,

Problem A. Sort and &

The minimum cost of sorting the array will be either 0 or 1. If the array size is $2^k - 1$ (for some k) and the last position does not contain the correct number then the answer is 1. Otherwise 0.

Let's analyze the case when the array size is not $2^k - 1$. We will try to put the numbers into the right position one by one from the end. Suppose N is at the position a and there must be a zero bit in a, say that is at x. Then with 0 cost we can swap numbers between the positions of 2^x and a. Similarly there must be a zero bit in N, say that is y. If $x = y$ we do not need to do anything, otherwise we can swap the positions 2^x and 2^y. And finally we can swap the positions 2^y and N. This way we can bring the correct number to the last position of the array.

It's not difficult to see that, when the original array size is $2^k - 1$ and we need to bring the correct number to the last position, it would take at least 1 cost.

This solution does not take more than $3N$ swaps.

Problem B. Gona Guni

Instead of counting $sum(the\ minimum\ vertex\ cover)^m$, we can use the ordered pair technique. This insight leads to solving a different problem: select a set of vertices of upto m size, in how many different subsets of vertices of the tree these vertices are guaranteed to be selected as the minimum vertex cover. Then we count the sum over the sizes of all sets of vertices.

What we really want is given a set of vertices from the minimum vertex cover, can we get back the original sub-tree. That way we can generate all possible subtrees and in turn count the subset of vertices that will contain this given set as minimum vertex cover. This turns out to be difficult to do, unless you consider the maximum matching instead of minimum vertex cover and always assign the vertex closer to the root (of the whole tree) as the cover vertex.

This approach can lead to a bottom up sibling dp solution. From each node we need to calculate 2 things, the number of ways to select $k \leq m$ maximum matching with the subtree below the vertex and the sibling vertices with the current vertices matching with one of the children or the current vertices becoming available for matching with the parent. Merging the count from the children and the siblings requires convolution, thankfully with some small optimization we can avoid doing fft.

This dp calculation is a little tricky, as you need to think about what happens in a lot of situations. Like, whether to cut off the edge from the parent and remove the subtree rooted at the current vertex, when the current vertex is matched with a child, do we count it as part of the k set or we don't, is the vertex included in the subset or not.

At the end, you will have the count of selecting exactly k different vertices and the number of subsets where they will be guaranteed to be a part of the minimum vertex cover, from there we can calculate the number of ordered lists of size m from k vertices (from the ordered pair technique). The sum of them is the result.

To avoid doing fft, we can do this trick: instead of multiplying $m \times m$, we avoid the suffix with 0s. So, we find the last non-zero position of both arrays, say non_zero_a and non_zero_b, so we multiply two arrays of size $non\_zero\_a \times non\_zero\_b$. We can also stop multiplying if the multiplication will be added to any position beyond m.

Problem C. Packet Transmission

Every query we receive can be classified in the following way.

- The query packets (source, destination pair) share a common path in the tree.
  - The packets are going in the same direction. In that case we need to assume one of the packets will never wait. So, the other packet will arrive at the first vertex of the common path, wait for the first packet to arrive and then go after it. Since an edge can't be used by multiple packets

at the same time, the other packet will have to wait until the first packet crossed that edge. We need to keep doing that until they reach the end of common path and then they can go their own ways. If we think carefully, we can see that the other packet will have to wait some additional time which is equal to the maximum edge cost on the common path. However if the first packet comes early then the wait time reduces. Do the same the other way around and take the minimum of the two.

- The packets are going in different directions. So, we need both packets to start their journey. Eventually they might meet on the two sides of an edge where if one starts crossing, the other must wait. Make one of them wait and the other one go and calculate the time it would take for both them to reach their destinations. Do it both ways and take the minimum time. However if they don't meet at the edge then they don't have to stop and their actual time to reach both their destinations is the answer.

- The query packets use completely different paths. In that case the time it takes for both to reach their destination is the answer.

We can calculate all of these using the Sparse Table to calculate Lowest Common Ancestors in a tree. Alternatively we can use heavy-light decomposition, but that may be very slow.

Problem D. Qwiksort

We can get inspiration from bubble sort. When n is even, we will sort following segments in order:

- $[1, n]$
- $[\frac{n}{2} + 1, n + \frac{n}{2}]$
- $[n + 1, 2n]$
- $[1, n]$
- $[\frac{n}{2} + 1, n + \frac{n}{2}]$
- $[1, n]$

When n is odd, we will need some extra operations, but we will not require more than 8 operations.

Problem E. Travel on the Grid

Let's say each mine only blocks its own cell. Then we can use a shortest path algorithm with the states (x, y, b) where $b = 1$ if that cell has a diffuser (either by carrying or placing). However, in our problem each mine blocks its four adjacent cells as well. We need to handle that. If we have somehow reached a cell with a mine, then that cell must be diffused and so we can visit any of the neighbors without worry. It turns out that this is enough to solve the problem!

Let (x, y) be the cell we are currently in and (x', y') be a neighboring cell. We simply run a shortest path algorithm over all states (x, y, b) with the following transitions.

- If (x, y) and (x', y') are both safe
  - $(x, y, 0) \to (x', y', 0)$ (no cost)
  - $(x, y, 1) \to (x', y', 1)$ (cost V_{xy})
- If (x, y) is safe and (x', y') has a mine (diagonal transition)
  - $(x, y, 0) \to (x', y', 1)$ (cost X)
  - $(x, y, 1) \to (x', y', 1)$ (cost Y)

,

- If (x, y) has a mine and (x', y') is safe
  - $(x, y, 1) \to (x', y', 0)$ (no cost)
  - $(x, y, 1) \to (x', y', 1)$ (cost V_{xy})
- We can't go to (x', y') if that is not a safe cell.

Clearly, these transitions are valid. But it turns out these are sufficient. Why? We can make a few observations.

1. If we diffuse a mine, we can assume that we move onto its cell next move.
2. We can assume that we only build a diffuser right before using it on a mine.
3. If we visit a cell with a diffuser, we will never visit it again.
4. If we visit a cell without a diffuser, we can only visit it again with a diffuser.
5. If we ever visit a blocked cell neighboring a bomb, we can assume that we visit it straight after diffusing the said mine.
6. We only ever visit at most one neighboring cell of any mine.

The time complexity is $O(m \cdot n \cdot \lg(m \cdot n))$

Problem F. Yet Another Crossover Episode

We know that $x\&y \leq \min(x, y)$ and it looks like $\gcd(x, y) \leq \min(x, y)$.

So, it looks like $\gcd(x \oplus y, x\&y) \leq \min(x \oplus y, x\&y) \leq \min(x, y)$.

Also notice that i can be equal to j, so it looks like the answer is $\max(a_1, a_2, \ldots, a_n)$ because this is the maximum possible gcd.

But this is wrong due to the case when $x = 0$ or $y = 0$ because then $\gcd(x, y) = x$ when $y = 0$ and $\gcd(x, y) = y$ when $x = 0$.

So, when $a_i \oplus a_j = 0$, we have $a_i = a_j$, so $\gcd(a_i \oplus a_j, a_i\&a_j) = \gcd(0, a_i\&a_i) = \gcd(0, a_i) = a_i$. So, for this case we need to take care of the maximum number of the array only.

And when $a_i\&a_j = 0$, we have $a_i \oplus a_j = a_i + a_j$ because there is no common bit in a_i and a_j so xor and addition are the same in this case.

So we need to find the maximum $a_i + a_j$ where $a_i\&a_j = 0$. So let's fix a_i and then find the maximum a_j such that $a_i\&a_j = 0$ which means we need to find the maximum a_j such that a_j is a submask of the flipped bits of a_i.

Finding the maximum submask of a mask that exists in the array is a standard problem that can be solved using Sum Over Subsets (SOS) DP. We can also find the count of such submasks using the same DP.

So overall we need to consider the maximum element of the array and the maximum sum of two elements such that there is no common bit between them.

Time Complexity: $O(m \cdot 2^m)$ where m is the number of bits in the maximum element of the array which is bounded by 23.

Problem G. Picturesque Skyline

First let's think about how to fix one segment of size $2k+1$ without thinking about creating any gaps. We just need to know what is the minimum number of swaps required to turn that segment into a pyramid-like pattern group. Here is one approach to calculate the minimum number of swaps needed to make a

segment [l, r] a pyramid-like. Let's find the smallest number in the segment. We must either push it to the beginning or the end of the segment, so how many moves are required to move this number? If we try to move it to the front then we would need to swap with all the bigger numbers before it in the segment. Similarly if we try to move it to the end then we would have to swap with all the bigger numbers after it in the segment. Now, let's think about the second smallest number, similarly we either move it to the front or move it to the end, but we can't move it past wherever we placed the smallest number. So, we continue placing all the numbers in sorted order either to the front or the end. But we can only place k numbers in the front and k numbers in the end and we don't do anything with the largest number as it will automatically be fixed in the middle.

Lets calculate for each building, the number of taller buildings that appear before the building in the segment and how many taller buildings appear after the building in the segment. Let's call these values a_i and b_i. To make a pyramid-like shape we need to have k sorted numbers at the beginning of the segment and k reverse-sorted numbers at the end with the largest number in the middle. Which means we need to move some of the buildings to the front and some of them to the back. The buildings that need to be moved to the front incurs a_i cost while the others incur b_i cost.

So basically, we assign each building to either type a or type b and make sure they both have the same size. We are not counting the tallest building in anything since it doesn't contribute anything to the minimum cost. To do the assignment, we can take $sum(b_i)$ and convert a_i to $a_i - b_i$. The smallest k values of $a_i - b_i$ will be added to $sum(b_i)$ and that would be the minimum number of moves required to make the segment pyramid-like. We have to do it for all odd length segments and for each segment we can easily do it in $O(n \cdot \lg n)$ complexity. But with the total number of segments this can grow very large.

We can improve it even more. For a segment of length 3, we never need more than a single swap. Which means, the upper bound of the result is $\frac{n}{3} + \varepsilon$. Here $\varepsilon = 7$ would work. We can get substantial speedup if we stop processing the segments that we know will need more than $\frac{n}{3} + \varepsilon$ swaps. One way to maintain that is to build [l,r] from [l+1, r-1]. If we know the number of swaps needed for [l+1, r-1] is more than $\frac{n}{3} + \varepsilon$ then we simply ignore these segments. However we are not sure how much speedup does this actually provide, what we only know is that it is very good. After that we can just do a dynamic-programming approach to split the groups. That part can be done in $O(n^2)$. Overall this performs somewhere close to $O(n^2 \cdot \sqrt{n})$, but we don't have an exact proof.

Problem H. Are the nodes reachable?

We can split the problem into two subproblems.

When $ans < 32$**:** First for each vertex calculate the list of vertex reachable from it. Also calculate the list of vertices from which it is reachable. Both can be done in $O(\frac{n \cdot m}{64})$ using bitsets. Now assume that vertex a is reachable from vertex b. In that case vertices $b - 1$ or $b + 1$ will be reachable by adding an edge of cost 1. We can find all the vertices from which vertex a is reachable with 1 cost by simply binary shifting the bitset by 1 in both directions and applying the bitwise OR operator on all three (no change, left shift, right shift) bitsets.
If we do this again then we find the vertices 2 distance away and so on. We keep doing this for distances upto 31. The complexity to do this is $O(\frac{n^2 \cdot 32}{64})$.
Now, let's split the bitsets into 64 sized buckets. Technically we could've used an unsigned long long array to perform the bitset operations in the first place. For each vertex we will have 33 bitset buckets. 32 of them are for the shifted bitsets from upstream and one for the downstream. Then for the query, we check if the first upstream bucket of V with 31 shifts and the downstream bucket of U has any match. We can do that by simply performing a bitwise AND operation. If there is a match then we check 30 shifts and keep reducing it. If no match is found then we move to the next bucket. At the end in this approach we will need $O(\frac{n}{64} + 32)$ operations per query.

When $ans \geq 32$**:** We again keep a bucket of vertices reachable from a vertex. Vertices numbered from [1..64] are in one bucket, [65..128] are in second buckets and so on. For each bucket we only keep the minimum and maximum id of the vertex reachable from it. Similarly we do it for the upstream

vertices. For a query, we merge those two lists of vertices (min and max of each bucket), sort them and find the smallest gap in consecutive vertices (one must be upstream while the other must be downstream). This takes $O(\frac{2 \cdot n}{64})$ per query.

Problem I. Unhappy Team

The trick is to select the unhappiness score of someone and count how many times that score appears in the top K values. This is basically counting the contribution of each score, which works because of linearity of expectation. We can keep a DP state $bit\_mask, bigger\_score\_count, selected\_score\_appeared$.

Basically we will put one people one after another, every time calculating their unhappiness score. The bit_mask will host the set of people already placed before the next person. If the score is smaller than the selected score then we ignore it. If the score is bigger then the selected score then $bigger\_score\_count$ is incremented by one. If the score is the same as the selected score then we can consider it as bigger or mark as $selected\_score\_appeared$. This DP will have the complexity of $O(2^n \cdot n^3)$. With some other minor optimizations, this runs reasonably fast.

Problem J. Hand Cricket

Please note: after the round we found the solution to be incorrect. The equilibrium can be reached where some index have their probability set to zero. The author's solution expected all the probability to be non-zero. As a result of this, we don't know of any solutions to this problem at the moment. We sincerely apologize for not catching that sooner. Below is the original solution.

The strategies are in Nash equilibrium. Therefore, Alice cannot improve her strategy. She will have the same expected points for any index. Let this expected value be X. Let $P_L, P_{L+1}, ..., P_R$ be Bob's strategy. Then for all $L \leq i \leq R$,

$$A_i \cdot (1 - P_i) = X$$

$$\implies P_i = 1 - \frac{X}{A_i}$$

Applying it on the definition of a strategy $P_L + P_{L+1} + ... + P_R = 1$, we get

$$\sum_{i=L}^{R} (1 - \frac{X}{A_i}) = 1$$

$$\implies X = \frac{R - L}{\sum_{i=L}^{R} \frac{1}{A_i}}$$

Alice needs to increase X. Reducing $\sum_{i=L}^{R} \frac{1}{A_i}$ will increase X. It can be proved that Alice always increments the smallest $\frac{1}{A_i}$. To implement this efficiently, we will build a persistent segment tree on the array values. The j^{th} root of the segment tree will account for array elements in the range $[1, j]$. The tree maintains the following,

- The sum of all A_i values in the range.
- The sum of inverses of A_i in the range.
- The count of elements available in the range.

We will walk the segment tree from the root for the $(L-1)^{th}$ element and the root of the R^{th} element. During this traversal, we will maintain,

- How much of K is left.

- The value and count of the small numbers that have been incremented.

On a segment tree node we can check if the whole subtree can be incremented to its maximum in $O(1)$ and also get the sum of inverses in $O(1)$ if K depletes.

Problem K. Island Invasion

We unfortunately don't have the editorial for this. We have the model solutions uploaded here.

- https://pastebin.com/XFbCgwuY
- https://pastebin.com/z7CSJkNe

Problem L. Uncle Bob and XOR Sum

Category: Maths, Gauss - Jordan Elimination, XOR Basis.

Given two arrays of integers **A** of length **N** and **B** of length **K**, how many subsets of array **A** are there so that the xor sum of the subset does not contain b_i as a submask where $(b_i \in B)$. That means $(S_{\oplus} \wedge b_i \neq b_i),$ for all $(1 \leq i \leq K)$.

That means if we choose a subset $X = \{x_1, x_2, x_3, \ldots, x_m\}$ then

$$(x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_m) \wedge b_i \neq b_i$$

or,

$$(x_1 \wedge b_i) \oplus (x_2 \wedge b_i) \oplus (x_3 \wedge b_i) \oplus \cdots \oplus (x_m \wedge b_i) \neq b_i$$

This can be solved using the XOR Basis. Instead of finding a good subset xor sum, we can find the total number of bad subset xor sums and then just subtract it from the total number of subsets.

As we could have duplicate results when calculating bad subsets, we need to carefully subtract them.

Let's say we have calculated the number of bad subsets that has b_1 as a submask and the number of bad subsets that has b_2 as a submask. There could be subsets that have both b_1 and b_2 as submasks. Hence, we need to subtract them. How can we do that? We know that if a subset xor sum $S_{\oplus}$ contains both b_1 and b_2 as submasks then

$$S_{\oplus} \wedge b_1 = b_1 \quad \text{and} \quad S_{\oplus} \wedge b_2 = b_2$$

We can derive a new equation from them:

$$(S_{\oplus} \wedge b_1) \vee (S_{\oplus} \wedge b_2) = b_1 \vee b_2$$

Or,

$$S_{\oplus} \wedge (b_1 \vee b_2) = b_1 \vee b_2$$

That means we need to find the subsets which contain $b_1 \vee b_2$ as a submask and then subtract it. More formally, we need to use the Inclusion–exclusion principle.

Complexity:

We need to calculate the basis $2^k - 1$ times. Hence, the complexity per test case is $\mathcal{O}(2^k N L^2)$ where L is the number of rows/equations we need to add. This could be as big as 32 because all numbers will fit in a 32-bit signed integer. If we use bitset, the complexity will be reduced by a constant factor of 32/64.

However, the above solution is too slow for the given constraints for this problem. We can actually optimize the solution even further. In our previous solution for each of the $2^k - 1$ bases we are inserting $a_i \wedge b$ where $1 \leq i \leq n$ and b is the OR of elements of the chosen subset of the array **B**. In reality, what we are doing is inserting submasks of a_i. Instead of doing this for all a_i, we can actually precalculate the basis of **A**. Then for each subset of **B** we need to calculate the basis over basisA$(i) \wedge b$ if the i^{th} bit of basisA(i) is set, where $0 \leq i \leq 32$. If we can make b, then the number of subsets that can make XOR Sum b would be $2^{(n-\text{rank})}$, otherwise 0.

Proof: Let, B_A be the basis vector of **A**. Let $f(a)$ be the number of subsets of **A** with XOR SUM a.

Let **X** be a subset of **B** and **x** be the OR of elements of **X**.

Lemma 1: $S_{\oplus} \wedge x = x$

Lemma 2: $f(a) = 0$ or $2^{(n-r)}$ where r is the rank of basis vector B_A.

For each subset **X** of **B**, we need to solve the following problem: how many subsets of **A** are there which contain all elements of **X** as a submask. Let, **x** be the OR of elements of **X**. Then we need to find how many subsets of **A** are there which contain **x** as a submask according to **Lemma 1**. In other words, we want to find the sum of $f(a)$ over all supermasks of **x**. Let $g(x)$ be the number of supermasks of **x** which can be made as a XOR SUM. Then by Lemma 2, we need to find $g(x) \times 2^{(n-\text{rank})}$. The $g(x)$ can be calculated with a modification of basis finding. Let's ignore all off bits of x and perform Gauss-Jordan Elimination only on the set bits. Let r' be the new rank. If we can make x, $g(x) = 2^{(n-r')}$ otherwise, $g(x) = 0$.

Complexity of precalculating the basis of A**:** $\mathcal{O}(NL^2/32)$

Complexity of finding the final answer: $\mathcal{O}(2^k L^3/32)$, here we are able to reduce N to L.

Overall complexity:

$$\mathcal{O}\left((2^k L^3 + NL^2)/32\right)$$

Note: Need to handle a few corner cases like 0 in array A and array B.

Problem M. Tree Flip

First let's think how we would have solved it if the limit on n was not so high. We will go down from the root to leaves in BFS fashion (or DFS would also work) and if we find a node with value 1, then we will make an operation at that node. This is the optimal way. Another way to think about it is, suppose we have 1 at a node x and all the other nodes are 0. To make the entire tree zero, we actually need to perform operations at all the nodes in the subtree of x (including x). Now just XOR all the corresponding subtrees of all the 1 nodes, you will know which nodes to perform operation at. So, if the root is u, the answer is the number of nodes v such that the xor of values from u to v is 1.

We can efficiently solve the problem using this last observation. First "centroid decompose" the entire tree. Then, for the query for a node x as root (Update 2), check the nodes in the path from x to the root in the centroid-decomposition tree. Say we are at y. Suppose we know the xor of values from x-to-y (this is easy to obtain using a segment tree on the euler tour of the original tree). Then we will need to find out how many nodes in the tree rooted at y have xor-path 1 from y. Depending on the xor value of the path x-to-y, you might need to actually find the xor-path 0 value in the tree rooted at y. All these are details to the implementation. But the main idea here is that we will need to maintain yet another set of segment trees for each of the components in the centroid decomposed tree to perform all these queries efficiently. Once you figure out how to do the Update 2 operation efficiently, you will be able to find out the efficient way to perform Update 1 operations.