply stating information such as "Ballot 123456 was cast for candidate A" can provide a human-readable (and thus human-verifiable) record of every vote cast at that machine.

Another means by which to protect data integrity is simple validation; ensuring, even before the data is recorded, that it makes sense. In most election systems, a voter can pick only one candidate from those running. If two or more are selected, that's invalid data. In the ranked choice voting system demonstrated in this problem, a voter must assign a preference to every candidate. If any candidate doesn't have a preference, this is also invalid data. Even seemingly valid data may not be what the user intended; if a voting machine locks in a choice as soon as a candidate is selected, an accidental touch could cause a voter to vote for a different candidate than they intended. Performing simple validation

checks and adding confirmation mechanisms to prevent those situations are basic tenets of "secure coding," the practice of ensuring that software does its part to support security.

These and other security methods are a critical component of ensuring fair and open elections in any democracy; and indeed, they are working. These same security principles can inform other aspects of our lives as well, particularly as we increasingly move into a digital world.

Further Discussion Topics

- What other methods could be used to secure an electronic voting system?
- Consider another system, such as the one used by your school to manage student grades. How could you apply the three tenets of cybersecurity - confidentiality, integrity, and availability - to that system?

Problem 20: Plink

Summary of Problem

Students must find the highest value path by traversing through a binary tree of unknown height from top to bottom.

Topics Covered

- Heuristics
- Machine Learning

Suggested Approach

As with many problems, there are multiple ways to solve this problem. However, for a problem of this scale, the approach taken must balance two main considerations: accuracy and speed. A working solution must find the correct answer, for obvious reasons, but it must also finish running within two minutes, according to the contest's rules. Any solution which fails on either count is not viable.

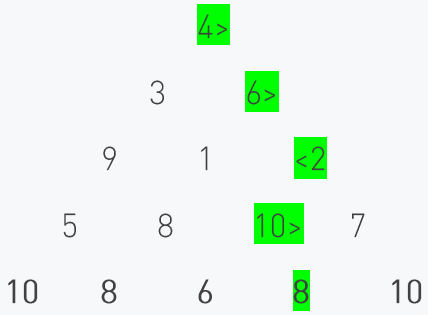
The simplest way to solve this would be with a "brute force" approach. Calculate the score of every possible path, identify the path with the highest score. This guarantees success under the "accuracy" header, since every path is considered. However, the approach fails at speed. Brute force solutions work well enough for very small cases, like the one shown in the problem description. With only four rows of paddles, there are only $2^4 = 16$ possible paths through the board; a small enough number that they could be worked through manually if needed. However, this number grows quickly, and the official test cases are much larger:

Official Test Case	Rows Present	Number of Possibilities
1	10	1,024
2	15	32,768
3	20	1,048,576
4	25	33,554,432
5	30	1,073,741,824

A brute force solution against this problem could easily take over five minutes to process the official input (some languages might be faster than others, but the fact remains that this is a poor way to attempt to solve this problem).

For large-scale problems such as this, a solution must be able to make more intelligent decisions about which path to take. Since our goal for this problem is to maximize our score, one approach could be to always take the path that offers the greatest increase in score (or smallest decrease, in the event both paths show a negative score). This algorithm, known as a "greedy" algorithm, allows us

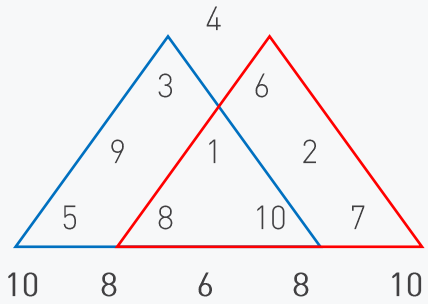
to complete the task nearly instantly, since we ignore all but one path. However, the accuracy of this solution is dismal; it fails to find the correct solution even for the sample test case.



A greedy algorithm would take the path RRLR, pictured above, always moving to the paddle with the greatest value. The optimal path, as seen in the problem description, requires a left turn right at the beginning. This approach is similarly inaccurate in the official test cases, falling short of the expected score in the final case by nearly two thousand points. Clearly, a middle ground between these two approaches is needed, one that runs quickly enough to meet our needs, but one that considers enough options to be able to make intelligent decisions.

Enter heuristics. "Heuristic" is an adjective, meaning something that enables someone (or in this case, some*thing*) to discover or learn something for themselves. In software engineering, heuristic algorithms are a class of algorithms that allow a program to make an approximation of the best answer based on a limited set of data. While the above two solutions were clearly unusable, we can take what we learned from them in order to create a more accurate algorithm. A brute force solution was slow, but was guaranteed to produce the correct answer eventually. A greedy algorithm was fast, but failed to consider other viable options. By combining the two, we can create a heuristic algorithm.

Let's begin with the basic structure of the greedy algorithm; moving down towards what the best option appears to be. Rather than simply looking at the next row of paddles, however, let's go a bit further and consider the next three rows of paddles. For each step, consider two miniature boards, each three rows high and centered on one of the two paddles we can move to next:



Within these triangles, conduct a brute force solution; that is, consider every possible path through each of these triangles and determine the highest value path through them. Since the triangles are only three rows high, there are only four possible paths through each, so this is a trivial operation for the computer. In the picture above, the blue triangle's highest value path is worth 3 + 9 + 8 = 20 points,

whereas the red triangle can only reach 6 + 2 + 10 = 18 points. Since the blue triangle has a higher value, we should move in that direction for our first step. This process repeats again and again for each step we take.

By considering all possible paths we could take over the next several steps, we are able to get a rough idea of what our score may look like if we move in either direction. We take advantage of the high accuracy of a brute force approach, but reduce the scope of that approach to allow us to make use of it in a reasonable amount of time. We then use the goal-oriented focus of the greedy algorithm to make an informed decision about each step. This approach allows us to get the correct answer for the sample test case, but fails to return the correct answer for three of the official test cases (although it does get two of them correct!). Some adjustment might be needed.

Heuristic algorithms rely on a heuristic function to inform their decisions. In this case, our heuristic function is the brute-force approach, to a depth of three rows. Making small adjustments to a heuristic function allows us to control the algorithm's accuracy. In this case, we can make a simple adjustment by increasing the number of rows examined at each step in the process.

Official Test Case	Rows Present	Expected Score	$h = 1$ (greedy)	$h = 3$	$h = 6$	$h = 9$
1	10	286	272	265	283	**286**
2	15	1093	1051	**1093**	1081	1081
3	30	2593	2393	**2593**	**2593**	**2593**
4	25	5533	4388	5272	5403	5358
5	30	9777	7788	8805	**9777**	**9777**

By increasing the heuristic depth to nine rows, we are able to get the correct score for three of the test cases... but one of the ones now incorrect is one we'd previously gotten correct with just three rows. The addition of more information actually served to mislead us in that instance; it's important to remember that the heuristic function only provides an approximation of the best answer based on the information available. Too many "red herrings" can throw off that approximation. Additionally, care must be taken to ensure your efficient heuristic algorithm doesn't become an inefficient brute force method; the first test case only has ten rows, so by searching ahead nine rows, we're not really "approximating" anything by that point.

Heuristic algorithms, as a result, tend to focus somewhat on trial-and-error. Adjustments to the function must be made in order to account for new data or unforeseen situations. Algorithms that are able to do this on their own, with little to no human intervention, are known as "machine learning" algorithms. It's not strictly necessary to develop such an algorithm for this problem, but to make this particular heuristic algorithm work, you will need to attempt multiple heuristic depths in order to find the ideal solution for each test case. A simple brute force approach to *that* problem is likely sufficient; by using this algorithm and looping through all possible depths between 2 and 20 for each test case (cutting off if the total number of rows is reached), a Java solution is able to return the correct answer

for all test cases within about ten seconds. More efficient solutions do exist, but for the purposes of solving this problem, this is sufficient.

Additional Background

Another possible approach to this problem steps outside the realm of simple heuristic algorithms and into the realm of machine learning. As mentioned briefly above, machine learning algorithms are those that are capable of making automated improvements to themselves over time based on their observations. This is a subset of the broader field of artificial intelligence, although falls well short of true artificial intelligence (what we typically see in science fiction, where computers are capable of making truly independent decisions).

Since artificial intelligence, and by extension machine learning, aim to replicate our own ability to think and reason, many topics in AI seek to mimic natural, biological processes. The simple existence of life on our planet proves that such methodologies work in practice, and adopting them into computer software allows computers to approach problems in a different manner from which they would normally be capable. Once such algorithm is particularly well suited to this problem - a genetic algorithm. To understand the motivation behind this algorithm, though, a short biology lesson is in order.

All living things depend on deoxyribonucleic acid - or DNA - to define what it is and how it functions. In complex lifeforms, DNA is organized into a series of chromosomes, within each of which a number of individual "genes" control various aspects of the lifeform's appearance and certain other traits. During reproduction, each parent contributes a random set of genes to create a new full set of chromosomes, resulting in a child whose DNA contains recognizable portions of each parent's DNA. As a result, a child often bears a similar appearance to both of its parents.

When Charles Darwin visited the Galápagos Islands in 1835, he was able to witness the long-term effects of this manner of genetic reproduction on an unusually stark scale. He observed that the mockingbirds and tortoises (and, upon later review, the finches) present upon each island differed significantly from those on the other islands. Due to the physical boundaries between each island, there was little opportunity for interbreeding of animals from different islands. As a result, each island developed its own somewhat unique subspecies of several types of plant and animal. These subspecies were particularly well suited to the specific features of their island. Tortoises from drier islands, with less low-lying shrubbery, typically had longer necks and higher shells than those from wetter islands so that they could eat leaves from trees. Cacti on islands without iguanas had developed softer spines, owing to the lack of a need to defend themselves against predation. Based on these observations, Darwin developed his theory of "natural selection," in which, over time, creatures with traits better suited towards survival would be more likely to reproduce and pass those traits onto their descendants. Less-useful traits - shorter necks and harder spines, to use the earlier examples - would eventually become less dominant and fade entirely.

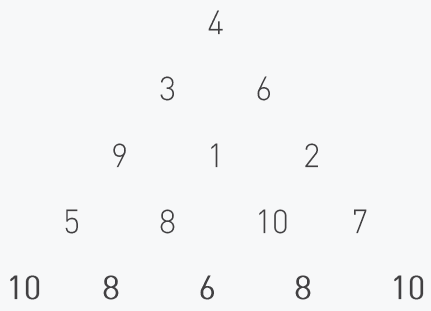
Natural selection forms the backbone of the genetic algorithm. In a genetic algorithm, the solution to a problem is broken down into a series of distinct values. A function - known as the "fitness function" - is developed that can analyze the values within a possible solution and produce a quantitative measure of how well it solves the problem. The best solutions are then used to generate yet more solutions, while those less ideal solutions are discarded; a direct implementation of the concept of "survival of the fittest" (it's worth noting that this particular phrase wasn't actually coined by Darwin himself, although he did approve of it). This process is repeated over and over again, simulating many generations of genetic reproduction.

A genetic algorithm includes a few other parameters to control how it operates: a population size (P), the number of offspring (O), and a number of generations (G) to calculate. The steps of a genetic algorithm can be summarized as follows:

1. Create an initial (random) population of size P
2. Loop through the following steps a total of G times:
    a. Randomly pair up members of the population
    b. Generate O offspring from each pair
    c. Rank all of the offspring according to the fitness function
    d. Retain only the top P solutions to form a new population

As mentioned previously, this problem is particularly well-suited to a genetic algorithm. A solution to this problem consists of a sequence of directions - left or right - and the best solution can be identified by the score earned by following that path. This fulfills the two requirements for a genetic algorithm; a solution consisting of a sequence of distinct values, and a quantitative means of determining a solution's fitness.

Let's walk through an implementation of this algorithm to solve this problem's sample test case. To begin, an initial population of solutions must be created. This initial population is mostly random; we don't expect the best solution to be included here. However, if possible, it's best to include any extreme edge cases as possible solutions, to ensure that they can be given due consideration. Let's create an initial population of six solutions, including the two extreme cases and four semi-random ones:



- LLLL = 31
- RRRR = 29
- LRLR = 22
- RLRL = 29
- RRLL = 28
- LLRR = 30

To process our first generation, these solutions must be randomly paired up so they can produce offspring. For demonstration purposes, we'll pair them up as they appear above; LLLL with RLRL, and so on. To create a child solution, a character is randomly selected from each parent for each position.

					Child 1	R	L	L	L
Parent 1	L	L	L	L	Child 2	R	L	R	L
Parent 2	R	L	R	L	Child 3	L	L	R	L
					Child 4	L	L	R	L

This particular pairing resulted in the child solutions RLLL, RLRL, LLRL, and LLRL. Obviously two of those are identical, and one is identical to one of its parents; such is the nature of utilizing random values in an algorithm such as this. Since both parents had an L in the second and fourth positions, all children they created also would have had L's in those positions. This limited the number of possible paths they could create. However, this mirrors what we see in nature; two parents with freckles are very likely to have children with freckles, because they both have the genes for it.

Once all three pairs of parent solutions create their child solutions, we now have the following list of 12 child solutions:

From LLLL and RLRL:
- RLLL = 27
- RLRL = 29
- LLRL = 32
- LLRL = 32

From RRRR and RRLL:
- RRRL = 29
- RRLR = 30
- RRLL = 28
- RRRR = 29

From LRLR and LLRR:
- LRLR = 22
- LLRR = 30
- LRRR = 26
- LLLR = 29

These are ranked according to their scores; only the top six (highlighted above) "survive" to form the next population, matching the size of the original population. Where ties exist, as they do here, they can be broken in a random or consistent manner as desired. This completes the first generation. Those six solutions then pair off to create their own children, and so on, until a pre-determined number of generations has been processed. Once this is done, the highest-ranked surviving solution is selected as the answer.

Over time, the top-ranked solution in each generation should improve. Even after this first generation, we've already stumbled across the correct answer not once, but twice; if those solutions should happen to be paired off for the next round of reproduction, they'll be guaranteed to produce identical offspring, further increasing the chances that the correct answer is found when processing ends. However, because this algorithm depends so heavily on randomness, this remains only a chance.

If, for example, LLLL had been produced as a child in this generation, it would have survived to the next generation; with a score of 31 points, it's second only to the correct answer, and would be likely to survive any time it was produced. The longer it survives, the more likely it would continue to produce LLLL children, which would themselves likely survive to create even more LLLL children. Within a few generations, it's entirely possible that the population could become entirely homogenous; only containing LLLL solutions, with no possibility of any other outcome. This is known as a "local maximum" - a solution that appears to be the best compared to other similar solutions, and a potential death trap for genetic algorithms.

In order to avoid local maxima, a genetic algorithm must be carefully tuned to ensure that the algorithm has the greatest practical chance to examine all possible solutions. The settings we used in this example - population size of 6, and only 4 offspring per pairing - are impractically low. Typically, genetic algorithms deal with populations on the order of hundreds, with hundreds of offspring created per set of parents, over hundreds of generations.

Other techniques can be incorporated to further reduce the risk of local maxima; a random "mutation" could cause a part of a child solution to change to an otherwise impossible value. In the homogenous LLLL example above, if a child's third character were to be "mutated" into an R, the correct answer (LLRL) would be suddenly available, and would gradually take over the population itself. Naturally, not all mutations would be as beneficial; changing the first character to R would yield a considerably inferior result (RLLL), which would be discarded immediately.

Even with these changes, a genetic algorithm is never guaranteed to find the best possible solution; it remains within the realm of heuristic problems and can only guarantee an approximation of the best solution. However, for relatively small-scale problems such as this one, they can nonetheless have a surprisingly high degree of accuracy. To quote *Jurassic Park*, "life finds a way."

Further Discussion Topics

- What are some other heuristic approaches that could be used to solve this problem?
- In addition to mutation, how might a genetic algorithm be modified to mimic other natural processes, such as interbreeding of different populations? What impact would these changes have on the algorithm's accuracy?

Problem 21: Shopping Spree

Summary of Problem

Students must implement a shopping cart system by reading and processing commands to purchase items from a given inventory.

Topics Covered

- Atomic operations
- Session isolation

Suggested Approach

This problem involves keeping track of a large amount of data of different types. While it is possible to solve this problem using primitive data types and arrays, it will be significantly more difficult. Students who are unfamiliar with their language's implementation of Maps (called "dictionaries" in Python) should read up on that first, as we'll be using that type of data structure extensively in this walkthrough.

After reading in the size of the inventory and the number of commands, we need to initialize the inventory. Since each inventory item will have a unique name, we can associate any data for an item with its name. We'll create two maps, as demonstrated in Java below, to hold pricing data and stock levels:

```
Map<String, Integer> stockLevels = new HashMap<>();
Map<String, Double> prices = new HashMap<>();
```

Both of these maps will use the item's name as the key. Any time we need to look up information about an item, this will allow us to quickly retrieve it with a minimum of fuss. With each line of inventory data, we simply need to split the data values and load them into the appropriate map.

Before we start processing customer commands, we need to initialize the data structure for storing each customer's shopping cart. Each customer is identified by a unique numeric ID; they then each have a list of items they'd like to purchase and the desired quantity of each those items. This will require a nested Map structure to track properly. In Java again, this could be set up as:

```
Map<Integer, Map<String, Integer>> carts = new HashMap<>();
```

This declares a map, which contains another map as its value. The outer map uses the customer's numeric ID as a key. When we need to access a customer's cart for any reason, we will use that ID to obtain the inner map. This represents the customer's actual shopping cart. It uses the names of requested items as keys, and the number of those items the customer has requested as its value.

To begin with, we won't place anything into the 'carts' map. We don't have any customers at this point, and even if we did, we don't know what they'd like to purchase. This map will slowly be populated as we process the commands.

With that said, let's begin. Each command must be fully processed before we can continue to the next, as the results of one command may impact our ability to process a later one. The problem description outlines what should be done for each command, but how to do it is a slightly different matter.

Regardless of which command we're processing, we first need to obtain the customer's cart. Each line of input will start with the customer's ID. With this, we can retrieve the associated value from the 'carts' map we created above. If this value is null, we're dealing with a new customer. We should initialize their cart by creating a new map and inserting it into the 'carts' map with the customer's ID. In Java:

```
Map<String, Integer> customerCart = carts.get(customerId);
if(customerCart == null){
   customerCart = new HashMap<>();
   carts.put(customerId, customerCart);
}
```

Now we're ready to process the actual command. The input line should be broken apart at spaces; the first item in each line is the customer's ID, which we used above. The second part will always be either "ADD", "REMOVE", or "CHECKOUT". We should set up a series of if/else blocks to handle each of these commands.

For ADD commands, we first need to validate the command's arguments. Get the name of the item and the requested quantity (the third and fourth items in the input line). Using the name of the item, we need to identify how many items of that type are already in the customer's cart; if no such entry exists, we can assume the amount is zero. We add this amount to the newly requested amount to determine how many total items would be in the customer's cart as a result of this command. Then, we check our 'stockLevels' map to determine how many items we actually have available. If this count is equal to or greater than the customer's new cart total, we set the customer's cart total to that level. We *do not* reduce the available stock at this time; remember, the customer has to pay for the items first! Either way, we end by printing the appropriate output message.

```
int totalCount = requestedCount;
if(customerCart.containsKey(itemName)){
   totalCount += customerCart.get(itemName);
}
if(stockLevels.get(itemName) >= totalCount){
   customerCart.put(itemName, totalCount);
   System.out.println("Added " + requestedCount + " " + itemName + " to customer " +
customerId + "'s cart");
}
else{
   System.out.println("Not enough " + itemName + " for customer " + customerId);
}
```

REMOVE commands are fairly similar. After getting the name and requested amount to remove, we'll need to check how many already exist in the customer's cart. Again, if no entry for that item exists, the count is zero. If this count is greater than or equal to the amount to remove, we subtract that amount from the cart and store the new value. For simplicity's sake, you may wish to remove the item from the cart entirely if the resulting count would be zero; this helps avoid any confusion when processing CHECKOUT commands and future REMOVE commands. Once items have been removed as appropriate, the corresponding output message should be printed.

```
int existingCount = 0;
if(customerCart.containsKey(itemName)){
  existingCount = customerCart.get(itemName);
}
if(requestedCount <= existingCount){
  int resultCount = existingCount - requestedCount;
  if(resultCount == 0){
    customerCart.remove(itemName);
  }
  else{
    customerCart.put(itemName, resultCount);
  }
  System.out.println("Removed " + requestedCount + " " + itemName + " from customer " +
customerId + "'s cart");
}
else{
  System.out.println("Customer " + customerId + " does not have that many " + itemName +
"s");
}
```

CHECKOUT commands are considerably more complicated. We have to verify that we have sufficient stock to cover all of the customer's cart items; other customers may have claimed and purchased some items since this customer added those items to their own cart. We'll want to create a list of strings to hold names for each item where we don't have enough stock. To populate this list and process the purchasing of items, we'll iterate over each existing key in the customer's cart, then check the corresponding value (the amount the user has requested) with the existing stock level stored in the 'stockLevels' map. If the stock level is equal to or higher than the requested amount, we should decrement the value in stockLevels by the amount the user has requested. If not, we should add the item to our 'outOfStock' list and remove the item from the user's cart. Once all items have been checked and - as applicable - removed from our stocks, we can begin processing output. First, the 'outOfStock' list should be sorted, then iterated over to produce our error messages. We'll then want to tally and print the user's total bill; the sum of the products of the number of each item the user has ordered and that item's cost, as recorded in our 'prices' map. Finally, the customer's cart should be deleted, in order to make way for any future purchases that customer might make.

```
List<String> outOfStock = new ArrayList<>();
double totalBill = 0.0;
for(String itemName : customerCart.keySet()){
  int requested = customerCart.get(itemName);
```

```
  int inStock = stockLevels.get(itemName);
  if(inStock >= requested){
    stockLevels.put(itemName, inStock - requested);
    double price = prices.get(itemName);
    totalBill += price * requested;
  }
  else{
    outOfStock.add(itemName);
  }
}
customerCart.removeAll(outOfStock);
Collections.sort(outOfStock);
for(String itemName : outOfStock){
  System.out.println("Out of stock of " + itemName);
}
String totalBillStr = String.format("%.2f", totalBill);
System.out.println("Customer " + customerId + "'s total: $" + totalBillStr);
```

Additional Background

Shopping carts are a good example of "atomic" operations. An atomic operation is any operation in which the context in which it operates is not changed by an external operator. Let's illustrate this by looking at a shopping scenario in which carts are not treated as atomic operations.

Alice and Bob are both shopping on a website for the latest gaming console. As usual with gaming consoles, stock is very limited and disappears quickly. When Alice and Bob find the entry for it on the shopping website, it says there's only one still available. Alice and Bob both quickly add it to their carts and scramble to complete the checkout process. It just so happens that Alice and Bob both click the final "place order" button at the exact same time. The website processes both of their orders simultaneously. As they're processing, both threads check the inventory and find there's still a single console available. Having passed the validation, both threads complete the order, and both Alice and Bob get a confirmation email telling them they can expect to receive their new gaming console in 3-5 business days.

Obviously, this presents a problem. The company only has one console left, so it can't send one to both Alice and Bob. They will have to cancel one of the orders, refunding the payment and possibly taking other action to avoid losing Alice's or Bob's future business. How could this have been avoided?

The problem occurred because the checkout operation wasn't atomic. Remember, in an atomic operation, the context of the operation cannot be changed by an external operator. Here, the context is the available inventory of gaming consoles. The "external operator" for each of Alice's and Bob's transactions is the other transaction. Both transactions are attempting to reduce the number of available consoles by one. If both succeed, that number will go negative, which doesn't make any sense. To avoid this behavior, only one transaction can be allowed to be processed at a time.

In order to improve performance, many website operations are conducted synchronously - that is, more than one operation can take place at a time. The internet would be considerably less useful if

only one person could access a website at a time; in reality, a web server is able to respond to multiple requests at once, processing incoming requests before previous ones have finished. However, as we've seen above, certain operations need to be run "asynchronously" - one at a time.

This is often achieved through the use of a "synchronization lock." A synchronization lock allows one thread of execution to "reserve" a particular data object for its exclusive use. As long as it holds this lock, no other operations can access that object. They must wait in order for the lock to be released. In our shopping example, a synchronization lock should be employed to ensure that only one checkout operation is able to access inventory records at a time. While holding a lock, a checkout operation should confirm that sufficient stock exists, then (if it does) remove the requested items from inventory. It can then release the lock and allow other checkout operations to perform the same validation and removal actions.

It's important that synchronization locks be used carefully. They should only contain those operations which actually depend on exclusive access; in the checkout scenario above, it should only contain the operations we mentioned. Payment processing and displaying confirmation messages can likely be conducted separately, as those operations don't rely upon the inventory records. Wrapping an entire operation in a synchronization lock effectively makes that operation fully asynchronous, which can have a severely detrimental effect on performance.

Further Discussion Topics

- What might occur if an operation never released its synchronization lock? How might such a scenario occur?
- Brainstorm additional scenarios in which synchronization locks would be useful.

Problem 25: Take Me Out to the State Machine

Summary of Problem

Students must read a set of baseball scorecards in order to recreate the state of the game at various points and determine the final score.

Topics Covered

- State machines

Suggested Approach

This complex problem requires carefully reviewing the information provided in the problem packet. Each action indicated in each team's scorecard has a specific outcome, which must be processed correctly in order to fully reproduce the course of the game.

To begin with, the scorecards must be processed. After reading in the number of innings and states to report, the away team's scorecard is next. The first line is used to associate player numbers with initials; this will be needed to generate the state reports once we are ready to start printing output. The line can be split along pipe characters, and each player's initials added to a map or simple array for later use. The following line, containing only dashes, serves no functional purpose and can be discarded.

Before we continue with reading in the scorecard proper, we need a data structure to hold the information presented in each cell. Each cell contains up to five possible points of data; if (and how) the player reached each of the four bases and if (and how) they were forced out. Each of these data points is represented by a string of two characters. These characters may both be spaces, but that nonetheless tells us some important information; namely, that the player never stopped at the associated base (or was never forced out). This could be a simple struct (structure) if supported by your language, or a formal class:

C++ Struct

```
struct scorecard_cell {
    string firstBase;
    string secondBase;
    string thirdBase;
    string homeBase;
    string out;
};
```

Java Class

```
private class ScorecardCell{
    public String firstBase;
    public String secondBase;
    public String thirdBase;
    public String homeBase;
    public String out;
}
```

We'll then need to declare a two-dimensional array, list, or similar data structure sufficiently large enough to hold all of the scorecard data. The first dimension will represent each inning; the second dimension will represent each player on the team. This will allow us to walk through the game, one player at a time, in order to recreate exactly what happened.

With our data structures prepared, we can begin reading in the scorecard. For each inning, we'll want to read in four lines of input. This includes the three lines that contain the actual data we need, and the divider line between this inning and the next. To process each player's information, we can slowly consume each of the three data-containing strings. The first two characters of each string can be removed, as they simply contain the inning number (which we can identify from the index of our for loop) and the divider marking the start of the first cell. After this point, each player's cell consists of seven characters; the seventh character being the pipe character dividing this cell from the next (or forming the right edge of the scorecard). Each of the strings in the data structure we created earlier can be populated as follows:

- First base: second line, characters 4 and 5
- Second base: first line, characters 2 and 3
- Third base: second line, characters 0 and 1
- Home base: third line, characters 2 and 3
- Out: second line, characters 2 and 3

Once each of these strings are read in, we can delete the first seven characters from each line and repeat for the other eight characters. This entire process then repeats again for each inning to complete processing of the away team's scorecard. We then repeat everything from the last few paragraphs again to process the home team's scorecard.

At this point, we have read in all the input representing the scorecards, leaving only the listing of what states we need to print out as output. While these can be read in at this time and stored in a list or array for later reference, we are not ready to process this output just yet. Since there is no guarantee about what order the reports will go in, we must assume that they could be presented in any order. As a result, we need to process the entire game, then go back and determine which states actually need to be printed in which order.

To begin processing, we'll need to declare a few variables in order to keep track of what's going on at each point in the game:

- Which player (if any) is on first, second, and third bases, and which player is at bat
- The scorecard cell data for those players, as applicable
- The number of the next player at-bat for both the home and away teams; this must be incremented after each player is processed, and reset to 1 whenever it exceeds 9.
- The current number of outs
- The number of runs scored for both the home and away teams

Most of this information will be needed to generate the "state of the game" images that we will have to print to the output very shortly. We'll also want to create a map, associating strings to strings - the keys in this map will be phrases describing points of time in the game, in the same format as provided in the input (e.g. "TOP OF 3 PLAYER 5"). The values will be strings containing the baseball diamond images that we must print. Once we have fully reproduced the game, we will have a full dictionary

depicting every point in the game. We can then read and process the final input, printing out the requested images in the expected order.

To illustrate how this will work, let's begin looking at how to reproduce the game. We'll want a large for loop to iterate over each inning. Within the for loop will be two while loops, both executing while the number of outs is less than three. The first while loop will represent the top of the inning, while the away team is at bat; the second loop manages the bottom of the inning, or the home team's at bat. Before each while loop, you will want to reset most of the values we listed above; namely, information about the batter and runners on each base and the number of outs. Both while loops will contain the same logic, and so you may wish to move that logic into a function.

```
for(int inning = 1; inning <= numInnings; inning++){
  firstBase = null;
  secondBase = null;
  thirdBase = null;
  atBat = null;
  outs = 0;

  while(outs < 3){
    // top of the inning; handle away team plays
  }

  firstBase = null;
  secondBase = null;
  thirdBase = null;
  atBat = null;
  outs = 0;

  while(outs < 3){
    // bottom of the inning; handle home team plays
  }
}
```

The beginning of each while loop represents the point in time at which a new player is stepping up to home plate to go to bat. This is the point at which we'll need to report in the output, so our first step is to generate one of the output images. Since we're only going to print out a handful of these images, generating one for every possible point in time could be seen as wasteful; if you've already read in the input indicating which moments to display, you could check that list and see if an image is actually necessary. Don't print it yet, however, as the order in which the images are printed is not guaranteed to be chronological. Once the image is created, save it to the map along with the phrase describing this point in time.

```
atBat = awayTeamNames[awayPlayerIndex];
atBatScorecardData = awayTeamScorecard[inning - 1][awayPlayerIndex];

String stateName = "TOP OF " + inning + " PLAYER " + (awayPlayerIndex + 1);
// or "BOTTOM OF " and home... variables for the home team

String stateImage = "  " + (secondBase == null ? "__" : secondBase) + "\n";
```

```
stateImage += " /  \\\n";
stateImage += (thirdBase == null ? "__" : thirdBase) + "  ";
stateImage += (firstBase == null ? "__" : firstBase) + "\n";
stateImage += " \\  /\n";
stateImage += "  " + atBat + "\n";
stateImage += homeRuns + "-" + awayRuns + "\n";
stateImage += outs + " OUT\n";

stateMap.put(stateName, stateImage);
```

With this done, we have to consider the possibility that the inning may have been abandoned. As seen in the sample input, if the home team is ahead when they come up to bat in the final inning, the game may end early, since there's no way for the away team to score at that point. If the at bat player's scorecard cell is completely blank, break out of the while loop in order to continue with processing the rest of the test case.

Assuming that's not the case, we can begin processing this player's actions, and the actions of any runners as a result. The problem description lists ten possible plays that could be made. Some of the plays are very straightforward and only allow one possible outcome, however several others allow the runners to advance bases. Usually when runners advance, there's a possibility that they could be tagged out. The table below summarizes all of the possible plays and their possible outcomes:

Play	Batter on base?	Runners advance?	Runners can be tagged out?
BB / Walk	First	1, as needed	No
S / Single	First	1	Yes
D / Double	Second	2	Yes
T / Triple	Third	3	Yes
HR / Home Run	Home	4	No
B / Bunt	No	1	No (would be reported as DP otherwise)
FC / Fielder's Choice	Maybe (could happen after at bat)	1 or more	
DP / Double Play	Maybe (could happen after at bat)	1 or more	Yes (the other DP)
K / Strike Out	No	No	No
PF / Pop Fly	No	No	No

Let's start with the easy-to-handle cases first.

Both a K and a PF result in the batter getting out, without impacting the runners at all. For both of these cases, the play will be indicated in the center "out" position of the batter's scorecard cell; we need not check the other data points in the cell, because the presence of either K or PF means they never made it on base. To handle these, we simply increment the number of outs by one, then move on to the next player.

A HR allows the batter to score immediately, along with any active runners. An HR will be reported in the bottom "home" position in the cell; as with the immediate outs above, we need not check other data points, as the presence of HR excludes the possibility of any other plays. Increment the relevant team's score by one, plus one more for each active runner. Reset all runners to null (or some other placeholder value to indicate nobody is on that base), then advance to the next player.

A BB lets the batter move to first without the risk of getting tagged out, along with any runners that need to move to make room. This is reported in the right-hand "first base" of the cell, and may be accompanied with other data (which we'll ignore for now). Before the batter moves, we must check each base in sequence to determine how many runners must also move to make way. If there are runners on all three bases, they all advance to the next base; this means that the runner on third scores, and the scores must be updated accordingly. If there are only runners on first and second, they both advance to second and third, respectively. If there is no runner on second, any runner on first moves to second (any runner on third stays put in this case). Regardless, at this point first base should be open, and the batter moves there.

A B is similar to the K and PF cases we saw earlier; the batter is out immediately and never makes it on base. However, each of the runners is able to advance one base (resulting in any runner on third base scoring). We don't need to be concerned about any of the runners being tagged out, as two outs in the same play would be reported as a Double Play, or DP. Since we know a DP did not occur here - the batter's out was reported as a B - the runners must have all made it safely to the next base.

Before we go further, we need to make an important point clear. Once a batter gets on base, they aren't done yet; they'll attempt to advance at every opportunity, and may be tagged out. Whenever we indicate that a particular player is on a particular base, we also need to keep track of their scorecard cell. This information will be important for properly handling the following plays. This cell data can be stored in a separate variable (which must be kept in sync with other variables indicating which player is on which base), or looked up from the scorecard when necessary, but must be readily available.

S, D, and T plays all indicate that the batter made it onto first, second, or third base (respectively) after batting. These plays will be marked on the respective bases, and all allow any existing runners to advance the same number of bases. As with the BB case, we need to handle the runners in sequence - third, second, first, then the batter - to ensure that each base is open for later runners to reach. In each case, we need to check the base that the runner should be able to attain. For the third-base runner, this will always be home plate, as it's the only base remaining; for other runners, this could be home, third, or second, depending on the play made by the batter. In any case, if the runner made it to their target base, their scorecard cell should show the current batter's player number, and the runner can be moved to that base (and the team's score incremented if they reached home). If it does not, the runner failed to advance and must have been tagged out along the way. This means that the runner should have an out method indicated in the center of their cell, either an FC or a DP. The runner should be removed from the field, and the number of outs incremented by one. Once all runners are processed in this manner, the batter can be moved to the base corresponding to their play.

If a batter does not have any other case covered up to this point, they must have hit the ball and attempted to run, but were tagged out before completing their play. The batter will not have any data for any base in their scorecard cell, but only a DP or FC in the middle. In such a case, the other runners may attempt to advance. If the batter had an FC, the other players must have all successfully reached a base, indicated by the presence of the batter's number on the base they reached. If the batter had a DP, one of the other runners will also have been tagged out and marked with a DP.

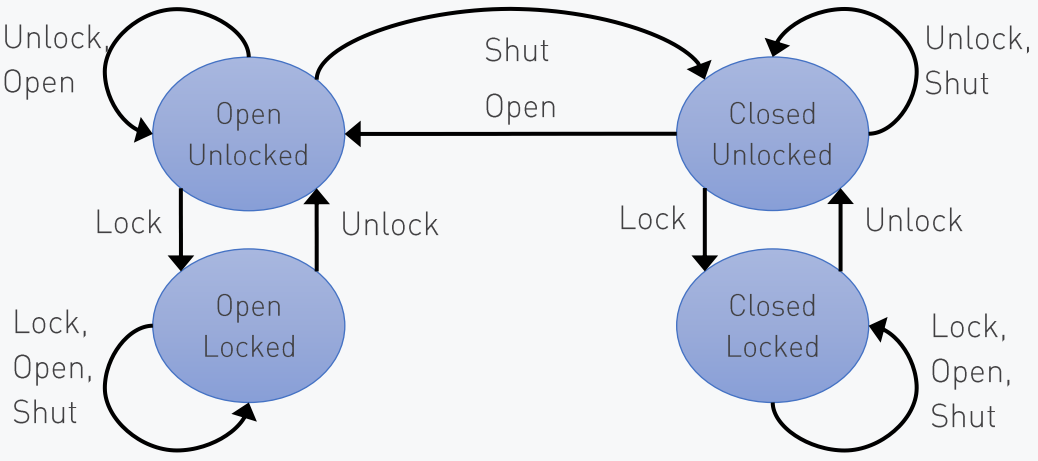
At this point, all plays have been processed. The index number of the current team's player should be incremented by one; if this index number then exceeds nine, it should be reset to 1. If the number of outs equals or exceeds 3, the team's at bat is over; the index number will then indicate which player will be first to bat in the next inning. This processing alternates between the away and home teams for each of the indicated innings, until the end of the game.

Now that the full game has been processed, we can finally begin our output. If not already done, the list of required reports should be read in. For each report, the corresponding 'stateImage' string must be printed to the output channel. Once each image is printed, the final results of the game must be announced. With that finished, the testcase is finally complete.

Additional Background

As mentioned in the problem description, a baseball game can be seen as a complex sort of state machine. A state machine is a conceptual construct in which a machine can be seen as being in one of several distinct "states." Certain inputs allow the machine to transition between states. These states and transitions can be illustrated in the form of a state diagram. Each state is represented by a circle, and transitions are drawn as arrows connecting those circles.

Consider a door with a lock. This door can be seen as having four states; open/locked, open/unlocked, closed/locked, and closed/unlocked. Certain actions can be taken to change the state of the door: namely, it can be opened, shut, locked, and unlocked. If we were to draw out a diagram showing the states of the door and how those actions allow transitions between them, it would look something like this:



As you can see, each of the four states have a number of arrows coming out of it, representing each of the four possible actions - or inputs - that can be applied to the system. A state machine must be able to accept any input in any state; however, as we can see here, it is possible to simply ignore some of those inputs if they don't make particular sense for that state. For example, one cannot open a door which is already open, nor shut one that is already shut. Furthermore, the only action one can take on a door which is locked is to unlock it; even attempting to shut an open & locked door will fail, as the deadbolt will prevent the door from closing. The door can start in any of these states, and will always be in one of these states; there is no "end condition" that would cause the door to stop functioning.

These transitions can also be summarized in a state transition table, which lists each state and how it responds to each input:

Initial State	Input	Final State	Output
Open, Unlocked	Open	Open, Unlocked	None
	Shut	Closed, Unlocked	Door is closed
	Lock	Open, Locked	Deadbolt is extended
	Unlock	Open, Unlocked	None
Open, Locked	Open	Open, Locked	None
	Shut	Open, Locked	None
	Lock	Open, Locked	None
	Unlock	Open, Unlocked	Deadbolt is withdrawn
Closed, Unlocked	Open	Open, Unlocked	Door is opened
	Shut	Closed, Unlocked	None
	Lock	Closed, Locked	Deadbolt is extended
	Unlock	Closed, Unlocked	None
Closed, Locked	Open	Closed, Locked	None
	Shut	Closed, Locked	None
	Lock	Closed, Locked	None
	Unlock	Closed, Unlocked	Deadbolt is withdrawn

This conveys the same information the diagram displays, but additionally provides details of the expected outputs that occur in response to an input. Here, that output is limited to obvious situations, but in a practical system, it may display error messages for invalid inputs, or may have other effects, such as writing information to memory.

In the case of our baseball game, there are too many states involved for us to accurately represent them with a single state diagram or table here; however, we can examine limited portions of a state machine as a subsystem. Let's consider a player at bat as a state machine; they will likely have at least five states:

- At Bat
- On First

- On Second
- On Third
- Out of Play

Unlike with the door, we have clear "start" and "end" states. The "start" state (or "initial" state) represents the state in which the machine (player) must begin. In this case, the player starts "At Bat," as that's the only way for them to begin interacting with the game. They remain active until they are in the final state, "Out of Play," which represents the player either being tagged out or crossing home plate and scoring. At that point, they can no longer interact with the game until they come up for bat again, and so the state machine represented by the player effectively shuts down.

Even this portion of the baseball game will have a complicated state diagram, but we can at least examine a portion of the state transition table to get an idea of how it might look. Here, the inputs are the possible plays that can be made by the player at bat.

Initial State	Input	Final State	Output
At Bat	BB	On First	Other runners advance as needed
	S	On First	Other runners advance one base unless out
	D	On Second	Other runners advance two bases unless out
	T	On Third	Other runners advance three bases unless out
	HR	Out of Play	All other runners score
	B	Out of Play	Other runners advance one base; increment outs by one
	FC	Out of Play	Other runners advance at least one base; increment outs by one
	DP	Out of Play	One runner out; all others advance at least one base; increment outs by two
	K	Out of Play	Increment outs by one
	PF	Out of Play	Increment outs by one

As you can see, there's a lot going on; many of the player's actions will further inform the actions taken by the other runners already on base. As a result, five states may not be quite enough to fully represent all possible outcomes for the player. For example, an input of "BB" for a runner in the "On Second" state may not have any outcome; the runner would advance to third base only if there is also a runner on first.

Regardless of the complexity, visualizing a system as a state machine can help to ensure that the system will respond in the way its developer intends. Control statements such as if, else, or switch can clearly map to specific lines in a state transition table. This can serve as a powerful debugging tool and a means of verifying that software meets all expected requirements.

Further Discussion Topics

- Expand the state transition table shown above for the baseball player, or develop a state diagram depicting those states and transitions. How can this be expanded to include a greater portion of the game as presented by this problem?
- Consider a past problem or programming project you have worked on. Could it be represented as a state machine? If so, draw the state diagram for that machine.