Problem A. xuanquang1999 and the Problem Selection Process

We define position i as beautiful if $a_i < a_{i+1}$ (meaning we can perform a transformation at position i).

Suppose we perform a transformation at beautiful position i. Consider the four integers $a_{i-1}, a_i, a_{i+1}, a_{i+2}$. After the transformation, we have three integers $a_{i-1}, a_i + a_{i+1}, a_{i+2}$. We have the following observations:

- If $i - 1$ is a beautiful position before the transformation ($a_{i-1} < a_i$), then this position remains beautiful after the transformation (since $a_{i-1} < a_i + a_{i+1}$).
- If $i + 1$ is a beautiful position before the transformation ($a_{i+1} < a_{i+2}$), then this position may not remain beautiful after the transformation (since $a_i + a_{i+1}$ is not necessarily smaller than a_{i+2}).
- Positions j before the element a_{i-1} and after the element a_{i+2} will still maintain their beauty (or non-beauty) property, as they are not affected by the transformation.

Therefore, the best way to perform the transformation is always to choose the last beautiful position to perform the transformation (so that we do not lose any beautiful positions). Additionally, since there are no more beautiful positions appearing after the transformation position, we only need to iterate i from $n - 1$ backwards to 1, and perform the transformation if position i is beautiful.

Complexity: $O(n)$.

Problem B. TrungNotChung and the Competition Preparation

The answer is ``NO'' if there exists a pair of indices i, j ($i \neq j$) that satisfies either of the following conditions:

- $p_i = p_j$ and $q_i \neq q_j$.
- $p_i \neq p_j$ and $q_i = q_j$.

Conversely, for k being the number of distinct pairs (p, q), there always exists an answer with the smallest c equal to $k + 1$. For the chosen desk manufacturers, t_{p_i} can take any value from 1 to k such that no two manufacturers have desks of the same height. The chosen monitor manufacturers will purchase monitors with heights: $m_{q_i} = k + 1 - t_{p_i}$.

For the desk and monitor manufacturers that no contestant chooses, the height of the monitors and desks purchased from these manufacturers can take any value from $k+1$ to 10^9 such that no two manufacturers have desks or monitors of the same height.

Problem C. ngfam the Navigator

From the definition of the problem, we can see that a vertex u is called a centroid if the subtree rooted at u does not have a child v ($v \neq u$) such that the subtree rooted at v has at least $\frac{n}{2}$ vertices.

Subtask 1: $n \leq 100$.

In this subtask, we can use the `adj` operation to reconstruct the tree. We iterate through each vertex from 2 to n. For each vertex u in the iteration, we find the path from the current vertex (denoted as r) to u by continuously querying:

1. `adj u` (returns `k`)
2. `move k`
3. Assign `r = k`

At each `move` step, we can add (u, k) to the edge list of the graph. After obtaining the graph, we can iterate through all vertices and check the centroid condition as described and move to that vertex.

Number of required queries: $O(n^2)$.

Subtask 2: $n \leq 1000$**.**

With the limit of this subtask, we realize that reconstructing the original tree is not feasible. Therefore, we need an approach to reach the centroid without relying on the graph structure.

Using the observation from the previous section, if the current vertex r is not a centroid, we always have at least $\frac{1}{2}$ chance to find a randomly chosen vertex that belongs to the subtree rooted at the centroid.

Thus, we can use a random algorithm, sampling m vertices that have not been chosen and check if the subtree containing those vertices has at least $\frac{n}{2}$ vertices. If it does, we move to that vertex.

Observation: The distance from any vertex in the tree to the centroid is always at most $\frac{n}{2}$. It is easy to see that if this distance is greater than $\frac{n}{2}$, then the subtree containing r in the centroid tree contains all the vertices on the path, which is equivalent to at least $\frac{n}{2}$ vertices.

Combining the random approach and the above observation, we can repeat the following steps until reaching the centroid:

1. Sample m vertices, for each vertex, query `adj` and `subtree` to check if the subtree contains the centroid.
2. If there exists a subtree containing the centroid, use the `move` operation to move to that subtree.

The algorithm has a complexity of $O(mn + \frac{n}{2})$ with an error rate of each step being 2^{-m}.

To ensure accuracy, we need at least $m = 30$ to achieve a small error rate, but it exceeds the problem's limit.

At this point, we can optimize in various ways, one of which is:

1. After each move, gradually remove the vertices that have been traversed from the sampling list. Each time the sampling list decreases by 2, we decrease m by 1.
2. Instead of querying `subtree` for all m samples, we query the entire subtree from `adj` and only query `subtree` for the subtree that contains the most samples.

Subtask 3: $n \leq 10000$**.**

Solution 1: Improved random algorithm.

We observe that instead of performing sampling after each step to find the next direction, we can use a target vertex to perform multiple steps at once.

1. At each moment when we don't have a direction, we perform sampling until we find a direction (a vertex *target* such that the subtree containing *target* has at least $\frac{n}{2}$ vertices).
2. Move towards *target* until the subtree containing *target* no longer has at least $\frac{n}{2}$ vertices.

Thus, for each `move` step, we will perform 3 operations: [`adj`, `subtree`, `move`], where `subtree` is used to ensure that we do not move in the wrong subtree. Therefore, we spend a maximum of $\frac{3n}{2}$ operations for movement.

Thus, we have 555 operations for sampling and ensure that we always find the centroid with a very small error probability.

Solution 2: Solution by one of our testers

Observation: For each root u that is not a centroid, there exists one and only one subtree containing the centroid. From this, we always know for sure that the subtree containing the centroid is the subtree in the root u tree that contains the most vertices.

Based on the above observation, we have the following algorithm:

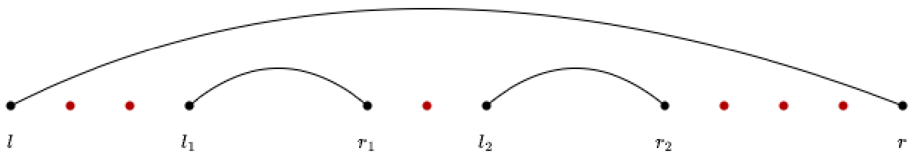
1. List the results of the `subtree` query for all vertices in the root 1 tree.
2. Sort the list of vertices in descending order based on the query results, denoting the resulting vertex sequence as $a_1 = 1, a_2, ..., a_n$.
3. Move from a_1 to a_2, a_2 to a_3, ..., a_{k-1} to a_k with a_k being the last vertex with a query result greater than or equal to $\frac{n}{2}$. And a_k is also the centroid we need to find.

Problem D. darkkcyan and Contestant Positions Planning

Subtask 1: $m \le 5000$.

In this subtask, we are allowed to recalculate the answer in $O(m)$. For convenience, we add a fictive network cable connecting 0 and $n+1$.

We call network cable x a child of y if y is the network cable with the largest u_y satisfying $u_y < u_x \le v_x < v_y$.



Consider a network cable y with children $x_1, x_2, \dots, x_k$, then the number of additional cables that can be connected as children of y is $cnt_y = \left\lfloor \dfrac{v_y - u_y - 1 - \sum_{i=1}^{k}(v_{x_i} - u_{x_i} + 1)}{2} \right\rfloor$. The answer to the problem is exactly the sum of all cnt_y for all available network cables.

We sort the network cables in increasing order of their left endpoints. We maintain a stack of nested network cables. When adding a cable x, first we remove all cables y at the top of the stack with $v_y < u_x$. After this step, the network cable y at the top of the stack (if any) will be the parent of x. We add x to the stack and continue this process with the remaining cables.

Subtask 2: $u_i > u_{i+1}$ for all $1 \le i < m$.

In this subtask, the parent of a cable can change at most once. We will use a `std::set` S to store the cables that are children of $(0, n+1)$. When adding (u_i, v_i), we perform a binary search on S (using the `lower_bound` function) to find the children of this cable, then remove them and add (u_i, v_i) to S.

Subtask 3: No additional constraints.

Considering the cables after adding the m-th cable, we construct a tree T representing the parent-child relationships of the cables. Instead of adding the cables to the set one by one, we will do the opposite and gradually remove the cables from m to 1. In this process, we need to know the current parent of a cable in order to update the result. It is easy to see that the parent to find is the nearest ancestor not yet removed in the tree T.

Here, we present a solution using *DSU* (Disjoint Sets Union). Initialize a DSU with m vertices corresponding to the cables. Let $par[x]$ be the parent cable of a cable x in the tree T. Traverse x from m to 1, perform the union of the edge $par[x] - x$ in the DSU, and the current parent of x will be the vertex with the lowest depth in the connected component containing x.

Note that we need to update the *cnt* array after each operation in subtask 2 and 3.

Problem E. Ianhf and the VNOI Cup T-Shirt Distribution Process

First observe that, with a ranking table consisting of m rounds, the organizers will choose the number k as follows:

- Let $r[i]$ be the best rank of contestant i after m rounds, and $c[j]$ be the number of contestants i satisfying $r[i] = j$.
- The organizers will choose the largest number k such that $c[1] + c[2] + \ldots + c[k] \leq s$.

Subtask 1: $n, m \leq 8$.

In this subtask, we only need to iterate through all $n!$ possible scenarios and calculate the best rank of all contestants.

Subtask 2: $n, m \leq 500$.

We observe that the condition for contestant i to receive a prize after m rounds is that there are at least $n - s$ other contestants j such that $r[j] > r[i]$.

Let $r_2[j]$ be the best rank of contestant i after $m - 1$ rounds. We see that there are three possible cases in round m:

1. *Contestant i achieves rank $x < r_2[i]$:*

   Assuming there are a contestants j with $r_2[j] > x$ (excluding contestant i), then there must be at least $n - s$ contestants among them who achieve ranks higher than x in round m.

   We can consider this problem as follows: given n people and n seats, with a bad people and $n-a-1$ good people. Knowing that seat x is already taken, arrange the remaining $n - 1$ people in the seats in such a way that exactly b bad people sit to the right of seat x.

   The answer for this case is $a! \cdot (n - a - 1)! \cdot \sum\limits_{b=n-s}^{a} \binom{n-x}{b} \cdot \binom{x-1}{a-b}$.

2. *Contestant i achieves rank $x = r_2[i]$:*

   Assuming there are a contestants j with $r_2[j] > r_2[i]$, then there must be at least $n - s$ contestants among them who achieve ranks higher than $r_2[i]$ in round m.

   Therefore, similar to case 1, the answer for this case is $a! \cdot (n - a - 1)! \cdot \sum\limits_{b=n-s}^{a} \binom{n-x}{b} \cdot \binom{x-1}{a-b}$.

3. *Contestant i achieves rank $x > r_2[i]$:*

   Similar to case 2, we assume there are a contestants j with $r_2[j] > r_2[i]$, then there must be at least $n - s$ contestants among them who achieve ranks higher than $r_2[i]$ in round m.

   Now, our problem changes a bit: given n people and n seats, with a bad people and $n - a - 1$ good people. Knowing that seat x is already taken, arrange the remaining $n - 1$ people in the seats in such a way that exactly b bad people sit to the right of seat $r_2[i]$.

   The answer for this case is $a! \cdot (n - a - 1)! \cdot \sum\limits_{b=n-s}^{a} \binom{n-r_2[i]-1}{b} \cdot \binom{r_2[i]}{a-b}$.

We iterate through all contestants, then iterate through the possible ranks in round m of contestant i, and compute the combinations. Therefore, the complexity of this algorithm is $O(n^3)$.

Subtask 3: $n, m \leq 2000$.

We will consider optimizations for each case:

1. *Contestant i achieves rank $x < r_2[i]$:*

   From the solution of subtask 2, we can see that the calculation formula does not depend on $r_2[i]$, so we can optimize this case using prefix sums instead of iterating through all ranks $x < r_2[i]$.

2. *Contestant i achieves rank $x = r_2[i]$:*

   We can forgo the optimization for this case

3. *Contestant i achieves rank $x > r_2[i]$:*

   From the solution of subtask 2, we can see that the calculation formula does not depend on the rank x, but only on $r_2[i]$. Thus, we can multiply the answer for one case by $(n - r_2[i])$ instead of iterating through all ranks $x > r_2[i]$.

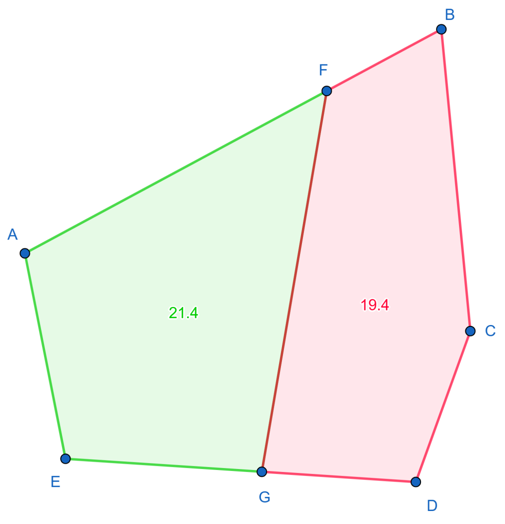
With these optimizations, the final complexity is $O(n^2)$.

Problem F1. FireGhost and Perfect Slice 1

First of all, we have the following important observation for both F1 and F2: any cut that satisfies the condition divides the cake into two parts with equal lengths. This is because the perimeter of each part includes the length of the polygon in that part, plus the length of the cut edge. Since the length of the cut edge is included in the perimeter of both parts, we only need the lengths of the polygons in the two parts to be equal in order to have equal perimeters for the two polygon parts. In other words, if P and Q are two cutting points on the cake, then P and Q are *antipodal points* of each other. It is easy to see that for any point P, the antipodal point Q is unique.

Now let's consider a simple approach for subtask F1 that can be applied to all subtasks, but cannot be extended to F2. Let f be a function that takes a point P on the polygon with the following definition:

- Take the point Q as the antipodal point of point P on the polygon. The value of $f(P)$ is equal to the area of the left part of the cake in the direction of the vector $\vec{PQ}$ minus the area of the right part of the cake in the direction of the vector $\vec{PQ}$ *without taking the absolute value.*



The left part of the cake in the direction of vector $\vec{FG}$ is colored red, and the right part of the cake in the direction of the vector is colored green. Therefore, $f(F) = 19.4 - 21.4 = -2$.

Observation: for any point P on the polygon and the antipodal point Q of point P, we have $f(P) = -f(Q)$. This is because the left part of the vector $\vec{PQ}$ is the right part of the vector $\vec{QP}$, and vice versa. Thus, the observation is proven by the definition of f.

Therefore, if we have a point A moving clockwise along the polygon boundary from point P to point Q, there must be a position where $f(A) = 0$ (since $f(P)$ and $f(Q)$ have opposite signs). Thus, the minimum difference in area can always be achieved as 0. We can also use a binary search algorithm to find a point A such that $f(A)$ is equal to 0 as follows.

- Initialize $A := P$ and $B := Q$. Here, assume that $f(A) > 0$ (and therefore $f(B) < 0$).
- Repeat the following steps several times: choose a point C that is "in the middle" on the clockwise path from A to B. If $f(C) > 0$, set $A := C$; otherwise, set $B := C$.
- When the two points A and B are close enough (or after repeating the necessary number of times), output the point A.

We can see that after each step of the above operation, we ensure that $f(A) > 0$ and $f(B) < 0$, while the clockwise distance from A to B is halved. Therefore, after enough steps, A and B will be close enough to have $f(A) \approx 0$.

As for the implementation, the simplest way to represent an arbitrary point P on the polygon is to use a real number to describe the clockwise distance from vertex 1 to point P. We only need to implement two additional functions: a function to find the antipodal point of an arbitrary point on the polygon, and the function f (or similarly, the function to find the area of the left part and the right part). These two functions can be implemented in $O(n)$ complexity, so our problem can be solved in $O(n \log \epsilon^{-1})$ complexity, where ϵ is the required accuracy.

(Those who want to learn more about the solution can read about the Borsuk-Ulam theorem and the bisection method for finding roots.)

Problem F2. Tahp and Perfect Slice 2

Subtask 1: $n = 3$.

With the condition that the cake is in the form of a triangle, we can find the optimal cutting line by solving a quadratic equation. Specifically, assuming that the two cutting points P and Q lie on the edges AB and AC.

We have the following conditions:

- $AP + AQ = \frac{AB+BC+AC}{2} = p$
- $AP \leq AB$
- $AQ \leq AC$
- $|AP \cdot AQ - \frac{AB \cdot AC}{2}| = |AP \cdot AQ - s|$ is minimized/maximized.

Assuming $AP \cdot AQ = q$, then AP and AQ will be the two solutions of the quadratic equation $x^2 - px + q = 0$ (according to Viète's formulas). Therefore, we need to find q such that $|q - s|$ is minimized and $x^2 - px + q = 0$ has two solutions $x_1 \leq AB$ and $x_2 \leq AC$.

Assuming $AB \leq AC$, the above conditions are equivalent to:

- $p - \sqrt{\Delta} \leq 2AB$
- $p + \sqrt{\Delta} \leq 2AC$

where $\Delta = p^2 - 4q$.

From the above two conditions, we can determine the range of q and calculate the corresponding values of AP and AQ.

Additionally, we can find q using ternary search.

Subtask 2: $n \leq 500$**.**

Assuming that the two cutting points P and Q lie on the edges A_iA_{i+1} and A_jA_{j+1}, when "moving" point P along the edge A_iA_{i+1} such that the corresponding point Q still lies on the edge A_jA_{j+1}, we can prove that the area of the left part of the cake with respect to the vector $\vec{PQ}$ has a parabolic shape (the specific proof is omitted here). Therefore, to solve problem F2, we only need to find the point P such that the corresponding area of the left part is maximized/minimized. Similarly, for problem F1, we need to find the point P such that the area is closest to $\frac{1}{2}$ of the total area. Both problems can be solved by constructing the equation of the parabola or using ternary search.

In summary, we can solve both problems F1 and F2 as follows:

- For each vertex A_i of the polygon, find the antipodal point B_i. Sort all $2n$ points and divide the perimeter of the polygon into $2n$ segments.
- In each segment, find the point P such that the difference in area between the two cake pieces is optimized by constructing the area equation or using ternary search.

The part of finding the antipodal point and calculating the area can be implemented in $O(n)$ complexity, so the complexity of this algorithm is $O(n^2)$ or $O(n^2 \log \epsilon^{-1})$, where $\epsilon = 10^{-12}$ is the maximum error.

Subtask 3: No additional limits.

To solve the last subtask, we need to optimize the algorithm in subtask 2.

Firstly, we can find the antipodal point in $O(\log n)$ by precomputing the cumulative sum of the lengths of the polygon edges, combined with binary search technique.

Next, we can quickly calculate the area of a cake piece in $O(\log n)$. Suppose we need to calculate the area of the polygon $PA_iA_{i+1}\cdots A_jQ$. Applying the shoelace formula, we have:

$Area(PA_iA_{i+1}\cdots A_jQ) = \frac{1}{2}(cross(P, A_i)+cross(A_i, A_{i+1})+\cdots+cross(A_{j-1}, A_j)+cross(A_j, Q)+cross(Q, P))$

where $cross(P, Q) = x_P y_Q - x_Q y_P$.

Let $pref_i = cross(A_0, A_1) + cross(A_1, A_2) + \cdots + cross(A_{i-1}, A_i)$, then $Area(PA_iA_{i+1}\cdots A_jQ) = \frac{1}{2}(cross(P, A_i) + \mathbf{pref_j} - \mathbf{pref_i} + cross(A_j, Q) + cross(Q, P))$.

With these two optimizations, we can implement the algorithm in subtask 2 with $O(n)$ or $O(n \log \epsilon^{-1})$ complexity, which is sufficient to solve subtask 3.

Problem G. MofK and Equipment Installation

In this problem, we need to find a topological order of a tree $[u_1, u_2, \ldots, u_n]$ such that $n \cdot u_1 + (n-1) \cdot u_2 + \cdots + 1 \cdot u_n$ is minimized.

Subtask 1: $a_{p_i} \leq a_i$ **for all** $2 \leq i \leq n$**.**

In this subtask, it can be easily observed that the optimal topological order must satisfy $a_u \leq a_v$ if vertex u comes before vertex v in the topological order. Therefore, we can use `std::priority_queue` (or simply sort the indices of the array a) to find the answer with a complexity of $O(n \log n)$.

Subtask 2: $n \leq 2000$**.**

We have the following claim (which will be proven in the following subtasks, but not necessary for this subtask): each subtree has an optimal topological order, and *this order remains unchanged even after merging it with the topological orders of other subtrees.* In other words, for each subtree, we only need to consider one unique optimal topological order.

Using this claim, the best topological order of the subtree rooted at u consists of vertex u at the beginning, followed by the concatenation of the optimal topological orders $t_1, t_2, \ldots, t_k$ of the subtrees rooted at $v_1, v_2, \ldots, v_k$, respectively, where $v_1, v_2, \ldots, v_k$ are the immediate children of u. Note that since we can shuffle the sequences $t_1, t_2, \ldots, t_k$ arbitrarily (as long as we keep the elements of each sequence in the correct order), we can perform a knapsack-on-tree dynamic programming to merge these sequences as follows:

For any two sequences a and b, let $dp_{i,j}$ be the minimum cost if we merge the first i elements of a with the first j elements of b. Then $dp_{i,j} = \min(dp_{i-1,j} + s \cdot a_i, dp_{i,j-1} + s \cdot b_j)$, where $s = |a| + |b| - (i + j - 1)$. After calculating $dp_{|a|,|b|}$, we can backtrack to find the best way to merge the sequences a and b. Since the complexity of each dynamic programming step is $O(|a| \cdot |b|)$, the problem can be solved with a complexity of $O(n^2)$.

Subtask 3: There exists $3 \le k \le n$ such that $p_k = 1$; otherwise $p_i = i - 1$ for all $2 \le i \le n$, $i \ne k$.

In this subtask, the tree consists of two paths connected at root 1. Since the topological orders of these two paths are already determined, let's assume these orders are represented by two arrays a and b. We need to find the best way to merge these two arrays a and b to form the most optimal array c with reasonable complexity.

First, we observe that if two consecutive elements in c are $c_i = a_x$ and $c_{i+1} = b_y$, and $a_x > b_y$, swapping these two elements in c ($c_i = b_y$, $c_{i+1} = a_x$) will result in a better answer. We can generalize this observation as follows: if there are two subsequences of a and b consecutively appearing in c (i.e., c has the form $[\ldots, a_i, a_{i+1}, \ldots, a_j, b_k, b_{k+1}, \ldots, b_l, \ldots]$), if the mean of the subsequence from a is greater than the mean of the subsequence from b (i.e., $\frac{a_i + a_{i+1} + \cdots + a_j}{j-i+1} > \frac{b_k + b_{k+1} + \cdots + b_l}{l-k+1}$), swapping these subsequences will yield a better c.

From there, we have an important observation: if $a_i \ge a_{i+1}$, these two elements will be adjacent in c. Assuming otherwise, c will have the form $[\ldots, a_i, b_j, b_{j+1}, \ldots, b_k, a_{i+1}, \ldots]$. Consider the following two cases:

- The mean of the elements from b in between is less than a_i. In this case, $c' = [\ldots, b_j, b_{j+1}, \ldots, b_k, a_i, a_{i+1}, \ldots]$ will be better than c.
- The mean of the elements from b in between is not less than a_i. In this case, $c' = [\ldots, a_i, a_{i+1}, b_j, b_{j+1}, \ldots, b_k, \ldots]$ will be better than c.

Therefore, we can "glue" these two consecutive elements a_i and a_{i+1} back into a segment of a. We continuously perform this pre-processing on a, and in the end, we will get a series of glued segments, and an important observation is that the mean of the elements forming these segments will be increasing. Similarly, after pre-processing b, we will also obtain a series of glued segments with increasing means. The final step is to concatenate these glued subsequences in order of increasing means to get c.

We have found a way to merge the topological orders a and b in complexity $O(|a| + |b|)$. Therefore, this subtask can be solved with a complexity of $O(n)$.

(Although not necessary for the above algorithm, we can further prove that the order for permforming the gluing operations does not affect the final glued subsequences that can be obtained.)

Subtask 4: No additional constraints.

Due to the observation above, each subtree not only has a unique optimal topological order, but we can also represent this order as glued segments with increasing means. We can use the data structure `std::set` or `std::priority_queue` to store these segments (since these data structures are already sorted in increasing order). When merging the subtrees together, we can merge the representation of the segments as well, while applying the small-to-large technique to ensure the final complexity. Additionally, after merging the direct subtrees of the subtree rooted at u, we need to add the value of the root u to the beginning of the topological order; this means we need to modify some glued segments at the beginning of the data structure to obtain the representation of the entire subtree rooted at u. As a result, the problem can be solved with a complexity of $O(n \log^2 n)$.

(Furthermore, using specific data structures that support merging in $O(\log n)$ such as leftist heap, randomized heap, or treap can reduce the complexity to $O(n \log n)$.)

Problem H. Kuroni and the Sharing of the VNOI Cup Problem Setting Process

Subtask 1: $n \leq 20$, $q \leq 1000$.

In this subtask, we can solve the problem using bitmask dynamic programming for all states of the sequence. Let dp_{msk} be the maximum value that can be achieved if the sequence starts with bitmask msk. We can precompute all the answers with a complexity of $O(n \cdot 2^n)$. Therefore, this subtask can be solved with a complexity of $O(n \cdot 2^n + q)$.

Subtask 2: $n, q \leq 1000$.

Our first observation is that if we can solve the problem with a binary sequence b, we can solve it with any sequence b by binary searching the answer. At each binary search step, we need to query if the answer for the subsequence in the query is greater than or equal to x; hence, the subsequence becomes a binary sequence with each value being 0 or 1 depending on whether the initial element is greater than or equal to x.

Our goal now is to find a way to solve each query for the binary sequence b with $O(n)$ complexity. For a binary sequence, in each query, we only need to check if the subsequence can be transformed to 1.

We have the following greedy observations:

- If there are still three consecutive 0's, we perform an operation on these three 0's to transform them into a single 0.
- Otherwise (assuming there are still 0 elements in the array), we want to perform an operation on three consecutive elements where at least one is 0 and one is 1. This operation is equivalent to "selecting two adjacent elements 0 and 1 and removing them". Therefore, in this case, we find the first 01 pair and remove it (we don't find the first 10 pair because in the case of 1001..., we want the first 0 to be connected to the rest to continue creating three consecutive 0's).

These two observations lead to the following stack-based algorithm. Starting with an empty stack S, we iterate through the elements from left to right. For the current element b_u:

- If the current stack is empty or ends with 1: we push b_u to the end of the stack.
- If the stack ends with a 0: if $b_u = 1$, we remove the 0 at the end of the stack; otherwise, we push $b_u = 0$ to the stack.
- If the stack ends with two 0s: regardless of the value of b_u, we remove the 0 at the end of the stack.

It is easy to see that the answer is `YES` only in the following cases:

- At any time, $S = [1, 1]$. This is because we can process the elements after it to transform the value to any x, then perform the final operation with the sequence $[1, 1, x]$.
- After running through the entire sequence, $S = [1]$.

(Note that if the stack ends with $S = [1, 0, 0]$, the answer is `NO`.)

Hence, we have solved the problem with complexity $O(nq \log(\max b))$ or $O(nq \log n)$.

Subtask 3: $n, q \leq 100,000$.

Similar to subtask 2, we will first solve the problem with the binary array b, and then the problem can be solved by applying the binary search method in parallel to find the answer. Our goal is to find a way to solve a binary subsequence in $O(\log n)$ using a segment tree.

Since the operations described in subtask 2 are not associative, we cannot apply the segment tree immediately. We have the following observations:

- At any time, all the 1 values in S come before all the 0 values.
- At any time, S contains at most two 0 values at the end.

Therefore, we only need to consider 9 (or actually 7) states of S, represented by two numbers:

- The number of 1 values at the beginning of S (equal to 0, equal to 1, or greater than or equal to 2).
- The number of 0 values at the end of S (equal to 0, equal to 1, or equal to 2).

Based on these observations, we can store 9 (or 7) pieces of information in each segment tree node: if the stack S starts with state i ($1 \le i \le 9$), then after adding the elements in the range in order, the stack S will end in which state. Using this information, we can solve each binary subsequence in $O(\log n)$. Together with parallel binary search, the problem can be solved with a complexity of $O((n + q)\log^2 n)$.