UAPC 2022

Solution Sketches

Thanks!

Problem setters, testers, and judges:

- Jason Cannon
- Noah Gergel
- Noah Weninger
- Johnson Wu

Kattis Team: Greg Hamerly, Fredrik Niemela

UACS - Event site organization and point-of-contact for sponsors.

And Thanks To Our Sponsors!

Division 2 Problems

Canadians, eh?

Read in the entire line as a string. If the last 3 characters are “eh?”, you are done!

Is It Even?

Recall that when you multiply two exponents with the same base, the exponents are additive. I.e, $2^x * 2^y = 2^{x+y}$. With this in mind, 2^k divides the product $x_1*...*x_n$ only if the sum of the factors of 2 across all x_i is >= k.

For then going and solving this, one can just loop through each x_i and count the number of 2 factors that divide into that x_i, and add to a running total.

Do not actually construct the entire product, that takes too much time (arithmetic operations are not O(1) time if the values are huge).

Magical Runes

Imagine the input string spells out a binary number backward (from right to left).

Example: ABBAA corresponds to binary 00110, i.e. the decimal number 6.

Each day is just like adding 1 to this binary number. So after 2 days, the new binary number is 6+2 = 8, which is 01000 in binary.

Convert back to the string format: AAABA

Quadratic Dissonance

Playing with a few examples (i.e. drawing a few pairs of quadratics) quickly shows the answer will either be one of the minima of the two quadratics, or else at the point of intersection of the two quadratics (there will be only one point of intersection since both quadratics have x^2 as their leading term).

Quick Tips:

- The minimum of a quadratic $x^2 + ax + b$ is at $x = -a/2$
- Two quadratics $x^2 + ax + b$, $x^2 + cx + d$ will meet at the point x satisfying:

  $ax + b = cx + d$

  If $a = c$ then there is nothing to do (i.e. the minima of one of the quadratics is the answer). If $a \neq c$, solve for x.

Trip Odometer

Simply recomputing the answer from scratch for each d[i] is too slow, you will be summing about 100,000 numbers 100,000 times.

Faster solution: Let S be the sum of all input values. For each distinct value d[i] in the input, include S-d[i] in the output.

Running time: O(n log n) due to sorting the output.

Number Colosseum

This requires an efficient implementation of the steps provided in the problem.

When a contestant appears it either *(based on the sign)*: **(1)** gets added to the current team or **(2)** faces off against contestants in a last-in first-out (LIFO) ordering.

We can implement this using a Stack. Pushing a number to the Stack takes O(1) time and retrieving the most recently added number also takes O(1) time.

Each combat involves at least 1 contestant that has never participated in a combat implying the total number of combats is O(N). So the entire running time is O(N).

Fun fact: You can easily determine which team wins by considering the sign of the sum of all numbers.

Patchwork

This is a longer implementation that requires storing the types of patches conveniently and handling possible index out of bound errors for arrays.

The small bounds of the problem allow us to “copy” each patch character-by-character onto the final quilt. However, this requires storing all of the type of patches in memory first using an container to hold 2-dimensional arrays.

Be careful with indices to ensure that you do not accidentally index out of bounds when copying patches that hang off the edges of the quilt! You can solve this by either **(1)** including boundary checks when copying patches onto the quilt or **(2)** allocating a larger array for the quilt and displaying only the “visible” portion.

Card Divisibility

Let f(L, R) denote the answer. The first observation is that the answer is the same as

$$L + (L+1) + (L+2) + \ldots + R \mod 9$$

For the same reason that the “divisibility by 9” test works.

Then notice is that there is a pattern:

$$f(L, R) = f(L, R + 9) = f(L, R + 2*9) = f(L, R + 3*9) = \ldots$$

This is because the sum of any 9 consecutive numbers is divisible by 9 so the remainder does not change if you add 9 more consecutive numbers.

Card Divisibility

So reduce R by the largest integer multiple of 9 such that the resulting value is still at least L. Then you can solve it directly with a loop that iterates <= 9 times (for each value from L to R).

Example: L = 5, R = 2000000

By the logic before, f(5, 2000000) = f(5, 11) since 2000000 - 11 is a multiple of 9. We calculate 5+6+7+8+9+10+11 mod 9, which is 2.

That is, f(5, 2000000) = 2.

How to reduce R in general?
R -= 9*((L-R)//9) (the division is integer division)

*Remember to use long long since the numbers can go up to 10^12

Travelling Caterpillar

There is exactly one way to travel between any two nodes because the input is a tree. So, one way to see the route travelled is as a partial depth first search (DFS) tree from the root. We can compute the minimum distance with a straightforward modification of DFS.

Perform a DFS where you keep track of

- the total distance you've travelled, and
- a boolean indicating whether there was any leaf to munch in the subtree

Then, in each node of the DFS, add a child node's distance to the total only if there was a leaf to munch in the subtree rooted at that child. Finally, multiply the answer by two because we need to do a return trip.

Ordinary Ordinals

We obviously have f(0) = {}. For N >= 1, notice:

f(N) = N+1 + f(0) + f(1) + f(2) + ... + f(N-1)

Where the N+1 is for N-1 commas and the two outermost brackets.

This simplifies to f(N) = 5*(2^{N-1}) - 1.

Use fast exponentiation (modulo M).

Running time: O(log N)

Division 1 Problems

The ones that were not already presented earlier

Problematic Polygons

Conceptually simple, but needs care to implement.

Simply try each integer angle θ. Rotate the initial location of the object by θ degrees about (0,0).

Then check to see if any pair of line segments between the object and the container intersect.

Finally, you need to know if the object is actually in the container or not. Use a standard point-in-polygon test.

Input is small enough that these can all be done the simple way (e.g. trying all pairs of line segments). Just have to be careful when you implement it to get all details correct.

Clumsy Cardinals

When cardinal A removes cardinal B, it is like A becomes B. So, rather, think of it as cardinal B eliminates cardinal A without moving.

Consider a graph G = (V,E) whose vertices are cardinals and edges represent two cardinals that could attack each other if no other cardinals were present. The answer is the # of components in this graph.

Quick Proof Sketch. Pick a spanning tree in each component. Iteratively pick a leaf and delete it: either the parent can eliminate it directly or there is something in the way that could eliminate the leaf.

But adding all edges makes a graph that is too big.

Clumsy Cardinals Cont'd

Instead, for each diagonal with a cardinal, pick any single cardinal to "represent" that diagonal.

For each cardinal C, put an edge between it and the representative cardinals for its two diagonals (if they differ from C). Also put an edge between C and the representative cardinals from the <= 4 diagonals that are adjacent to the 2 diagonals for C.

The connected components of this graph are the same as the graph with all possible edges. But now the number of edges is O(n). So any search algorithm works.

Running time: O(n) if using a linear-time search algorithm (eg. DFS).

Cubic Cycle

A smart brute force works. From vertex 0, try to generate a cycle by recursively generating a path from 0. Apart from the initial call at the root, the branching factor is <= 2 at each node (since the incoming edge is already used).

But this is too slow. Need one more pruning.

When considering the two unused neighbours u, w of a vertex v in the recursion (i.e. whenever the branching factor would be 2), only consider recursing with u if w would still have at least *two* unused neighbours (and vice versa).

If so, then w will never again be one of the two guesses for a call with branching factor 2.

Cubic Cycle Cont'd

So only n/2 of the calls in any root-to-leaf path in the recursion tree will have branching factor 2. Thus, the number of leaf nodes (and also the answer itself) is O(2^(n/2)).

Total running time is O(n * 2^(n/2)) when you account for the work done across all recursive calls (even the ones that don't branch).

Similar Spacing

First, let's look at a simpler question: given a (max distance) and b (min distance), what is the maximum number of restaurants that we can build?

We can answer this in O(n log n) time using a priority queue (and if you use a fast language you can probably get AC with O(n^2)).

- Precompute d(i,j) to be the distance between site i and site j
- Let S(i) be the set of all j < i such that a <= d(i,j) <= b. This is the set of possible previous sites if we build at site i.
- Let f(i) be 1 + the max f(j) for j in S(i). This is the max # we can build if we build at site i.
- Answer is max f(i) over all i.

S(i) and f(i) can be built up iteratively since they only depend on S(j) and f(j) for j < i. Use a priority queue for answering the "max f(j) for j in S(i)" query efficiently.

Similar Spacing Cont'd

So now we have a function F(a,b) that tells us the maximum number of restaurants we can build if we pick a as the maximum distance and b as the minimum distance.

How many possible values for a and b are there? There might be as many as n choose 2 distinct values of d(i,j). So, if we naively try all possible values of a and b, the running time would be O(n^5 log n). Too slow.

However, notice that if we fix b, then the number of restaurants we can build is monotone in a; that is, if we increase a we can always build at least as many restaurants, and if we decrease a we won't be able to build more restaurants.

So, we can try all n^2 values of b, and binary search for a value of a that lets us build the desired number of restaurants. Carefully choosing the upper bound for the binary search based on the best so far can save a lot of time. Running time: O(n^3 log(n) log(d_max))