Welcome On Board

We can first read in the Crew member's name, then print what the statement asks for.

Time complexity: O(1)
Problem Credits: Daniel Wu
Code: https://ideone.com/VWjLKF (Daniel Wu)

Astronomical Calculations

First, we should read in v, r, and f. Then, we should find a using the formula a = v^2/r. Then, we can find m by taking the formula f = m*a and dividing both sides of the equation by a to get m = f/a.

Time complexity: O(1)
Problem Credits: Daniel Wu
Code: https://ideone.com/DNZjZ4 (Daniel Wu)

Orgs and Borgs

Make a boolean variable for whether the planet is habitable for the Orgs and for the Borgs separately. We can compute the variables by checking whether the planet temperature is less than or equal to each alien's maximum temperature and greater than or equal to each alien's minimum temperature. Then, we can check all 4 cases on these boolean variables and print what the statement asks for in each case.

Time complexity: O(1)
Problem Credits: Daniel Wu
Code: https://ideone.com/WMHAxU (Daniel Wu)

Alien Interpreter

There are of course ways to do this problem in other languages, but the .split() operation in Python was very helpful. After splitting the string along the hyphens, we can put the remaining strings in groups of two. Within each group of two, we can form the original word using an if-statement on whether the second word was “org” or not. We can put together the results for all word-pairs to get the original English message.

Time complexity: O(|S|)
Problem Credits: Daniel Wu
Code: https://ideone.com/NnBxaQ (Daniel Wu)

Galactic Council

Make a list for all of the aliens in each species. You can brute force all combinations of one alien from each list. The easiest implementation was making a boolean variable for each species to store whether they had the most of something, and using a for loop to go through each body part. Then, you can check whether all of these variables are true and add 1 to a counter.

Time complexity: O(N^3)
Problem Credits: Daniel Wu
Code: https://ideone.com/gvZsbu (Daniel Wu)

Constellation

Find the two stars x and y that are farthest away from each other. One of them is the tip and the other is the end of the handle. Then, find the number of stars that are the minimum distance from x and the number of stars that are the minimum distance from y. The tip is guaranteed to have two points that are minimum distance from it, while the end of the handle is guaranteed to have one point that is minimum distance from it.

Time complexity: O(1)
Problem Credits: Daniel Wu
Code: https://ideone.com/86IDPo (Daniel Wu)

Ores

Iterate through all pairs of left and right borders for the subrectangle. Then, assign each row a type - MB, MI, MT, BI, BT, or IT based on which elements are on the borders at that row. Rows that have two of the same element at the borders can be ignored. Also note that having “b”, and “m” is the same as having “m”, and “b” on the borders of a row, and they should both be added to type MB (and the same for the other types).

We should then match up MB with IT, MI with BT and MT with BI to count the answer for that left and right border pair. For example, we add (the number of MB) * (the number of IT) to the total.

Time complexity: O(N^3)
Problem credits: Julian Kovalovsky and Daniel Wu
Code: https://ideone.com/Iht4Qx (Daniel Wu)
https://ideone.com/JMvmtI (python)

Moon Trip

If c*n < p, then it is clearly impossible.

Once we have chosen which rocket ships to use, we should greedily assign people to ships in increasing order of b. Thus, we should have at most one ship at less than full capacity.

Let f(i) := a[i] + c * b[i]. This is the cost of sending a ship at full capacity. Sort ships by f(i). When we say cheapest from now on, we mean cheapest by f.

If p is divisible by c, then we should use exactly p/c ships, all running at full capacity. Greedily choose the cheapest p/c ships. Otherwise, let k = floor(p/c). We should now send k ships at full capacity with one additional ship running at (p%c) capacity.

Let x be the partially-filled ship. Now, what are the ships that we will send at full capacity?

- If x is not among the cheapest k ships, we should greedily send the cheapest k ships at full capacity.
- If x is among the cheapest k ships, we will still send all of the other cheapest k ships that aren't x, but we will also send the (k+1)th cheapest ship at full capacity.

Then, if we precompute the sum of costs over the cheapest k entries of the array, we can simply iterate over x to check all possibilities.

The time complexity is O(n* log n) for sorting
Problem credits: Daniel Wu
Code: https://ideone.com/82KNyn (Patrick Deng)

Asteroid Belt

Notice that the answer can never be more than sqrt(2*n) because partitioning into i subarrays requires the array to be at least 1 + 2 + … + i length.

Now, how should we find the answer for just the entire array? We can construct the partition from right to left. If we are on our ith subarray, we should keep extending the subarray to the left, until the subarray obtains i distinct elements, at which point we should greedily cut it and start the next subarray.

After this process, the number of distinct elements in each subarray will be decreasing from left to right except for the leftmost subarray, which we will have to delete and add to the next subarray to the right. We can keep track of how many distinct elements are in a subarray while we add and delete elements by additionally a count of how many of each element are in each subarray.

Now, how do we delete elements off the end, in order to answer every prefix? Just like the common two-pointers technique, all of the subarray endpoints are monotonic so we should just move them into each other following the same process as before to maintain the property. First, we move the left endpoint of the first subarray, then because the right endpoint of the second subarray (aka the left endpoint of the first subarray) moved, we also have to move the left endpoint of the second subarray, and so on. There are O(sqrt n) pointers and each pointer moves at most N times.

The total time complexity is O(n * sqrt n)
Problem credits: Daniel Wu
Code: https://ideone.com/W0ujqI (Daniel Wu)

Bored Game

Let x := sum(depth[i]) for all tokens i + (# of tokens)/2. The observation is that after someone takes a turn, x is guaranteed to change parity.

Proof:

- If the player chooses to move a token, the depth of that token either increases by one or decreases by one. The # of tokens doesn't change, so in total this operation will flip the parity of x.
- If the player chooses to delete two tokens, then (# of tokens)/2 decreases by one, and sum(depth[i]) decreases by an even number because the depth of the two tokens was the same. Thus, the parity of x flips.
- If in addition to the above, the player also chooses to delete an edge, x doesn't change.

If x = 0 when it is your turn, then you lose as a base case. Therefore x being even means you lose. We should thus just calculate the parity of x with a DFS at the start, and if it is even then the Orgs win, and if it is odd then the Borgs win.

Because of the edge deletion, if x is odd on your turn, then you can always "shrink the game" by edges or tokens on your turn. Therefore, the game will always end.

Time complexity: O(N)
Problem credits: Daniel Wu
Code: https://ideone.com/wxG9U4 (Daniel Wu)

Satellite

Let left[i] be the nearest j when going cyclically left such that a[j]>=a[i]. Let right[i] be the nearest j when going cyclically right such that a[j]>a[i]. Perhaps the easiest way to compute these arrays is with a set. We can put elements into the set in order of size, and when we place an element in, look at the closest elements to the left and to the right that are already in the set. Cyclicity can be handled by putting in each value twice, at i and at n+i.

Element i contributes to the answer when left[i] has cycled so it is to the right of i. This makes a contiguous interval of rotations based on i and left[i]. It contributes a[i]*(right[i]-i) to this interval. Use a difference array to recover the answer for each rotation and report the minimum.

Note: my implementation uses priority queue instead of set to compute left and right arrays.

Time complexity: O(n * log n)
Problem credits: Daniel Wu
Code: https://ideone.com/jbqa2w (Daniel Wu)

Potatoes

The answer is equal to the sum of all range products. This is because the expected value of the yield on each day i is equal to pr(that days i..i) flourish + pr(days i-1..i flourish) + pr(days i-2.. I flourish) + .. + pr(days 1 through i flourish).

Now, we will store a segment tree where if a node stores the segment [l,r] then the node should store the following quantities:

- product(a[i]) for l<=i<=r
- sum(a[i]) for l<=i<=r
- sum(product(a[i]) for all x <=i <= y) for all l <= x<= y <= r
- sum(sum(a[i]) for all l<=i<=k) for all l<=k<=r
- sum(sum(a[i]) for all k<=i<=r) for all l<=k<=r

This can be understood as the sum of the segment, the product of the segment, the sum of all range products on the segment, the sum of prefix sums on the segment, and the sum of all suffix sums on the segment. Armed with all of these values, we can also merge them by:

- Range product = left range product * right range product
- Range sum = left range sum + right range sum
- Sum of subrange products = left sum of subrange products + right sum of subrange products + left sum of suffix sums * right sum of prefix sums
  (This last addend comes from all of the range products that bridge the middle, and can be derived with clever use of how multiplication distribution over sums works.)
- Sum of prefix sums = left sum of prefix sums + left segment sum * right segment size + right sum of prefix sums
- Sum of suffix sums = right sum of suffix sums + right segment sum * left segment size + left sum of suffix sums

Thus, nodes can be easily point updated and merged with the standard segment tree. The answer is the sum of subrange products for the root node.

Time complexity: O(n + q * log n)
Problem credits: Daniel Wu
Code: https://ideone.com/zkNK4E (Daniel Wu)

Mystery Alien

Let's compute two things for each node i: 1) dist[i] := the distance to node 1, and 2) next[i] := the index of the earliest sighting that a shortest path from i to 1 can go through. These can be computed with djikstra's algorithm on the reverse graph, where the elements of the priority queue are ((distance, earliest sighting through), node). This way, earlier sightings will pop out first, all else being equal.

For each sighting s, next[s] must be the index of the next sighting, otherwise the answer is -1. Now, from each sighting node, we should find the heaviest edge s -> x such that x is ether the next sighting or next[x] is the index of the next sighting, and where taking edge s -> x can still be a shortest path from s to 1 (if dist[j] = weight(j,x) + dist[x]).

Based on the distance to 1 and the heaviest edge from each sighting, we can find the upper and lower bounds of the distance from 1 for a possible answer. Also, all answers i must have next[i] = the first sighting.

Time complexity: O((n+m)* log n)
Problem credits: James Attis
Code: https://ideone.com/z7v46B (Daniel Wu)

Lazer Signals

Let DP[i] be the quickest way to type S[i…n-1] if one lazer is on S[i-1], and the other is on S[i]. Let d(i,j) = abs(a[i] - a[j]). Let dist(i,j) be the number of seconds required to type S[i…j] with one lazer. We can calculate dist using O(N) precomputation with prefix sums.

Now, DP[i] = max(dist(i,j-1) + d(i-1, j) + DP[j]) for all indices j>i.

This corresponds to casework the next time the two lazers are used for consecutive elements in a. The lazer on S[i] is used for all elements up to j-1 and then the lazer on S[i-1] is used for element j. We can break up dist into the parts made up by the prefix sums. This gives us:

DP[i] = min(pref[j-1] -pref[i] + DP[j] + abs(a[i-1]-a[j]) for all indices j>i.
Dp[i] = min(pref[j-1]+ dp[j] - a[j]) - pref[i] + a[i-1] for all indices j>i, a[j] <= a[i-1]
Dp[i] = min(pref[j-1]+ dp[j] + a[j]) - pref[i] - a[i-1] for all indices j>i, a[j] > a[i-1]

This allows us to isolate the effects of i and j separately. We can make range min queries using two segment trees: one that stores values of pref[j-1]+ dp[j] + a[j], one one that stores pref[j-1] + dp[j] - a[j]]. The leaf values of the segment tree will be the dp values in sequence-compressed order of a. This lets us query the two cases while only looking at the dp values that satisfy our a[j] constraints for each case. We will calculate dp values going right to left.

Time complexity: O(n*log n)
Problem credits: Daniel Wu
Code: https://ideone.com/07IBsT (Daniel Wu)

Wormholes

First of all, we can basically ignore distances. The answer for each query is d + the amount of waiting time required. From each wormhole, we should also subtract its position from the wormholes starting and ending times so that we can compare when in distance-adjusted time events will happen.

Let par[i] for wormhole i be the first wormhole that you have to wait at, after "being released" from wormhole i. Let wait[i] for wormhole i be the amount of time they have to wait at par[i] before they get released by par[i]. Once you are released from a wormhole, the time in the cycle and your position is always the same, so you always get stuck on the same wormhole next.

We can compute par[i] for all i using a set. At each wormhole i, we look at all previous wormholes that got stuck at a distance-adjusted time that is in the range at which we trap people, and set the parents of all those wormholes to be i, and then delete those wormholes from the set.

Now, we have an auxiliary forest. For each query, we should find the first and last wormhole it gets trapped at. We can do this offline if during our for loop above if ,once we have reached the wormhole that corresponds to a queries left endpoint, we also add in the location-adjusted times of each query to the running set. Now, during the for loop, when the current wormhole looks for all past wormholes that finished in its range, it should also look for and delete all queries that started in its range. For these queries, the first wormhole that the gets stuck at is the current wormhole.

When wormholes get merged into their parent, we can use DSU to access the current root of any past wormhole. Once we reach a query's right endpoint, we use the union_find to find the wormhole that they most recently got stuck at. This is the last wormhole that they get stuck at.

We can find the amount of waiting for each query as a node-ancestor path sum where edge weights are the weight[i] values. Now, we can compute the sum of the path from each node to its root using a dfs. This lets us find node-ancestor paths by taking the difference of (the node-root sum for when the query gets on) and (the node-root sum for when the query gets off).

Time complexity: O((n+q) log n)
Problem credits: Daniel Wu
Code: https://ideone.com/s5Xp0b (Daniel Wu)

Terraforming

dp[i][mask] := number of ways to do the last i rows such that exactly the cells represented in bits of mask are water cells in row i.
submasksum[i][mask] := sum(exact[i][mask']) for all submasks mask' of mask

Once we compute dp for a row, we can compute submasksum using SOS dynamic programming as in https://codeforces.com/blog/entry/45223.

Now, how do we compute dp[i][mask] for all masks? We know that for all streaks of 1's in mask, we need at least one 1-bit in the row below so that the water in that streak can flow out. Let's complementary count. Now, we want the bitmasks that have all 0's in at least one of the streaks in mask. Say there are t streaks, and let s_j be the subset of bitmasks that have all 0-bits in the jth streak of mask. We can want the sum of DP[i-1] over the union of (s_1, s_2, … s_t).

Let's use principles of inclusion and exclusion to calculate the sum over a set-union. We can iterate over all subsets (including the empty set) of (s_1, s_2, … s_t) and ask, what if at least these streaks have all 0's underneath them? This corresponds to the sum of dp[i-1] over all submasks of some bitmask (in particular, the bitmask is (2^m - 1) - sum(bitmask(str)) for all 1-streaks str in the subset). Luckily, submasksum[i-1] already directly stores this sum. We then multiply this sum by (-1)^(parity of the size of the subset) and add this to dp[i][mask]. As for the grid constraints, we will manually set all bitmasks that violate the initial R,W cells to have dp[i][mask] = 0.

We can get rid of the bits on either side because if they exist they are guaranteed to flow out. This reduces the number of states for a nice constant factor optimization. For each bitmask, we can also precompute the bitmasks corresponding to each subset of streaks, so we don't have to do it when computing every state.

Note: Patrick Deng has an alternate solution that many competitors used, but requires a lot more effort with constant factor optimization to ac.

Time complexity: O(n*m*(1+sqrt(2))^m) (estimated by Ben Qi for reasons I don't understand) but the model solution runs in 952 ms.
Problem credits: Daniel Wu
Code: https://ideone.com/Fc7H7z (Daniel Wu)

mBOT

When I originally proposed this problem, I thought my solution was super cool and pretty hard to think of. Literally the day before mBIT, we realized that it can be killed by suffix tree, which made it fairly standard for people who knew about the data structure. Oh well. I still think this solution (while much slower) is still cool so I encourage you to read it.

A key observation is that the output string must be a suffix of S. For each query, we will find i such that mBOT's final output is the string S[i, n]. Call this the answer for that query.

First, build a suffix array of S (suff[i] is index of ith smallest suffix), and create equivalence classes for powers of 2 subarrays (equiv[i][b] is equivalence class of S[i, min(i+2^b-1)]). Use these to construct an LCP array (LCP[i] is the longest common prefix between stuff[i] and stuff[i+1]).

The set of all suffixes that match with any prompt forms a contagious interval of the suffix array. Every time we add a new character, the new interval becomes the longest subinterval of the current interval which shares a common next character. Now, all that is left is to efficiently narrow down our interval until it is of size 1. The following process halves the interval size, so we will repeat it O(log n) times to get the answer.

1) Find the largest K such that a subinterval of length at least c = ceil(interval size/2) shares an LCP of at least K.
    - Let's find the subinterval of length at least c that shares the largest LCP
    - The subinterval must contain the middle element
    - Iteratively construct the subinterval by greedily extending whichever endpoint leads to a larger shared LCP
    - First extend by ceil((c-1)/2), then by ceil((c-1)/4), and so on until you the subinterval reaches size c
    - Each time you extend, you know that the entire subinterval will either contain the entire left extension or the entire right extension, so you will have to bite one of the bullets eventually, so choose the best one now
    - K is the LCP of the subinterval
2) Find the longest subinterval [l,r] that shares the length K prefix.
    - Binary search for the furthest left and furthest right suffix array indices which share a length K LCP with the middle element
    - Use range mins of the LCP array to direct the binary search

- Since [l,r] constitutes a next-k-character majority for the current interval, it must constitute a next character majority for the interval brought about for k steps. Thus, [l,r] will become our new interval after k characters are added.

3) Within [l,r], find the longest subsubinterval [sl, sr] which shares a length K+1 prefix
   - This is equivalent to finding the longest subsubinterval for which the LCP array contains no K's because all LCP entries in [L,R] are >=k
   - Let lcpind[L] = {indices i such that LCP[i] = L}
   - We want to answer queries Q(l,r,k) := out of all elements in lcpind[k] that lie in a [l,r], what is the maximum distance between any two, or between one and a query endpoint
   - Create a sparse table on the indices in lcpind[L] that stores the distance to the next index
   - Answer queries Q(l,r,k) by binary searching for the greatest and least elements in lcpind[k] and taking range maxes on the sparse table
4) Reduce the original interval to [sl, sr]
   - [sl, sr] becomes our new interval after k+1 characters are added
   - [sl, sr] is guaranteed to be no more than half the size of the original interval

Now, all that is left is to find the original suffix array interval that matches the starting prompt. To do this, compute the equivalence classes of P in power-of-2 blocks and then binary search for the endpoints of the interval using the equivalence classes to compare.

Time complexity: O(n*log(n) + q*log^2(n))
Problem credits: Daniel Wu
Code: https://ideone.com/s8WD81 (Daniel Wu)