A Planetary Status

Author: Zeki

Notice that the distance from any "tip" of any normal inscribed star to the center of the planet is simply the radius of the circle representing the planet. This means that checking whether Pluto qualifies to be a planet or not comes down to checking whether its radius is bigger than or equal to 1000 miles.

B Path to Pluto

Author: Prayaag

C Calibration Complications

Author: Jimmy

Let's frame this problem as finding the target number t, the value that all elements of both arrays should be equal to in the end.

For the first array, values can only increase. Therefore, t needs to be greater than (or equal to) every element a_i. This is true iff $t \geq$ the maximum a_i. For the second array, values can only decrease. Therefore, t needs to be less than (or equal to) every element b_i. This is true iff $t \leq$ the minimum b_i.

Putting these together, we get that t needs to be between the maximum a_i and the minimum b_i. We can choose anything in this range, and then calculating the number of operations is fairly straightforward; it's just the sum of all (absolute) differences between t and all a_i, b_i. Additionally, if the maximum $a_i >$ the minimum b_i, there are no values that satisfy our above requirements, so there is no solution, and we print -1.

D Celestial Sky

Author: Michelle

First, we initialize a 2D array and set the grid coordinates of all stars to 1. Then, for each black hole coordinate, we will scan the 9 cells adjacent to it and set the grid coordinates of any cells with stars to 0. We then initialize a 2D prefix sum array and fill each cell $p[i][j]$ with values of all the stars contained in the rectangle from $(0, 0)$ to (i, j). We can compute this sum from left to right and top to bottom via `p[i][j] = p[i-1][j] + p[i][j-1] - p[i-1][j-1] + v[i][j]`. Then, to answer a query we use the inclusion-exclusion principle to determine how many stars are in each rectangle. The related code looks like this: `rectangle = p[x2][y2] - p[x1-1][y2] - p[x2][y1-1] + p[x1-1][y1-1]`. For more details on how these formulas were derived, please see https://darrenyao.com/usacobook/cpp.pdf#page=60.

E Gluing Pluto Back Together

Author: Bennett

We can solve this problem by trying all orderings recursively, but this is too slow, with 12! = 479001600 orderings. However, we can speed this up significantly with two optimizations.

First, we can note that since we are building a ring, we can begin with chunk 1 first, without loss of generality. This reduces us to only checking 11! = 39916800 orderings.

Secondly, we can track the cheapest ordering and prune if we exceed this cost. With these two optimizations, we significantly reduce our search space and can bash this in time.

A harder version of this problem with a higher N would require a bitmask DP.

F Plutonian Hot Dog Stand

Author: Yu

Observe that if we distribute a discount ticket to the i^{th} Plutonian buying M_i hot dogs, and the $i-1^{th}$ Plutonian is buying $\geq M_i$ hot dogs, then we might as well give the ticket to the Plutonian before.

For the j^{th} Plutonian, if we can find the longest sequence of consecutive Plutonians that are buying $\leq M_j$ hot dogs, then we can distribute the D tickets to the D Plutonians with the longest non-overlapping sequence to get an optimal distribution.

To find the sequence for each Plutonian, we can sort the Plutonians in order based on the number of hot dogs M_i each is buying in decreasing order. For the i^{th} Plutonian we can use a Fenwick Tree to keep track of which Plutonians have already been processed and binary search rightward for a position j from position i in the line until the sum from $[i, j] \neq 0$. We store this position somewhere and then mark the position as 1 in the Fenwick Tree.

Now that we have calculated the longest sequence length from each Plutonian, we can sort the Plutonians based on their sequence length in descending order. We also maintain a visited array to keep track of which Plutonians have been visited. If the current Plutonian has been visited, we skip it. Otherwise, we mark all the Plutonians in the current sequence as visited and repeat the process until we have processed D sequences.

Finally, we can loop through the visited array and simply count the number of Plutonians visited as our answer.

G Path to Pluto

Author: Dylan

First, notice that the graph given in the input is a tree. Before adding an edge, we can calculate the shortest distance of each node to node 1 by simply finding the length of the each path to the root, since there is only one simple path between any two vertices in the tree. Note that a non-simple path can never be optimal since all edge weights are positive, so any cycle would have a positive sum. To find the path lengths to the root for each vertex we can take note of the fact that the path length of a vertex is equal to the path length of its parent plus the edge between the two. Thus, we can iterate over nodes from highest to lowest, and determine each path length this way.

Now we must deal with the problem of adding an edge. One important observation is that it is always optimal to place on endpoint of the edge on the root. To prove this, lets assume that the optimal choice of edge placement does not connect to the root. Denote u as the endpoint of this edge which has a closer distance to the root in the original tree and v the endpoint which is further from the root. Also denote p_i as the parent of node i in the tree rooted at node 1. We know that there are no shortest paths that go from u to v because then they would need to reach node 1 from v which when added to the weight of the new edge must be greater than the distance from u to the root. If we replace u with p_u, then the only shortest paths that change are the ones that originally travelled on the edge, and they now have less distance to travel, thus decreasing the sum of all travel costs. So after repeating this process it is clear that one endpoint of our new edge must be at the root if we want it to be optimal.

We only have $n-1$ options for where to place the endpoint of the edge which isn't at the root, and we can think about how much each edge will decrease the sum of costs. I'll refer to u and v here again. All

vertices in the subtree of v will have its distance decreased by $dist[v]$ if we add the edge. Thus we want to maximize $dist[v] * cnt[v]$ where $cnt[v]$ is the number of vertices in v's subtree. We can iterate from bottom to top on the tree to find $cnt[i]$ for each i, and then we have everything we need to determine the answer: just take the maximum out of all i.

H Planetary Observations

Author: Neo

First, notice that if we add characters to string O, the smallest period of the string can only increase. Thus, lets store the current period of the string as we process the queries as k. Whenever we add characters to the string, we can keep incrementing k while it is not valid anymore. Eventually we will each a valid value of k since $k = |O|$ is always valid. To check if a given k works, we iterate over every kth substring of length k, which can be done easily be calculating a hash for every prefix of the string (which we can do in parallel with processing the queries). Since we can only increment k, we are essentially checking each k once, and the sum of $1/i$ for all i from 1 to N is about $N * log(N)$. Another case to note is that we could be checking the same k many times, but to make sure this isn't costly we can recall that all substrings of length k that weren't added in the current query were already valid, so we only need to process the new ones.

I Wormholes

Author: Colin

This problem boils down to finding two disjoint shortest paths in the graph. However, this can be hard to approach all at once. Instead, let's think about a version where all edges have cost 0. Then, this problem is just finding two disjoint paths from the start to the end, which is equivalent to checking if the flow of the graph is exactly 2! We can just run any of our favorite max flow algorithms for this.

To generalize to edges with weight, we need to combine some ideas of max flow and shortest path. We can borrow some intuition from how augmenting paths work — by adding backwards edges, we allow new paths to "undo" actions by old ones. This motivates a solution where we simply find any path from start to end, and reverse all the edges taken by that path (with negative edge weights). Then, finding the shortest path in this new residual graph will be equivalent to constructing two disjoint paths.

J Rocket Fuel

Author: Aaryan

Suppose $a = 1$ for each of the queries, i.e. we want to compute the fuel used by engines ql through qr up until time b. We can sort queries by time, so we process all the queries at time t, process the update between t and $t+1$, then process the queries at time $t+1$, etc. To keep track of the sum at each time, we can use a lazy segment tree. Each engine can be represented by a linear function, where $mx + b$ is the amount of fuel used at time $t = x$ for some m and b. m is the current amount of fuel the engine needs per time step, while b is some constant. If we update $mx + b$ at time t, increasing the slope by d, then $(m + d)(x - t)$ is the linear term of the new function and $mt + b$ is the constant term of the function. Overall, we get a new function $(m + d)(x - t) + mt + b = (m + d)x + (b - dt)$. In other words, we update the function by adding d to the linear term and adding $-dt$ to the negative term.

With the above method, we can get answer the queries $f(ql, qr, b)$. To answer the sum between times a and b, we can answer $f(ql, qr, b) - f(ql, qr, a - 1)$ to get the answer.

link