Problem 164: The Good Ship Input

Coach's Resource Guide

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:

```
// 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#
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:

```
// 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 = Console.ReadLine();
int space = foo.IndexOf(‘ ‘);
int databaseCount = Convert.ToInt32(space.substring(0, space));
int shipCount = Convert.ToInt32(space.substring(space + 1));

// C++ (with ‘using namespace std;’)
string foo;
getline(cin, foo);
size_t space = foo.find(‘ ‘);
int databaseCount = stoi(foo.substr(0, space));
int shipCount = stoi(foo.substr(space + 1));
```

Finally, some languages 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++.

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

// C++
int databaseCount, shipCount;
cin >> databaseCount >> shipCount;
```

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 allow retrieval of specific items by their index number. 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.

```
// 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.

```
// 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.

```
// 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(list<string>::iterator iterator = database.begin();
    iterator != database.end();
    ++iterator){
  cout << *iterator << '\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?