CerealCodes II Editorial

CerealCodes Team

August 2023

1 World's Hardest Math Problem

The statement tells us that $42 + 27 = 69$ is a desirable integer. As such, $69 - x$ will always be a valid solution.

Time Complexity: $\mathcal{O}(1)$

2 Palindromicity

Note that the number of differences when comparing a string to its reverse will always be even. Therefore if k is odd, there will never be an answer. If k is even, the following construction guarantees an answer: Fill the first $\frac{k}{2}$ positions of s with 0, and the rest with 1. Then the first $\frac{k}{2}$ and last $\frac{k}{2}$ positions will be different, adding up to k different positions total.

Time Complexity: $\mathcal{O}(n)$ per test case.

Time Complexity: $\mathcal{O}(a+b+c)$ per test case.

3 Reordering Red Pandas

Let $maxdist$ denote the maximum distance between any two pandas. Since $maxdist = p_n - p_1$, if x is a panda with $maxdist$ as one of its distances, x must lie at p_1 or p_n.

Knowing this, we can iterate over the pandas to find the desired panda x, and then sort the remaining pandas in order of increasing distance to panda x, d_{ix}. For both options p_1 and p_n for panda x's position, we can try placing pandas in the sorted order away from panda x, and check if the distances align with the assigned locations.

Time Complexity: $\mathcal{O}(n^2)$ per test case.

4 Allen's XOR(z)

For each bit b, let num_b be the number of elements of a that have bit b set to 1. Notice that the parity of num_b remains constant after each operation, meaning that num_b can become either 0 or 1 after enough operations. Practically, as long as $num_b >= 2$, we can repeatedly choose i and j $(1 \leq i < j \leq n)$ such that both a_i and a_j have the b-th bit turned on, and perform an operation with $x = 2^b$.

Once we reduce all num_b to 0 or 1, we notice that it is possible to "move" the b-th bit from element a_i to element a_j by performing an operation with a_i, a_j, and $x = 2^b$, where a_i has the b-th bit turned on, while a_j doesn't.

Now let M be the greatest bit such that $num_M = 1$. We claim the answer is 2^M. Since the sum of powers of 2 up to 2^{M-1}, $\sum_{p=0}^{M-1} 2^p$, is less than 2^M, we may "move" the M-th bit to some element a_i, and move all other bits b such that $num_b = 1$ to some other element a_j $(i \neq j)$, which always works since $n \geq 2$ is given. Then the greatest element in a indeed becomes 2^M.

Time Complexity: $\mathcal{O}(n \log A)$ per test case, where A is the maximal value of a_i.

5 Cascading Sums

Notice that if $x \geq y$, then $casc(x) \geq casc(y)$ holds as well, where $casc(a)$ means the cascading sum of a. This is because if any prefix of x were less than that same prefix of y, $y > x$ would have to hold, which is a contradiction.

Solution 1: We can use binary search to figure out the largest x such that $casc(x) \leq n$. Our answer would then be $n - x$. Calculating a number's cascading sum can also be done trivially, since we can iterate on all $\log x$ prefixes of x naively and sum them up.

Time Complexity: $\mathcal{O}(q \log^2 n)$

Solution 2: We actually don't need to binary search for the largest x; it is possible to build x digit by digit. Let's iterate on the digits in x's decimal representation from most significant (largest) to least significant (smallest). If we use the largest number possible for each digit while ensuring that $casc(x) \leq n$, we will have our answer.

The contribution of some digit d with i digits to the right is $\sum_{j=0}^{i} d \cdot 10^j$. For example, if we look at the hundreds digit of 1434, we get $d = 4$ and $i = 2$, which is a contribution of $400 + 40 + 4 = 444$. We can find the optimal d for each digit in constant time.

Time Complexity: $\mathcal{O}(q \log n)$

6 Cereal Trees III

We will call the "trees" from the problem "nodes" to avoid confusion.

Let us first solve the problem without considering the constraint of some node t_i having no cereal. We are asked to connect all nodes labeled from $1 \dots n$ with edges, where the weight of an edge between trees a and b is $a|b$, and the sum of edge weights is minimal.

It is both possible and optimal to use $n-1$ edges to connect all n nodes. Now we look at how to make our edges. An important property of the bitwise OR operator is that for any number x, applying the OR operator to x with another number will never decrease the value of x. In other words, the minimum value x can become after OR operations is x. Also, connecting every node means we have to draw an edge from each node, meaning each node x will contribute at least x weight to the total sum. Since we are minimizing, we want to connect each node x to another node that doesn't increase x.

We use the following construction. For each node $x \geq 2$, connect x to the lowest set bit in its binary representation. For example, $7 = 111$, so connect 7 to 1. Similarly, $10 = 1010$, so connect 10 to 2. Thus we can see that every node will connect to a power of 2. Nodes that are powers of two themselves are problems, because their least set bit is equal to themselves. In order to minimize the edge weight connecting these nodes, we connect them to node 1.

This construction will make sure that the entire tree is connected after $n-1$ edges and also minimizes the edge weight sum. Every node $x \geq 2$ that is not a power of two will contribute a weight of x, and every node $x \geq 2$ that is a power of two will contribute a weight of $x+1$. Therefore we can see the minimum sum of edge weights will be $\sum_{i=2}^{n} i + \log_2(n) = \frac{n(n+1)}{2} - 1 + \log_2(n)$.

Now we deal with the extra condition of removing a node. Actually, not much has to change. If the node t_i removed is not a power of 2, you can subtract t_i from the answer because you can just ignore the weight of the edge that would have connected t_i. If the node t_i is a power of 2, you can subtract $t_i + 1$ from the answer because the edge would have contributed that weight to connect to node 1. As for the construction in this case, every node x that is not a power of 2 has at least two set bits, so connect each x to the lowest set bit that exists in the tree. Therefore the answer for each test case is $\frac{n(n+1)}{2} - 1 + \log_2(n) - (t_i + \text{is } t_i \text{a power of } 2)$.

Time Complexity: $\mathcal{O}(1)$ or $\mathcal{O}(\log_2(n))$ per test case.

7 Mismatched Material

Notice that the only problematic positions in the given array a are $2 \leq i \leq n-1$ such that $a_{i-1}, a_{i+1} < a_i$, since that would mean $b_i, b_{i+1} < a_i$ and thus $a_i \neq max(b_i, b_{i+1})$. It is possible to fix all such positions with a single edit each by setting $a_i := max(a_{i-1}, a_{i+1})$, but can we do better? Notice that if we have another problematic position at $i+2$ $(a_{i+1}, a_{i+3} < a_{i+2})$, then we may correct both i and $i+2$ by setting $a_{i+1} := max(a_i, a_{i+2})$. So whenever we come across an "adjacent" pair of problematic positions, we can save one edit.

Time Complexity: $\mathcal{O}(n)$ per test case.

8 Panda-monium

Because pandas meeting at the root node is allowed, we can treat the subtrees of the root's children as independent problems and combine the answers later for the output.

Call a panda *active* if it has been freed. Within each of the root's children's subtrees, we have one restriction: we can never have have two *active* pandas at the same depth. Otherwise, those two pandas would eventually meet at some node which is not the root and then fight. Therefore, at every point in time, there should be at most one *active* panda at each depth in the subtree.

Now let's think of a bound for the minimum number of seconds to place the pandas. Ideally, we would free a panda at *each* depth in the first second, and then free one more panda each second after that, because the number of "occupied" depths decreases by at most 1 each second depending on whether a panda ascends to the root. Therefore, if the maximum depth in the subtree is d and the number of nodes is m, then we can free d pandas in the first second, and free the remaining $m - d$ with an additional $m - d$ seconds, for a total of $m - d + 1$ seconds.

We know the ideal number of seconds, but how do we actually find a way to free pandas in $m - d + 1$ seconds? Consider the following strategy: sort the pandas in increasing order of depth, and free in this order. That is, designate each panda its own second, but if whenever we see two adjacent pandas with different depths, we free them in the same second and subtract 1 from our total. Since we come across $d-1$ such pairs, the number of seconds to place the pandas is $m-(d-1) = m-d+1$, so it is optimal.

Time Complexity: $\mathcal{O}(n \log n)$ per test case.

9 Asteroid Trek

Solution 1: Since Jesse and Jerry will make a combined total of $n + 1$ moves, and each move will either be A or B, at each step we can try command B, check if it works, and if it doesn't, use command A instead. So now the difficult part is checking whether or not a sequence of commands can be completed such that Jesse and Jerry meet up. Note that when checking a possible sequence, we are no longer looking for the lexicographic largest, but instead whether it is possible or not.

If, whenever we move Jesse, Jerry is on the smallest asteroid that he could have feasibly reached, we will find a solution if it exists. Why? Well, if we have a sequence of commands that does not satisfy the aforementioned condition, an exchange argument shows that, if Jerry were at the smallest reachable asteroid every step of the way, Jesse's moves would still all be valid.

To simulate this, we can just keep track of Jerry's current position, the minimum position that he can reach, and the furthest position he can reach. Then, after moving Jesse one step, we will extend the furthest position that Jerry can reach and update the minimum as we go. Once Jesse is on the largest asteroid in the belt, we will just move Jerry left until they meet, and we will have found a good solution. If we ever get to the point where Jerry is currently on the minimum position in reach and Jesse is unable to move, it is impossible to construct a valid sequence of commands.

Time Complexity: $\mathcal{O}(n^2)$ per test case.

Solution 2: Participants are not expected to know Dynamic Programming since it is a Novice problem, but we include the DP solution here as well.

Let $dp[i][j] = true$ if having Jesse at asteroid i and Jerry at asteroid j allows them to eventually meet. Then we may start with setting $dp[i][i] = true$ for all i, and then transition as such, for all $i < j$:

$$dp[i][j] = s_i \geq s_j \ AND \ (dp[i+1][j] \ OR \ dp[i][j-1])$$

After calculating the dp, we can build the sequence of instructions from the beginning, always prioritizing moving Jerry left when possible. If Jesse is currently at position i and Jerry is at position j, then we move Jerry to $j - 1$ if $dp[i][j-1] = true$ and move Jesse to $i + 1$ otherwise.

Time Complexity: $\mathcal{O}(n^2)$ per test case.

10 Candy Machine

For node u, let p_u be the parent of u, k_u be the probability of that node being chosen from its parent, and t_u be the expected amount of money you need to spend to make u dispense one time. Then, we have $t_1 = 1$, and $t_u = t_{p_u}/k_u$ (since the amount of time it takes for (u, p_u) to be taken is a geometric random variable). The answer for node u is simply the sum of all t_v for v on the path from u to the root.

Time complexity: $\mathcal{O}(n \log MOD)$ per test case, where the $\log MOD$ comes from taking modulo inverses.

11 Jack-o'-Lanterns

There are many correct Dynamic Programming solutions to this problem. We will present a couple of them.

Solution 1: We would like to define a DP state that allows us to perform all 3 operations easily. Let $dp[i][j][k]$ be the maximum tastiness Envy can achieve if he is about to perform the ith operation, the rightmost illuminated pumpkin is at index j, and k is a boolean flag of whether or not the last pumpkin was a carved jack-o'-lantern.

Now both the eating and carving transitions are simple. If i is illuminated by j, we can set $dp[i+1][j][0] := max(dp[i][j][k] + a_i, dp[i+1][j][0])$ to simulate eating, and $dp[i+1][i+b_i][1] := max(dp[i][j][k], dp[i+1][i+b_i][1])$ to simulate carving. Notice that it is only useful to swap pumpkin i with pumpkin $i-1$ if pumpkin $i-1$ is carved with a jack-o-lantern. Thus, swapping can also be done by extending j by 1 if $k = 1$ by setting $dp[i+1][j+1][1] = max(dp[i+1][j+1][1], dp[i][j][k])$. Our answer will then be $max(dp[n+1][n+1][0], dp[n+1][n+1][1])$.

Time Complexity: $\mathcal{O}(n^2)$ per test case.

Solution 2: Alternatively, since the usefulness of a carved pumpkin helps us only "later" in the dp if we go left to right, this motivates us to simulate the process backwards, from right to left, for a simpler dp. Let $dp[i]$ denote the maximum tastiness derived from pumpkins $i \dots n$, and let $sum[i][j]$ be the sum of tastinesses of pumpkins in the range $i \dots j$, $\sum_{k=i}^{j} a_k$. Then we may iterate over pumpkins from index $i = n$ to $i = 1$, and try using pumpkin i to illuminate as far to the right as pumpkin j for each $i \le j \le n$. This allows us to eat pumpkins in the range $[\max(i+1, j-b_i), j]$, so the dp becomes:

$$dp[i] = \max_{j=i}^{n}\left(sum\big[\max(i+1, j-b_i)\big][j] + dp[j+1]\right)$$

Our answer is then simply $dp[1]$.

Time Complexity: $\mathcal{O}(n^2)$ per test case.

12 Pollination

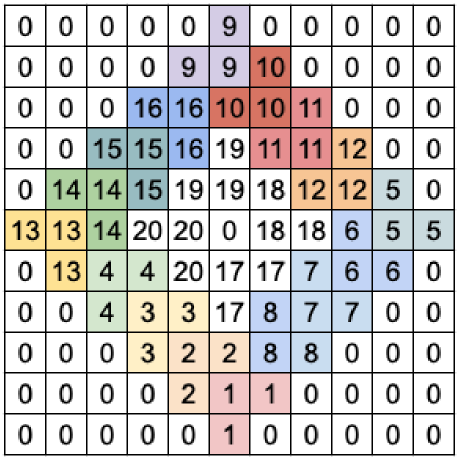
Let's first find how many squares are in a flower of size n. Ignoring the hole in the middle, we can see that a flower is two pyramids of odd numbers, one with a base of $2n+1$ and one with a base of $2n-1$. Since $1+3+5+\cdots+(2n-1)=n^2$, the total area of a flower is $n^2+(n+1)^2$. To compensate for the hole in the middle, we subtract one, getting our answer of $2(n)(n+1)$ squares.

We claim that a solution exists if $2(n)(n+1)$ is divisible by 3. Clearly, if it is not a multiple of 3, it is impossible to use tiles that have an area of 3 to tile the flower. $2(n)(n+1)$ is divisible by 3 if n is $0,2 \pmod 3$, and not divisible by 3 if n is $1 \pmod 3$.

Now we show a construction that will always tile the flower. First, we make a spiral with a set of L tiles as shown below. In fact, this will leave a smaller untiled flower in the middle! This new flower has a size of $n-3$, meaning the remainder upon dividing the side by 3 is the same. This means we can repeat this process for the next smaller flower that remains untiled in the large one. Diagrams are attached below of the first cycle and the completed coloring.

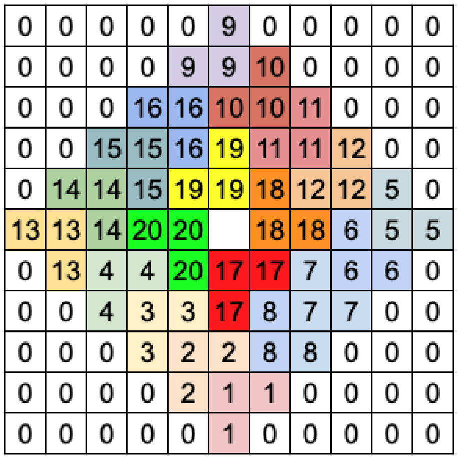
There are many ways to implement this solution, one clean implementation involves using complex numbers.

Time Complexity: $\mathcal{O}(n^2)$ per test case.



0	0	0	0	0	9	0	0	0	0	0
0	0	0	0	9	9	10	0	0	0	0
0	0	0	16	16	10	10	11	0	0	0
0	0	15	15	16	19	11	11	12	0	0
0	14	14	15	19	19	18	12	12	5	0
13	13	14	20	20	0	18	18	6	5	5
0	13	4	4	20	17	17	7	6	6	0
0	0	4	3	3	17	8	7	7	0	0
0	0	0	3	2	2	8	8	0	0	0
0	0	0	0	2	1	1	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0

Figure 1: Step one of Pollination construction



0	0	0	0	0	9	0	0	0	0	0
0	0	0	0	9	9	10	0	0	0	0
0	0	0	16	16	10	10	11	0	0	0
0	0	15	15	16	19	11	11	12	0	0
0	14	14	15	19	19	18	12	12	5	0
13	13	14	20	20		18	18	6	5	5
0	13	4	4	20	17	17	7	6	6	0
0	0	4	3	3	17	8	7	7	0	0
0	0	0	3	2	2	8	8	0	0	0
0	0	0	0	2	1	1	0	0	0	0
0	0	0	0	0	1	0	0	0	0	0

Figure 2: Finished Pollination construction

13 Friend Groups

For a pair of friends a, b, it is clear that the only edges they will impact are those that lie on the path from a to b. It may seem like this problem requires Heavy Light Decomposition with range set updates (although it can be solved this way, which is why this problem was only used in the Intermediate division), but that is not the case.

Notice that we want to know the minimum possible value of an edge, so we can process all friend groups in order from $1 \dots n$. This allows us to process each edge once: if we are currently processing the path from a_i to b_i, we want to find all edges on this path that don't have a value assigned to them yet, and set them to i. This can be done with a Disjoint Set Union data structure. Whenever we process an edge, we join the two nodes (meaning we never look at this edge again) and keep track of the node closest the root for each disjoint set, who will be the only node in the set that hasn't been joined with its parent.

To iterate on all unused edges from a to b, we will join the closest node to the root in a's disjoint set with its parent until we reach lowest common ancestor of a and b. Then we will do the same to b until a and b are in the same Disjoint Set. To know when to stop joining a's disjoint set with its parent, we can perform an Euler Tour on the tree and stop once a's disjoint set is an ancestor of b.

Time Complexity: $\mathcal{O}(n\alpha(n))$ per test case.

14 Aquamist

We will solve the problem in reverse, consider the final arrangement and making the initial one.

Consider each "color" separately. We will solve each color to create a stack of just that color and then exclude it from the stacks we are using ("solving" the color). We will then mark all colors we are solving for currently as red and all other colors indistinguishably as gray.

First, let's try solving in the case of $n = 3$. Let's make sure that two containers are full and one is empty (we will call the empty one the third stack). If one stack is completely red, we have finished, otherwise our goal will be to make the first stack red only.

We will do this by repeatedly moving at least one red from the second stack to the first stack (and one gray from the first to the second). Consider where the first gray is on the first stack and where the first red is on the second stack. If they are closer to the bottom of a stack, flip the stack upside down. now take the tops off of these stacks and put them on the third stack until the first gray and first red are visible (this cannot cause the 3rd stack to overflow because the tops are both smaller than $m/2$). Now take the first gray and first red and swap them by putting the gray on the third, the red on the first, and then the gray on second. now put the tops on both stacks. This procedure swaps a pair of gray and red blocks. We can keep doing this until all grays are on the second stack and all reds on the first.

Now we can extend this to solve for bigger values of n. First, make one arbitrary stack completely empty. Next, ignore all stacks with just gray in them. Then, take two stacks with at least some red in them both (if two dont exist, we are done). We can arbitrarily label gray blocks as red until we have exactly M red in this subset of two, and then also include the empty stack and, doing the $n = 3$ case on this subset. This will make the number of gray-only stacks increase by 1. We should repeatedly do this until n actually equals 3 when considering the non-gray-only stacks and then solve the final case for the reds. This process will be performed for each color.

To speed this up we should label one stack that we will always consider "red" and then merge reds into it as necessary, stopping to merge reds from a non-gray container once we have made it gray. The flip operations can be done in $2m$ commands, and everything else gets amortized to be under the bound.

Time Complexity: $O(n^2m + m^2)$

15 Roses

First notice that both Andre 3000 and Big Boi's choices form a functional graph. More specifically, since there cannot be cycles, both graphs are forests. If we have a start node with edges that lead to an end node, that's the same as the end node being an ancestor of the start node in that forest. Now our problem becomes equivalent to checking, on two forests, if there exists two nodes x and y such that x is an ancestor of y on both forests.

Let's call Andre 3000's forest G_1 and Big Boi's forest G_2. If we do a DFS on G_1, when we reach some node v, only ancestors of v are candidates for the pair of two nodes. These ancestors are specifically everything on the DFS stack at this point in time. Additionally the DFS stack only gets pushed to or popped from with every DFS call and return. Because the list of possible candidates changes very little between each operation on our DFS, we are motivated to use this to compute our answers. We should only start our DFS from root nodes in the forest (nodes with indegree of 0), because otherwise our DFS stack once we reach node v will not be the full list of ancestors of v.

When we are performing a DFS on G_1, we also want to consider the ancestors of v on G_2. If there is any node which is an ancestor of v on both forests, we have our answer. We can imagine that any time we put a node onto our DFS stack for G_1, we "activate" the respective node on G_2. Likewise, when we pop it off the stack, we "deactivate" the node on G_2. We will only have a solution if there exists at least one activated node on the path from the root to v on G_2. Since we have reduced our problem to doing fast (at most $O(\log N)$ for each of the $O(N)$ nodes we visit) node updates and path queries, we are motivated to flatten the tree with an Euler Tour and use a Segment Tree for path queries.

Specifically, we will want to do node set updates (set the nodes start and end times on the Segment Tree to 1 and -1 if activating or 0 and 0 if deactivating) and a path sum query to check if the sum of the path from root to v is greater than 0 (which means we have at least one activated node on this path).

Time Complexity: $\mathcal{O}(n \log n)$

16 Removing Subarrays

The important observation is that we only have to remove subarrays of size 2 or 3. To motivate this consider removing some subarray of size greater than 3 with a single operation. We can decompose that subarray into smaller ones of size 2 and 3 and remove those instead for a less than or equal cost.

With this observation we can solve the problem with range dp. Let $dp[l][r]$ be the minimum cost to remove subarray $[l, r]$ and infinity if it is impossible to remove the subarray. Let's consider the last operation we do.

Case 1: The last operation did not include **both** l and r. Then there must exist some splitting point i in which $[l...i]$ and $[i+1...r]$ are independent in the sense that no operation removes elements from both these intervals (think back to the fact that we are removing subarrays). Thus, we can iterate over the splitting point to transition.

$$dp[l][r] = \min(dp[l][r], \min_{l \le i < r} (dp[l][i] + dp[i+1][r]))$$

Case 2: The last operation was of size 3 and removes l, r, and some i $(1 < i < r)$. We can iterate over i to transition.

$$dp[l][r] = \min(dp[l][r], \min_{l < i < r} (dp[l][i-1] + dp[i+1][r] + 1))$$

Case 3: The last operation was of size 2 and removes exactly l and r. The transition is as follows:

$$dp[l][r] = \min(dp[l][r], dp[l+1][r-1])$$

This dp allows us to know which subarrays we can remove and the minimum cost of removing each subarray. To find the minimum size array we can get and the minimum cost, we must do another dp and use the calculated results from the range dp to help us.

Let $dp_2[i]$ store the following two elements as a pair: the minimum size array we can get if we only consider the first i elements (and pretend elements after i don't exist) and the minimum cost to do so.

To calculate $dp_2[i]$, loop over l and try removing $[l...i]$ for a cost of $dp[l][i]$. We must also consider the case in which we don't remove i.

$$\begin{aligned}dp_2[i] = \min(&\{dp_2[i-1].\text{first}+1, dp_2[i-1].\text{second}\},\\ &\min_{\substack{1\le l<i\\ dp[l][i]\neq\infty}}(\{dp_2[l-1].\text{first}, dp_2[l-1].\text{second}+dp[l][i]\}))\end{aligned}$$

Time Complexity: $O(n^3)$

17 Mark and Add

We can do square root decomposition on k.

$\boldsymbol{k > \sqrt{n}}$: There will be at most $\sqrt{n}$ contiguous segments that we add x to. One way to get these $\sqrt{n}$ contiguous segments is to keep track of adjacent marked elements that are a distance of more than $\sqrt{n}$ away.

$\boldsymbol{k \leq \sqrt{n}}$: Let's look at how we would handle updates of k, x naively. For each marked element i, we consider the nearest marked element to the left and right which we can call l and r. Then we add x to the interval $[\max(i-k, \frac{l+i}{2}+1), \min(i+k, \frac{i+r}{2})]$. To do this efficiently, we can keep track of these intervals for each $k = 1, 2, ...\sqrt{n}$ in constant time for each k. For some update k, x, we need to globally add x to the set of intervals belonging to k which can be done lazily. Specifically, for each k we keep a global variable storing the number we must add to all intervals belonging to k and we also tag along an offset to each interval. When a new interval is created, its offset is set to the negation of its global addition constant. So the true value that we must add to some interval is its global addition constant plus its offset.

Time Complexity: $O(q\sqrt{n})$

18 Vacation II

First, suppose that Ariel cannot be deported. For $1 \le x \le n$ et E_x be the event that Ariel visits city x. Also, let b_x be $\sum\limits_{i=x}^{n} a_i$.

We then have the following claim: $P[E_x] = \frac{a_x}{b_x}$.

Proof: suppose that Ariel is in state Y if he is currently at one of cities 0 through $x-1$, and in state Z if he is currently at one of cities x through n. Ariel will start his trip at state Y and end at state Z. Consider the moment Ariel transitions to state Z; if Ariel flies to city x, then E_x takes place, but if Ariel flies to city $w > x$, then E_x does not take place. We then have

$$P(E_x) = \frac{1}{1 + \sum\limits_{i=x+1}^{n} \frac{a_i}{a_x}} = \frac{a_x}{a_x + \sum\limits_{i=x+1}^{n} a_i} = \frac{a_x}{b_x}$$

Moreover, via the above proof we have that, for any $w \ne x$, E_w and E_x are independent, since $P[E_x]$ does not depend on the cities Ariel visits in state Y.

Now, consider the full problem. If cities l through r are at war, for each $k \notin [l, r]$, the contribution of E_k to the answer is $P[E_k]$. The contribution of the entire range $[l, r]$ is

$$\frac{\sum\limits_{i=l}^{r} \prod\limits_{l \le j \le r, j \ne i} P[\neg E_j] P[E_i]}{\sum\limits_{i=l}^{r} \prod\limits_{l \le j \le r, j \ne i} P[\neg E_j] P[E_i] + \prod\limits_{i=l}^{r} P[\neg E_i]},$$

which is the probability that a trip visits $[l, r]$ given the trip is successful.

When both the numerator and the denominator are divided by $\prod\limits_{i=l}^{r} P[\neg E_i]$, the expression simplifies to

$$\frac{\sum\limits_{i=l}^{r} \frac{P[E_i]}{P[\neg E_i]}}{\sum\limits_{i=l}^{r} \frac{P[E_i]}{P[\neg E_i]} + 1}.$$

Using prefix sums on $P[E_i]$ and $\frac{P[E_i]}{P[\neg E_i]}$, we can sum the contribution of every city in $\mathcal{O}(\log MOD)$ per query with $\mathcal{O}(n \log MOD)$ precomputation.

Time complexity: $\mathcal{O}(n \log MOD + q \log MOD)$ per test case.

19 Cereal Bushes

Because we are expected to be able to create a grid for every value of k within $0 \leq k \leq 10^{18}$, we might be motivated to make the construction of our grid analog to the construction of a number in some base b.

For example, in a (2 x 2) square of open cells, there are 2 paths from the top left cell to the bottom right cell. If we then attach another (2 x 2) square of open cells to the bottom right of this one so that their overlap is 1 cell, then there are $2 * 2 = 4$ paths from the top left cell to the bottom right. If we continue chaining (2 x 2) squares of open cells in this manner, each one multiples the number of paths through the grid by 2.

This might motivate us to write k in binary and then create a chain of i (2 x 2) squares of open cells for every bit i turned on in k. Unfortunately, we soon find that such constructions are impossible to fit within a (64 x 64) grid.

Plan B... Let's try base 10. A (3 x 4) rectangle of open cells leads exactly 10 paths from its top left cell to its bottom right. A chain of 18 of these rectangles gets us to 10^{18}, and has a bounding rectangle of size (37 x 55). Now we can build k. Each digit in k represents $a * 10^p$ where $0 \leq a \leq 9$ and $0 \leq p \leq 18$. So for each digit in k, we can lead exactly a paths from the top left cell of the grid into the top left cell of the p-th to-last (3 x 4) rectangle in our chain. The particulars are of course, implementation dependent, but here is one example of this construction:

20 Purchasing Cereal

First, let's consider the case where the tree is just a line. In this case, we can consider the minimum cost for every amount of cereal that is purchased. We observe that this reduces to a piecewise convex function, where each piece is a quadratic function. When we take a flight, we are essentially adding $ax^2 + b$ to every segment of this piecewise function, and when we consider buying cereal at a node, we are cutting the function off whenever the distance is greater than c, and adding a new segment up to infinity that is a line with slope c. Now, when we query for the answer, we can answer each query as we reach the node, and then we just need to binary search within this function to figure out which segment it is in, and then answer the query. The time complexity here is $O(n \log n + q \log n)$, however the important thing to note is that this is actually amortised: the solution requires deleting all of the useless nodes, which breaks a rollback/persistent solution on a tree.

Now, we consider putting it on a tree. We can store a similar data structure on the tree. However, to get around the amortization aspect of the data structure, we can use another trick. To get around deleting all of the irrelevant segments of the data structure, we can think of storing a pointer to the last segment that is still relevant. By doing this, we have converted this piecewise convex hull into a tree-like data structure. This motivates using binary lifting: if we consider the last segment that is still relevant to be the parent of the new node, then we can store the 2^kth parent for every node and every k. Now, when we query a node, we can use this structure to binary search for the right segment: if the 2^kth parent's segment starts after the query point, we can recurse onto that parent, otherwise we go down to the 2^{k-1}th parent. With this, each query takes $O(\log n)$ time with $O(n \log n)$ preprocessing, so the total time complexity is $O(n \log n + q \log n)$.

21 Magical Zoo

Root the tree. Split each path into 3 parts:

1. Start to LCA(start, end)
2. LCA(start, end) to the first occurrence of a feeding station on the path to end
3. First occurrence of feeding station to end

For the path of first type and second type, we can just keep track of counts of each color, and use small to large merging. When we reach a feeding station, we just merge all counts into one. For each path, to keep track of counts, we add one to its original color at its bottom endpoint. Then, at its top endpoint, we check which color it would become (with binary lifting) and subtract one. Note that the ”original color” for type one path is 0, while the ”original color” for type 2 paths is the color the panda becomes after reaching the LCA. For paths of type 3, we merge all of them into a single count variable, representing the first feeding station you reach if you keep moving up towards your parent.

Note that small to large merging with a map is $O(n \log^2 n)$, which would probably TLE. Therefore, implementations should either use a hash table or a shared global array.

Time Complexity: $O(n \log n)$

22 Date

The condition for node a to be reachable from node b on day d is that all nodes on the path from a to b have a an opening time that divides d. Another way to put this condition is that the least common multiple of all opening times of nodes on the path from a to b must divide d. Furthermore, if node a can reach node b and node a can reach another node c, then node b can also reach node c. Thus, we can move are starting node to any node that it can reach on the given day without changing the answer.

With these observations, we can apply centroid decomposition to solve the problem. Lets first assume that Kana will not block any nodes. To solve this subproblem, we can find the highest ancestor of the starting node on the centroid tree that is reachable on day d, call this node p. Then, the answer is just the sum of all values on nodes in p's subtree of the centroid tree that can reach p. We can easily store this by finding the least common multiple of all opening times on the path from each node to p and then checking all divisors of d to find which nodes are reachable. Finding the least common multiple for each node can be done during the centroid decomposition. To find the answer even with Kana blocking a node, we can subtract out a subtree's sum of value using prefix sums and a Euler tour traversal found during the centroid decomposition. Note to account for Kana's blocked node when finding the highest reachable ancestor. Implementation can be done offline either using a hash map or with some careful implementation using prefix sums for a time of $\mathcal{O}(\text{maximum \# of divisors of } d)$ per query. However, using a time complexity of $\mathcal{O}(\text{maximum \# of divisors of } d \cdot \log n)$ per query will also pass and may result in easier implementation.