GP of Tokyo Editorial

writers: yosupo, maroonrk, sigma425

A: Cookies

This problem can be rephrased as the following:

> Process N queries. In each query, you are given an integer v and do the following operation for $i = 1, 2, \cdots, M$.
>
> - When S_i ='S', swap v and B_i if $v > B_i$.
> - When S_i ='B', swap v and B_i if $v < B_i$.
>
> After the operations, return the value of v.

First, if all S_i are the same in some range $[l, r)$, we can merge them into one priority queue $q = \{B_l, B_{l+1}, \cdots, B_{r-1}\}$. Then, the abovementioned operation can be done by pushing v into q and popping the smallest/largest element from q. From now on, we denote by $|q|$ the size of the queue and $top(q)$ the smallest/largest element of q corresponding its type.

Now we can assume we have a sequence of priority queues, with Min queues and Max queues alternating.

Consider some adjacent Min queue x and Max queue y. If $top(x) \geq top(y)$, we can swap the positions of x and y, and then merge them to neighboring ones. We call this operation a "reduction procedure."

If we run reduction procedures as long as possible after each query, we can get a nice bound on the time complexity. We will prove it now.

Let L be the current number of Max queues in the sequence. We will see that, after $2M/L$ queries, the number of Max queues is at most $5/6L + constant$.

Let's name the queues as $x_1, y_1, x_2, y_2, \cdots, x_L, y_L$, where x_i denotes Min queues and y_i denotes Max queues. There may be y_0 or x_{L+1}, but we discard them since they are only two. We denote the pair (x_i, y_i) by "pair i". There are at least $L/2$ pairs such that $|x_i| + |y_i| \leq 2M/L$.

Let's define $overlap(i) =$ (the number of $e \in x_i$ such that $e < top(y_i)$) + (the number of $e \in y_i$ such that $e > top(x_i)$). We can see that, after each query, $overlap(i)$ decreases by at least one if it was non-zero. So, if $|x_i| + |y_i| \leq 2M/L$, within $2M/L$ queries, one of the following must hold:

- $overlap(i)$ becomes 0 at some point. In this case, x_i and y_i are swapped, and they are merged with x_{i+1} and y_{i-1}.
- $overlap(i-1)$ becomes 0.
- $overlap(i+1)$ becomes 0.

Therefore, within $2M/L$ queries, the event of swapping a pair happens at least $L/6$ times in total. In other words, we merge Max queues at least $L/6$ times, and the number of Max queues is at most $5/6L + constant$ as desired.

The time complexity is $O((N + M \log M) \log M)$.

B: Evacuation

Consider a query (l, r), and let x be the position where people are visiting and $f(l, r, x)$ be the minimum cost needed. We can see that if $x \leq (l+r)/2$, the value of r doesn't matter, and for $x \geq (l+r)/2$ the value of l doesn't. So let $fleft(l, x) = f(l, r, x)$ for $x \leq (l + r)/2$ and $fright(r, x) = f(l, r, x)$ for $x > (l + r)/2$. Then we only need to consider the right part, that is, $\max\{(l + r)/2 \leq x \leq r\}\ fright(r, x)$, because the left part can be calculated in the same way by the symmetricity.

Let $g(r, x) = fright(r, x)$ for a convenience. With some precalculation, we can query $g(r, x)$ for given r and x in $O(1)$ time. Also, we can prove that $g(r, x) + g(r + 1, x + 1) \geq g(r, x + 1) + g(r + 1, x)$. This fact strongly suggests using SMAWK or Divide and Conquer optimization, but how?

Let's build a segment tree, where each node maintains a list of "r"s. For each query, say, $max\{lower \leq x \leq upper\}g(r, x)$, put the value of r to the nodes corresponding to the range $[lower, upper]$. After we process all queries, in each node, we can run SMAWK or D&C to get the answer for each r. By combining the results of all nodes, we can get answers to all queries.

The time complexity is $O((N + Q) \log N)$ or $O((N + Q) \log^2 N)$ depending on the algorithm used in calculating row maxima of anti monge matrix.

C: Sum Modulo

Let's rephrase the statement as follows:

> You have an integer X, which is initially i. You repeat decreasing X mod M with random numbers. What is the expected number of operations until x becomes 0?

Let $f(i)$ be the answer to this problem. If $i \geq N$, $f(i)$ can be represented as a linaer combination of $f(i-N), f(i-N+1), \cdots, f(i-1)$ and some constant. This means that we can represent any $f(i)$ as an N dimensional vector, whose coefficients correspond to weighets of $f(1), \cdots, f(N-1)$ and 1 (constant). Note that $f(0) = 0$ so we don't need to consider its weight.

Let's say we have vector representations of $f(M-N+1), f(M-N+2), \cdots, f(M-1)$. Since $f(i)$ $(1 \leq i \leq N-1)$ is a linear combination of $f(M+i-N), f(M+i-N+1), \cdots, f(0), \cdots, f(i-1)$, we can get non-obvious vector representation of $f(i)$. This means that we have $N-1$ linear equations of $N-1$ variables. If we solve it, we get values of $f(i)$ for each $1 \leq i \leq N-1$.

The value of $f(K)$ can be calculated with its vector representation.

What's the complexity? Calculating the vector representation of $f(M-N+1), f(M-N+2), \cdots, f(M-1)$ can be done in $O(N^2 logM)$ time by binary lifting. Note that you can't just multiply $N \times N$ matrix since it takes $O(N^3)$ time. Instead, you should do polynomial things. (See "How to calculate k-th term of a linear recurrence?" section of this article for a start.) Solving the system of linear equations takes $O(N^3)$ time. Thus, the overall complexity is $O(N^2 \log M + N^3)$.

D: Xor Sum

If $S < X$ or $(S - X) \neq 0 \mod 2$, the answer is impossible. Note that there are necessary but not sufficient.

Let's do a binary search, and then we have to determine if, given a max value M, it is possible to construct a sequence that satisfies conditions, and all numbers don't exceed M.

Let b_i be the number of a_j such that the i-th bit of a_j is 1. If we fix b_i for all i, it is easy to solve the problem. To put it more precisely, we can do greedy from top bit to bottom. We maintain how many numbers are decided to be strictly less than M (let's say this is group A and others are group B). Then the transition is clear: we first try to add 1s to group A and remainings (if any) to group B. In this way, we always maximize the size of the group A.

Let $optb_i$ be the b_i of one of the solutions. We define $initb_i$ as follows:

- $initb_i =$ (the i-th bit of x) + (the i-th bit of $(S - X)/2$) $\times$ 2

We can prove that we can obtain $optb_i$ from $initb_i$ by repeating the following operations:

- Choose an integer i $(1 \leq i)$, and replace $initb_i$ by $initb_i - 2$ and $initb_{i-1}$ by $initb_{i-1} + 4$.

Now we can consider a dp like this:

- $dp[i][j] =$ minimum value of carry to b_{i-1} when we have decided the value of $b_{60}, b_{59}, \cdots b_i$ and the size of the group A is j.

The index j in this dp can be enormous, but we can reduce it. We don't need to consider the following transition:

- $dp[i][j] \to dp[i+1][k]$ where $k \geq 5$ and $k - j \geq 2$ and $dp[i+1][k] > 0$.

If $dp[i+1][k] = 0$ and $k \geq 3$, we know the answer is possible. Therefore practically we need only $j \leq 5 + \log V$ where $V = max(S, X)$. This observation leads us to a $O(\log^2 V)$ dp. Taking the $\log V$ factor of the binary search into account, we can solve the whole problem in $O(\log^3 V)$ time.

E: Count Modulo 2

Let's fix a frequency b_i of each interger A_i. Our first problem is to, given frequencies, find whether there are an odd number of ways to arrange integers. After some calculations, it turns out that there are odd number of ways if and only if $b_i \& N = b_i$ for all i (here & denotes bitwise and operation) and $b_i \& b_j = 0$ for all $i \neq j$. We can prove it by Lucas's theorem. In other words, we need to consider the following problem:

> For each i where the i-th bit of N is 1, consider adding one of $2^i A_1, 2^i A_2, \cdots, 2^i A_K$. In how many ways the total sum is equal to S?

A standard digit dp can solve this. More precisely, you can decide values in a bottom-up way. After you determine the value of $2^i A_j$, the total sum has only $O(V)$ candidates where V is the maximum of A_j, because the sum is no larger than $2^{i+1}V$ and the remainder modulo 2^i must be the same as S.

This dp runs in $O(KV \log N)$ time. If you use bitset, this is fast enough.

F: Robots

The obvious lower bound is $\sum_{i=1}^{N} |a_i - b_i|$, and we can achieve it.

If there exists an i such that $a_i < b_i$ and $b_{i+1} < a_{i+1}$, we can activate either antenna i or $i+1$. If such an i doesn't exist, we can activate either antenna 1 or N.

Repeating this process N times leads us to the lower bound. If we implement it carefully, we can run this operation N times in $O(N)$ time.

G: Matrix Inversion

Let Z be the number of pairs of two cells $((i_1, j_1), (i_2, j_2))$ such that

- $(i_1, j_1) \neq (i_2, j_2)$
- $i_1 \leq i_2$ and $j_1 \leq j_2$
- $M_{i_1,j_1} > M_{i_2,j_2}$

Besides, let W be the number of pairs of cells $((i_1, j_1), (i_2, j_2))$ such that

- $i_1 < i_2$ and $j_1 > j_2$
- $M_{i_1,j_1} > M_{i_2,j_2}$

We can see that $X = Z + W$ and $Y = Z + (N(N-1)/2)^2 - W$. So, from X and Y, we can determine the values of Z and W. Also, we can get upper bounds for Z and W: the number of pairs of two cells whose relative position satisfies the condition. Let these bounds be $Zmax = (N(N+1)/2)^2 - N^2$ and $Wmax = (N(N-1)/2)^2$. We can prove that these bounds are sufficient conditions.

Consider writing integers from 1 to N^2 in this order. When writing an integer, there are only 4 condidates of positions to consider:

- The empty cell (i, j) with the pair (i, j) being the lexicographically smallest possible.
- The empty cell (i, j) with the pair (i, j) being the lexicographically largest possible.
- The empty cell (i, j) with the pair (j, i) being the lexicographically smallest possible.
- The empty cell (i, j) with the pair (j, i) being the lexicographically largest possible.

Consider a set S of empty cells which may appear in the above construction. Let $Zmax(S)$ and $Wmax(S)$ be the bounds of Z and W in this set, respectively. By induction, we can prove that we can achieve arbitrary $0 \leq Z \leq Zmax(S)$ and $0 \leq W \leq Wmax(S)$.

Therefore, we can write an integer to one of the four cells so that $0 \leq Z \leq Zmax(S)$ and $0 \leq W \leq Wmax(S)$ still hold. You can get the answer by repeating this process.

H: Construct Points

Choose two lines which are almost parallel and apart from each other. For example $(a, b) = ((-10^9, 0), (1, 10^9))$ and $(c, d) = ((0, -10^9), (10^9, -1))$ work.

I: Amidakuji

We will use the 0-based index. Let $K = 1 + \lceil \log_2 N \rceil$.

If N is odd, we can set $p_{i,j} = j + 2^i \mod N$ for all i and j. This construction satisfies the condition, because for an input x the output can be $x - constant + (2 \text{ or } 4 \text{ or } \cdots 2^{K+1}) \mod N$

This construction fails if N is even. However, if N is a multiple of 4, we can set $p_0 = (2, 3, 1, 0, 6, 7, 5, 4, \cdots, N-2, N-1, N-3, N-4)$. For any input, after applying p_0 or p_0^{-1}, the output can be both odd and even number. Then, for $p_1, \cdots, p_{K-1}$, we can just use the similar construction for odd N.

We can handle the case $N = 2 \mod 4$: let $p_0 = (2, 3, 1, 0, 6, 7, 5, 4, \cdots, N-4, N-3, N-5, N-6, N-2, N-1)$ and $p_1 = (0, 1, 2, 3, \cdots, N-6, N-5, N-2, N-1, N-3, N-4)$. Note that only $N = 2$ doesn't have a solution.

J: Median Replace Hard

We can prove that, for any P, there is a DFA that accepts a string if and only if it is beautiful. If we have such a DFA, we can calculate the answer by simple DP.

How to get the DFA? We can assume two strings x and y are on the same state in the DFA if, for any string z of length no larger than L, $x + z$ is beautiful if and only if $y + z$ is beautiful. Using this condition to determine the equivalence of states, we can create a DFA. We checked $L = 10$ works.

We now have a solution, but how to prove it? Let's say we have DFAs $d_0(k)$ and $d_1(k)$ that correctly determine if a string of length no larger than k can be transformed into 0 and 1, respectively. By combining these DFAs, we can create NFAs $e_0(k + 2)$ and $e_1(k + 2)$ that works for any string of length no larger than $k + 2$. Since we can convert any NFA to a DFA, we get $d_0(k + 2)$ and $d_1(k + 2)$. Then, we check if these are equivalent to $d_0(k)$ and $d_1(k)$. If so, we get $d_1(k) = d_1(k + 2) \cdots d_1(\infty)$ and we can be sure that it's the desired DFA. The authors ran the above procedure for all P and verified DFAs.

The time complexity is $O(NS)$, where S is the size of the automaton. In this problem, S is at most 35, so this algorithm fits in time.

K: Game and Queries

Let $freq[i]$ be the number of monsters whose HPs are i.

Firstly, let's solve the game without queries. By induction, we can prove that the optimal strategies for both players are:

- Alice: choose the monster with the smallest HP.
- Bob: If the number of monsters whose HPs are 1 is not less k', choose one such monster, and otherwise choose the monster with smallest HP, but not HP=1. (Here, k' denotes the number of monsters he needs to beat from now)

The induction step is not easy, but it's just complicated casework. Thus we will omit the details.

Then how to compute the number of turns? We can see that one monster with HP=1 takes 1 turn. So we need to solve the case where $freq[1] = 0$.

Let x be the smallest HP of monsters. We can see that, after some turns, $freq[x+1]$ increases by $floor(freq[x]/2)$ and the smallest HP of monsters is $x+1$. Therefore we can know how many monsters disappeared at the moment when the smallest HP of monsters becomes x (let's call this value $g(x)$) by updating $freq[i]$ for $i = 2, 3, 4 \cdots$ in order. This calculation is similar to the addition of binary numbers. We can use this property to process queries.

Let's build a segment tree on the array $freq$. What we want to calculate is the largest x such that $g(x) \leq k$, because the other parts can be done easily. We can see that $g(x) = (\sum_{i=2}^{x-1} freq[i] \times i) - ($ the number of monsters whose HPs are initially less than x but became $x) \times x$, which means we need to get $(freq[2] \times 2^2 + freq[3] \times 2^3 \cdots freq[x-1] \times 2^{x-1})/2^x$ to calculate $g(x)$. For this purpose, each node of the segment tree keeps the following data:

- let $[l, r)$ be the interval corresponding to the node.
- $carry = (\sum_{i=l}^{r-1} freq[i] \times 2^i)/2^r$.
- $a =$ the smallest value such that adding a to $freq[l]$ changes $carry$.

With these values, we can merge nodes. Note that the calculation involves big numbers like 2^{r-l}, but we can cap them at something like 2^{30} because of the constraints of this problem.

Time complexity is $O(Q \log 10^6)$ if you do a binary search directly on the segment tree.

L: Yosupo's Algorithm

Let's do Divide and Conquer on the y coordinate.

Let's say we divide the points at $y = threshold$. Then, we can enumerate the pairs of red point i and blue point j such that:

- $r_i^y < threshold$
- $b_j^y > threshold$
- the pair (i, j) can be the answer to a query.

We can see that, in such a pair, point i or j must have the largest weight among the points of the same color that satisfy the condition of y-coordinate.

Therefore, we have $O(NlogN)$ candidate pairs in total because there are $O(N)$ pairs in each level of the Divide and Conquer.

The rest is easy: do a sweep line stuff and get answers. The time complexity is $O((N \log N + Q) \log N)$.