1st Annual

Brookfield Computer Programming Challenge

2017

Problem Analysis

Passing Bills

This is an input-output problem. Teams need to output the first integer given to them. This can be done by reading an integer using scanner.nextInt() and printing it to System.out by using the println method.

Apple-Pen

This is a relatively straight-forward string manipulation problem. We can use a for loop to go through each test case. Then, for each test case, we can get the last word of the next two lines, and print them in the desired format.

Teams have to be careful to clear their buffer after calling scanner.nextInt() if they choose to use scanner.nextLine() to read the line of input.

The Halting Problem

This is an implementation problem. We have to check if the time any two cars arrive are within **s** units of each other. This can be done in two ways. The first is by using two nested for loops and comparing the difference between the values in the array at the two indices, being careful to avoid off-by-one errors and making sure to ignore the case when the two indices are equal. This will run in $O(n^2)$ efficiency, which is fast enough for the input constraints.

A more efficient solution is to store the arrival times of the cars in an array and sort it in n*log(n) time. You can then look through the array and only compare adjacent indices. This can be done because all cars take the same amount of time to cross the intersection.

One

We must first account for the special case in which 1 is the input, in which we output "Neither." For all other cases, we must determine whether the input is prime.

You can check whether an integer is divisible by another by seeing if the first modulo (%) the second is 0, or a%b==0.

After that, this is a mathematical problem. We can check if **q** is prime by seeing if it is divisible by any of the whole numbers less than it, not including 1. This will not be efficient enough, however. To improve performance, we only need to check if it is divisible by any of the numbers between 2 and Math.sqrt(q), inclusive. This will result in maximum number of comparisons per test case of less than 10^5, which is fast enough. Note that this is true because if **q** is not prime, it has at least 2 factors (which we will call a and b), and the largest possible value of the smallest of these two numbers must be less than the square root of **q**.

Negligent Norbert

This is a greedy and combinatorial problem. This problem can be solved using a greedy algorithm and counting efficiently. We must count the total number of characters in all of Norbert's responses. Imagine that Norbert was using a language with the **c** characters. There are **c** one letter words. It would never be wrong to use all **c** of these words first. Norbert would then use all words with 2 letters, of which there are $\mathbf{c}^2$ words. (The first character can be any of **c** characters, and for each of those, there are **c** possibilities for the second). By continuing this reasoning, there are ($\mathbf{c}^x$) unique x letter words.

Note that we don't actually have to look through each combination, and we can find in constant time how many characters we will need for all words of a given word length. This solution will run in the O(log(n)) time, which is fast enough for the input constraints. (log(10^9) is less than 100)

Calculatus Eliminatus

This is a set-transformation, or perhaps unique-data-structure, problem. You cannot use the straightforward approach of using a boolean array because it will result in running out of memory, and, even if it didn't, would take too long. We instead have to transform this data into range-pairs, for the constraints to be realistic.

This problem asks us to eliminate **u** ranges of numbers out of a set containing integers from 1 to n and report which number is left over. We cannot make a boolean array of a billion elements because iterating through it will be too slow. Instead, since there are a relatively small number of ranges, we can combine adjacent ranges by searching through them with two nested for-loops. After combining the ranges, we can delete both of them from a list of available ranges and add the combined range until no more ranges can be combined.

At this point there are 3 cases which must be accounted for:

1. There are two ranges left, in which case the object is at the position not included

2. There is one range left which does not include position 1, in which case the object is at position 1
3. There is one range left which does not include position n, in which case the object is at position **n.**

This will have an efficiency of O(**u**^{2}), which is fast enough for the input constraints. Note that an efficiency of O(**n**)--where **n** is the number of positions, not the number of ranges--would not have been efficient enough.

*It is also possible to sort the list of ranges in such a way that only adjacent ranges in the sorted list need to be compared, so only one for loop would need to be used. This could improve the efficiency to O(**u***log(**u**)), but is much more complex to implement, and the code for it is harder to read.

Bike Race

This was a graph theory problem. The problem asks the user to find the diameter of an undirected graph. This can be found in multiple ways. The easiest for the time constraints is to use an adjacency matrix and the Floyd-Warshall algorithm, which has an efficiency of v^3, to find the distance from every node to every other node. You can then search through the adjacency matrix for the maximum value (because all nodes are reachable). The maximum distance of the race is one more than this value.

Another possible solution is to use Dijkstra's algorithm from each node and record the maximum value of any node to any other node.

LAST Robotics (Bonus Problem)

This is intended to be the most difficult problem in the set and, while not complicated to implement, is very difficult to figure out. It would most likely be solved as a recursive definition problem. Since the input is much too large to actually calculate **s** up to the insanely large team numbers given, we must find a pattern.

  0   10   20   30   40   50   60
 0123456789 123456789 123456789 123456789 123456789 123456789
1234
s=r**bb**r**b**rrb**b**rrbrbbr**b**rrbrbbrrbbrbrrb**b**rrbrbbrrbbrbrrbrbbrbrrbbrrbrbbr**b**...

First we should notice that **s** does not change, it only increases in length as the number of iterations change.

Next, since the length of the sequence is doubling each time, we might begin by looking at values at powers-of-2 indices. All of these appear to be ‘b’. Interesting. It makes sense, though, because the zeroth character of **s** is ‘r’, so every power of 2 will be the opposite of ‘r,’ which is ‘b’. (Because **s** doubles in length each iteration, so each power-of-two index will be the start of the string added, which will be the opposite of the zeroth character in **s**)

But what about things that aren't powers of 2? Let's look at index 12, which is ‘r’. The golden question to ask is, “Why is it ‘r’? What determined that?” Well, the step before we increased the length of **s** to 16, it was of length 8. The character that ended up mapping onto index 12 is the opposite of the character at index 4. More specifically, it is at index 12-8. In general, for index i, the index that mapped onto i is the opposite of the character at i-p where p is the largest power of 2 less than i. This approach can be implemented in a recursive solution, stopping at index 0, which, of course, is ‘r’.

There is also a more interesting approach that yields an equivalent answer, but is even easier to implement. If you notice the pattern in the above recursive strategy, what we are really doing is subtracting the largest power of 2 that we can, flipping our answer each time. We don’t have to flip for every number of 2 less than ‘i’ though, just for the the powers of 2 that we actually subtract. This turns out to be the same way binary numbers are defined. You start with a number, put a 1 in the column representing the largest possible power of 2 that is less than your number, subtract that from your number, and keep going. So, we actually get the same answer as the recursive solution by just counting the number of ‘1’ bits in the binary representation of the index of question, and outputting ‘r’ if it is even or ‘b’ if it is odd. It also makes sense that all the powers of 2 are ‘b’ now. They only flip once, because they are represented with a single 1 and following 0’s.

Please email 13dharmeyer@gmail.com for any questions, comments, or complaints about the problems. We hope you enjoyed them! Solutions will be posted to wumbogames.github.io under the BCPC link later this weekend.