2nd Annual

Brookfield Computer Programming Challenge

2018

Problem Editorial

Everybody Do the Flop

The problem required you to be able to detect either the number of lines in the input or the number of lines before “everybody do the flop” was reached. The problem can be solved by using a while loop and printing out “wait for it” until the target string is detected.

Funny Pairs

The pair (x, y) is funny iff x*x == y+y. This can be simplified to x^2 == 2*y, or $x^2/2$ == y, where y is an integer. Notice that if x is divisible by 2, x^2 is as well; if x is not, x^2 is not. Therefore, we can check to see if the problem has any solutions by checking whether x is even. If x is even, the solution is y = x*x/2. Otherwise, there are no solutions.

Goat Simulator

This problem can be solved in any number of ways. One interesting solution uses Fenwick trees (You can learn about them here: https://tinyurl.com/algorithmsLiveFenwickTrees). Other possible solutions involve sorting the obstacles and queries and processing everything in a sweep, using segment trees, or using prefix sums. Since the prefix sums method is probably the easiest to implement, I will quickly cover that solution path.

Say we know, for each position, how many obstacles are before it. Then, to calculate the number of obstacles between *a* and *b* (*a* < *b*), we can subtract the number of obstacles before (inclusive) *a* from the number of obstacles before *b*. We can precompute a prefix sum array in linear time and then count the number of obstacles between each query in constant time. This gives us a final runtime of approximately O(10^6 + n + q) which is around to 10^6 operations.

Codeforces Handle

Notice that the problem guarantees that no input string will have two adjacent vowels that are different. That means all strings will contain sets of identical vowels, each separated with consonants, (e. g. “ccaaaceeecuccociccooo”). Because of this, we don’t have to worry about the tricky cases like “aea” for which deleting the first vowel leaves a different string from deleting the second, but deleting the first and second leaves the same as the second and third. Also, notice that a set of *n* identical vowels could have *n*+1 possible outcomes (it could be empty or include up to all *n* of the characters). Since all these choices are independent, the total number of ways is the product of the lengths of all vowel-sections. That is, “steeve” has (2+1)*(1+1) possible mappings.

Tax Evasion

The only way to save money at all is to abuse the rounding property of the tax. We want to round down as much as possible to avoid paying tax when we can. The first key observation is that this only happens when the checkout price modulo 3 equals 1. When the checkout price modulo 3 equals 0, no rounding is done (we can pay the tax exactly), and when the checkout price modulo 3 equals 2, the tax is rounded up (we want to avoid this when possible).

A greedy strategy clearly works. The goal is to round down the greatest number of times. If we can round down with what is currently in our cart, we definitely want to checkout now (because you cannot ever round down more than once per checkout, and things can’t possibly get strictly better than they are now if we include more items in this checkout). If what we have is a multiple of 3, it doesn't matter whether we checkout now or later because no rounding is applied anyway. If what we have is one less than a multiple of three, we should try to include another item that is one less than a multiple of 3 before checking out. If we can get one, the total will be one more than a multiple of 3 and we can round down, only paying one extra cent in tax instead of two (ex: 2 + 2 == 4, which rounds down. We should include both 2's, which each round up, in the same checkout, so that we only have to pay one extra cent in tax instead of two). However, if there are no more things equal to 2 modulo 3, then we just have to pay the extra cent in tax for this item.

Two

A naive anti-prime generator will show that anti-primes are very rare. Actually, there are only about 50 of them in the 10^7 range we care about. Even without writing a generator, one can see that there will be very few anti-primes, since the upper bound of the number of factors of *n* is about the cube root of *n*, so there cannot possibly be more than the cube root of 10^7 anti-primes (which is about 215). Since there are so few, we can easily iterate over all 50 anti-primes for each query if only we can find them quickly.

There are several ways of finding anti-primes. The intended solution was to count the number of factors of each number up to 10^7 efficiently and then iterate over that list keeping track of the most number of factors seen so far to find all the antiprimes and add them to an ArrayList or TreeSet. We can efficiently count the number of factors each number has by incrementing a count for all numbers that are divisible by 1, then incrementing for all numbers divisible by 2… then incrementing by all numbers divisible by 10^7. To increment everything divisible by *i* in an array of size *n*, we need to make about *n*/*i* array accesses. In total, this will have a runtime of n+n/2+n/3+n/4…+n/n, which is n*(1+½+⅓+¼…1/n) which is bounded by n*log(n). This is provable with some quick calculus, but I won't bore you with that. This way, we can precompute all anti-primes in 10^7*log(10^7) ≈ 2*10^8 time, which takes about 2 seconds to run. Then we can answer all queries easily. In Python (a language that is notoriously slow), this takes FOREVER to run, so this method has to be used just to do the cheesy solution mentioned below.

A real solution in python is also possible by noticing some properties of anti-prime numbers discovered by Ramanujan. Of course, this gets pretty technical and no one probably cares (if you do care, it's super interesting; I recommend watching the numberphile video on it), so I won't keep discussing them.

A cheesy outside-the-box solution is also possible that involves somewhat-naively counting the factors of a number by counting up to the square root of each number. This runs in about 10^7*sqrt(10^7) time or a little over 10^{10} operations. This took my computer around 200 seconds or 3 minutes to run. Once all the anti-primes have been calculated, they could just be hard-coded in since there are so few.

Bonus Problem: Tic-Tacs

The first key observation is that skips don't matter at all. If Alice is going to win on a board if neither player had any skips, then if Alice uses a skip, she will have just given Bob the winning position. If, on that same board, Bob uses a skip, he gives Alice a losing position, so Alice will just counter his skip with one of her own resulting in the same board but with each player have one fewer skips.

Next, suppose there is an even number of rows. We can state an invariant as follows: the first player must always take from an even-numbered row (the second-from-bottom, forth-from-bottom, et cetera). We can enforce this by having Bob always take the tic-tacs directly below the ones Alice takes. Since Alice always takes from even numbered rows, Bob will take all the tic-tacs on row 1, including the bottom right one. Therefore, in a game with an even number of rows, the second player (Bob) always wins, regardless of the number of columns.

In a game with an odd number of rows, Alice can just take the entire top row on the first turn. We are left with a game with an even number of rows and Alice moving second, in which Alice will win as we have shown above.