Problem A. Factory Balls

We would like to apply a breadth-first search, but there are too many states to run a BFS on: each region has one of the K colors, and each piece of equipment is either equipped or unequipped, leading to $K^N 2^M$ states in total.

However, if the color of a certain region does not match the target color, then the exact color does not matter. All we need for each region is whether the color matches the target or not. Therefore each region has only two "colors." This reduces the number of states to 2^{N+M} and the time complexity of the BFS to $O(2^{N+M}(KN+M))$. To represent each state, we can use a pair of bitmasks (or a single bitmask if you combine them).

Using bitwise operators more cleverly, We can do even better by immersing the ball into a paint can in $O(1)$ time, leading to $O(2^{N+M}(K+M))$.

Problem B. Distance Optimizing Triangulation

If you simply add $N-3$ edges from vertex 1 to vertices $3, 4, \ldots, 2N-1$, then the shortest path between any two vertices has length at most 2. This immediately gives a solution with value at most $2N-1$. The value can't be less than N, so it lies somewhere in $[N, 2N-1]$.

Let's denote arc i as an arc that connects vertices x_i and y_i. Consider another graph of N vertices, where each arc is a vertex, and two vertices are connected by an edge if and only if the corresponding arcs intersect. For each connected component of this graph, apply the above solution independently: Take any vertex in the component, and connect it with all others. This solution is valid because no arcs are intersecting (the convex hulls of each connected component do not intersect). The value of this solution is $2N$ minus the number of connected components.

Now we show that we can't do better. Let's think about a different (yet similar) problem. We start with $2N$ vertices and no edges in a planar graph. We need to add the minimum number of edges between those vertices so that the graph remains planar and all vertices of the same color are connected.

In this problem, the answer is at least $2N$ minus the number of connected components. The arcs that belong to the same connected component belong to the same connected component in that planar graph as well, so we have the upper bound on the number of connected components in the planar graph we build. Even if we assume that each connected component is a tree (that has the smallest number of edges), we need at least $2N$ minus the number of connected components.

Let's show that the original problem has an answer not less than in the modified problem. If we mark the shortest path between x_i and y_i in the augmented graph, this corresponds to the solution for the modified problem. Obviously, the number of the marked edges is not greater than the sum of the shortest paths.

Now what remains is to implement the above algorithm efficiently. The naive implementation of finding the connected components takes $O(N^2)$ time. There are numerous ways to improve this algorithm. One way is to use segment trees for doing DFS efficiently in the graph. Another way is to use hashes. Let $x_1, x_2, \ldots, x_n$ be random 64-bit integers. Then each connected component is a minimal partition where the XOR of all vertices is zero. Iterate through the vertices in order, and maintain the XOR sum of all vertices encountered. If you find the duplicated value of such XOR sum, you have found a connected component, which you can delete and continue.

Also, remember that you may need to add some redundant edges in the graph because you must print exactly $2N-3$ edges.

Problem C. UCP-Clustering

The RC of two clusters uniquely defines the set of next RC of two clusters, and such next RC can be computed in $O(N)$ time by computing the median in linear time. Therefore, the states and their transition takes the shape of a functional graph, where every node has an outdegree of one. Since the algorithm is guaranteed to terminate, every initial node will eventually reach a vertex that self-loops. The expected value of the iteration count can be considered as the average distance from all initial states that can reach the current loop. Since the graph takes the shape of a tree, this can be easily computed.

As a result, we can obtain a conceptually simple (though possibly tedious to implement) $O(MN)$ solution where M is the number of states. Naively, $M = O(N^8)$, since there are $O(N^2)$ possible values that each coordinate can take. However, we show how to obtain a much tighter upper bound.

Draw a line between the current RCs, and consider the perpendicular bisector. The bisector divides the plane into two halfplanes. Then, the points belonging in each halfplane are the clusters in the next iteration. As a result, for all states with in-degree greater than zero, there exists a dividing line between each cluster.

Suppose that two clusters have a dividing line. Then we can rotate the line until it hits the points from both clusters. The lines touch two points from the input and define a unique cluster. As a result, there are at most $O(N^2)$ states with in-degree greater than zero. States with in-degree zero are exactly the valid initial state, of which there are $N(N-1)/2$ possible candidates.

As a result, we obtain an $O(N^3)$ algorithm.

If you want a challenge, you can try to optimize the above algorithm to $O(N^2 \log N)$.

Problem D. Triple Sword Strike

Let's assume that we perform at least two sword strikes parallel to the x-axis. The other case can be handled by reflection along the line $x = y$.

For the case where three sword strikes are parallel to the x-axis, maintain the array $count[y]$ that denotes the sum of values for all monsters with y-coordinate y. You can see that the answer is the sum of the top three elements.

For the case where two sword strikes are parallel to x-axis, we will fix the x-coordinate of one sword strike which runs parallel to y-axis. Let this value p, and let S_p be the set of monsters with x-coordinate x.

If we decrease the $count[y]$ value for the monsters in S_x, we can simply pick the top two elements in the array. However, we need to recompute the top two elements every time, which results in $O(N^2)$ time complexity if done naively.

Construct a list that maintains all possible indices of y, sorted by decreasing order of $count[y]$. Let's say we decreased the value $count[y]$ for elements in S_x. You can observe that the top two elements are one of the top $|S_x| + 2$ elements in the list. The list contains at least 2 elements that have their $count[y]$ unchanged, so the following elements must have their value not larger than those two elements.

In conclusion, for each x coordinate, you can compute the top two in $O(|S_x|)$ time. This results in an $O(N)$ time algorithm except for the sorting of $count[y]$ value, which can be also done in $O(N)$ time with counting sort.

Problem E. RPS Bubble Sort

Suppose that there are exactly two distinct characters in the string. In this case, Yihwan's game actually tries to sort a string where the losing character comes before the winning character.

Here, it is helpful to analyze from the perspective of the losing character. In each pass of the game, the losing character will move left by one if there is a preceding winning character and not move otherwise. In conclusion, if the k-th losing character was in the position i, it will be at position $\max(i - T, k)$ after T iterations of the game. Using this observation, this special case can be solved in $O(N)$ time.

Now let's go back to the general case. Partition the string by repeatedly cutting the longest prefix of the string that has at most two distinct characters. Here are the key observations for this problem.

Observation 1. In the first pass of the game, no character is swapped across partitions.

Proof. Let A be a winning character, and B be a losing character in the first partition. Since the partition did not extend further, the first character of the second partition is C. After the first pass swapped all characters until the first partition, the last character of the first partition is A. A can beat B, but it loses to C. Thus, the swap does not happen. By induction, you can show that this holds for all subsequent partitions.

Observation 2. In any pass of the game, no character is swapped between different partitions.

Proof. The first pass leaves a losing pair at the border of different partitions, and this does not change after further passes.

These observations reduce the problem to the two-character case. The time complexity is $O(N)$.

Problem F. Stones 1

If a contiguous segment of stones contains only one color, you can gain points from at most one of the stones. Partition the array into maximal contiguous monochromatic segments, and only leave the stones with the largest weight for each segment. After this procedure, you can assume that adjacent stones have different colors. In other words, the color of the stones alternates.

Let's see how many times we can gain points from removing the stones. Assume $N \geq 3$ since otherwise, you can gain no points.

You can not gain any points from the leftmost and rightmost stones. If you remove the stones in the middle, you can only possibly gain points from one of the adjacent stones. Repeatedly applying this argument, observe that you gain the points from at most $\lceil (N-2)/2 \rceil$ stones.

Mark any $\lceil (N-2)/2 \rceil$ stones of your choice that are not the leftmost or rightmost stones. There is a strategy that enables you to get points that are at least the sum of points of all marked stones.

We will show this by induction. If $N \leq 3$, the claim is trivial. Otherwise, there exists a pair of adjacent stones such that one of them is marked, and the other is unmarked. If you take the marked stone, then the adjacent stones are merged into a single stone with a point maximum from both, and one of the adjacent stones is not marked. If the other one was marked, mark the new stone. Otherwise, do not mark the new stone.

By inductive argument, there is a strategy that enables you to get points that are at least the sum of points of all marked stones. If the previously marked stone had a larger point than the other, this exactly gives what we want, otherwise, this gives slightly more than what we want. ■

In conclusion, there is a strategy that enables you to gain points that are at least the sum of the top $\lceil (N-2)/2 \rceil$ stones. Obviously, you can't get more points than that. The maximum possible point can be calculated by sorting all points. The time complexity is $O(N \log N)$.

Problem G. Stones 2

Each action of obtaining points from a stone can be described as a triplet of integers $1 \leq i < j < k \leq N$, where you gain A_j points by removing a stone j which is adjacent to the stone i in the left, and stone k in the right. The color of the stone (S_i, S_j, S_k) should be either `('W', 'B', 'W')` or `('B', 'W', 'B')`. We will only consider the former case as the other can be solved symmetrically.

To obtain a situation where two stones i, k are adjacent to j when j is being removed, all elements $\{i+1, \ldots, j-1, j+1, \ldots, k-1\}$ should be removed before the j's removal, and i, k should be removed afterwards. It can be shown, that the number of permutation which satisfies these conditions is:

$$2\frac{(k-i-2)!}{(k-i+1)!}n!$$

Let this quantity be $f(k-i)$, and let W_i be an indicator which is 1 if $S_i =$ `'W'` and 0 otherwise, We can obtain an $O(n^3)$ algorithm which precomputes all values $f(k-i)$ and computes the following value.

$$\sum_{1 \leq i < j < k \leq N} W_i(1-W_j)W_k \times A_j \times f(k-i)$$

Take a prefix sum on $A_j(1 - W_j)$, and denote this value Sum_i. We can rewrite the above formula as

$$\sum_{1 \le i < k \le N} W_i W_k \times (Sum_k - Sum_i) \times f(k - i)$$

which is

$$\sum W_i W_k \times Sum_k \times f(k - i) - \sum W_i W_k \times Sum_i \times f(k - i)$$

which can be computed with two convolutions. By using FFT, you can obtain an $O(N \log N)$ algorithm.

Problem H. Beacon Towers

Let the village i be *important* if $h_j < h_i$ for all $1 \le j < i$. You can make the following two observations.

Observation 1. If a segment contains no important villages, then the division does not satisfy the rules.

Proof. For a beacon installed in such segment, there exists a preceding beacon that is higher.

Observation 2. If all segments contain at least one important village, then the division satisfies the rules.

Proof. Every segment will install a beacon on one of the important villages. The subset of important villages has increasing heights.

Let the indices of important villages be $i_1 < i_2 < \ldots < i_k$. Such sequence can be computed in $O(N)$ time.

Consider the problem now as putting the barriers between two adjacent villages $i, i + 1$ if we want to put them in different segments. By the above observation, we can install at most one barrier between two important villages. The answer is therefore $\prod_{j=1}^{k-1}(i_{j+1} - i_j + 1)$.

Problem I. Marbles

The problem can be formulated as a linear program, but the matrix is too large to directly solve with simplex. This fact hints toward a solution using flow techniques.

Consider the easier problem where there are no type-2 entries. The merging of sets can be described as a rooted binary forest, where each leaf corresponds to a single marble, and each non-leaf corresponds to a set that is a union of its two children. Type-3 entries pose a restriction on the number of red marbles in a subtree.

Consider a flow network where each marble is connected from the source, and each root of the tree is connected toward the sink. The marble is red if and only if there is a unit flow from the source to the corresponding leaf. In this modeling, the type-3 entries can be considered as lower and upper bound on capacities. This is a special case of circulation problem (LR-flow, flow with demands) and can be solved with maximum flow.

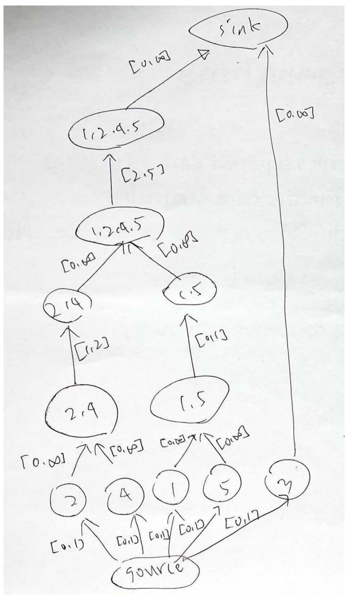

Figure 1. Illustration of above modeling for first 6 entries on sample input 1.

We need to support type-2 entries. From the vertex corresponding to a set which we will delete the marble, direct out a new edge that routes the flow from removed marbles. Here, the challenge is to maintain the same amount of flow between the two edges. For example, if the removed marble is red, there should be an outgoing flow on this edge and vice versa.

A clean way to do this is to simply remove the source and sink, and direct the edge toward the leaf node corresponding to the removed marbles. For the unremoved marbles, simply remove them at the end of the diary. If there is an outgoing flow from the removed set, then this flow directly flows into the marbles. This is exactly a circulation problem and can be solved by with maximum flow.

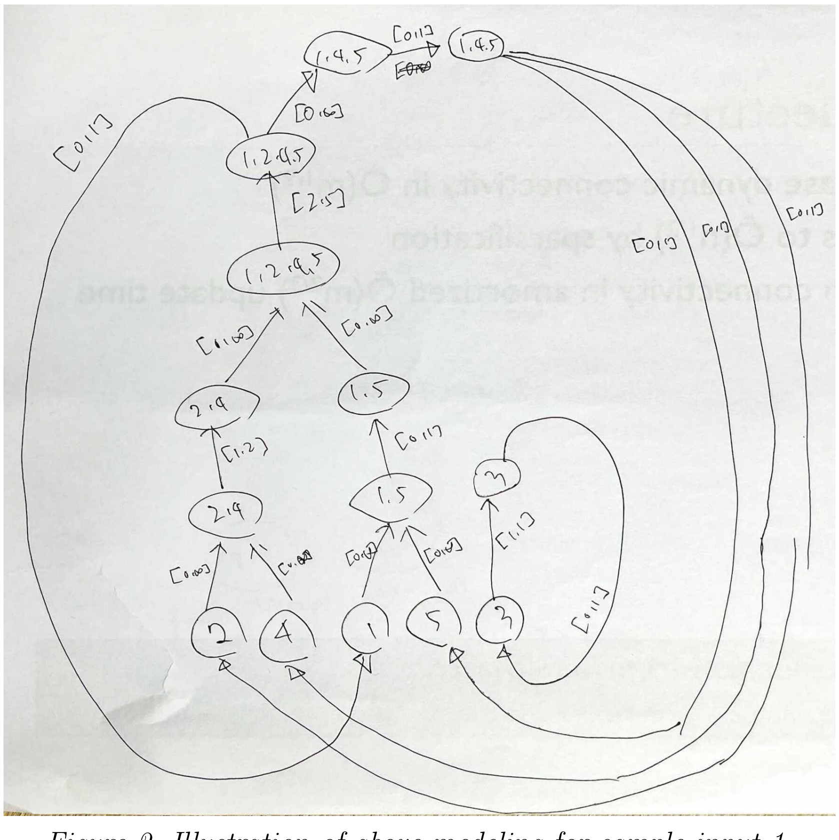

Figure 2. Illustration of above modeling for sample input 1.

Problem J. Exam

Apply meet-in-the-middle along the minor diagonal of the grid ($i + j = N + 1$). There are 2^{N-1} ways to reach some cell in the diagonal from $(1, 1)$, and 2^{N-1} ways to reach (N, N) from some cell in the diagonal. Enumerate all such paths and store the following two values:

- The maximum subarray sum of the elements of the path (x_i).
- The maximum non-empty suffix or prefix sum of the elements of the path (y_i). For a path from $(1, 1)$, we are interested in the suffix, and for a path to (N, N), we are interested in the prefix.

Let U_x be the set of paths from $(1, 1)$ to $(x, N + 1 - x)$ and D_x be the set of paths from $(x, N + 1 - x)$ to (N, N). For each cell $(x, N + 1 - x)$, we want to count the number of pairs $i \in U_x, j \in D_x$ such that $\max(x_i + x_j, y_i, y_j) = K$. This is equal to the number of pairs where $\max(x_i + x_j, y_i, y_j) \leq K$ minus the number of pairs where $\max(x_i + x_j, y_i, y_j) \leq K - 1$.

Let's count the number of pairs with $\max(x_i + x_j, y_i, y_j) \leq K$. If you ignore all paths with $y > K$, the problem is reduced to the counting the number of pairs with $x_i + x_j \leq K$. This can be solved with sorting and two pointers. The total time complexity is $O(2^N \times N)$.

Problem K. Board Game

For simplicity let's call the first player (Jeyeon) A and second player (Deokin) B.

If there is a grid with $N_i, M_i \geq 2$, take any of them and remove its last row and column. It can be shown that the winner does not change after this reduction.

Consider the case where A is in the winning position. In the original strategy, if the next move of A reaches the last column of a chosen grid, you can see that B will want to perform the next move on a chosen grid. Intuitively, the choice of B generally gives more opportunity for A to chose its next move. In the chosen grid, A cannot do any action as the token already reached the last column. Thus, the action of B does not benefit A. It is not worse to move the token from the chosen grid early to restrict A as much as possible. From this, if the move of A reaches the removed column, then the next move of B will be in the same grid. This applies to all possible state branching. By eliminating this pair of adjacent moves, you can apply the original strategy in the new grid.

As a result, the grid can be reduced to satisfy $N_i = 1$ or $M_i = 1$. Beware that the resulting grids may not have the size $(1, M_i - N_i + 1)$ or $(N_i - M_i + 1, 1)$. The above argument only holds when the corner (N_i, M_i) is inside the fence. If a removal does not make $(N_i - 1, M_i - 1)$ inside the fence, you should adjust either N_i or M_i so that the bottom-rightmost cell is reachable.

From the fence string, you can implement a query function that tells you in $O(1)$ time if you are above the upper fence, below the lower fence, or inside the reachable cell. By using this query function, you can simulate the removal procedure in $O(N + M)$ time, solving the problem in linear time.

To formally prove the main argument, you can try something similar to the surreal number. For a grid, when we are removing a last row and column, assign an integer 0 for the bottom-rightmost cell, and $-k$ if a cell is k unit above from the cell, and k if a cell is k unit left from the cell. Let X be the sum of all numbers written in each token's location. Then, it can be shown by induction that:

- If $X > 0$, A wins.
- If $X = 0$, the player who is not in a turn wins.
- If $X < 0$, B wins.

and you can observe that this is equivalent to the above algorithm.

Problem L. Make Different

Naively, you can solve the problem with a breadth-first search. Each state can be represented as a pair (x, y) denoting the position of two robots. Note that $(y - x) \pmod N$ does not change, and there are at most N different reachable states. By this observation, if the answer exists, it is at most $N - 1$.

By symmetry, assume that the first command always moves the robot clockwise (CW).

First, you can observe that the optimal solution changes the direction at most once. This holds because:

- To change the direction twice, you have to jump over the original position after first direction change.
- To do it, you have to change your direction with the first type 1 button you encountered.
- If you have a solution that goes in CW-CCW-CW direction, you can find a better solution that goes only in CCW direction.

This observation also gives a naive solution that tries all the detour points.

Second, observe that you never have to detour on the type 2 button. If your previous button was 2, this is obviously a waste. If it was 1, you can just detour on that button.

The first type 1 button you encountered by a CW walk is a special case since it can go over the original position. Suppose you detoured in the k-th type 1 button you encountered by a CW walk. In this case, you only have to care about going to the left using button 2. If you meet button 1, you either visited the $k-1$-th type 1 button (which indicates no solution of such type) or you visited a different button for each robot.

Let $LeftOne[x]$ be the smallest k such that $A[(i - 1 - 2k) \pmod N] = 1$. The button is interesting if $A[x] = A[y] = 1$, and $LeftOne[x] \neq LeftOne[y]$.

Let D be the length of CCW walk you made in the k-th interesting button. Observe that in the $k + 1$-th interesting button, you only have to make the detour of length $D/2$, and in the $k+2$-th interesting button, you only have to make the detour of length $D/4$, and so on. This shows that there are at most $\log N$ interesting buttons that you can detour on.

Now all we need is a data structure to optimize the above solution. To find the next interesting button, you can do a binary search on a sparse table with hashes. You can also find the length of the first detour similarly. Actually, the first detour can be solved a bit more easily, which is left as an exercise. The time complexity of this solution is $O(N \log N + Q \log^2 N)$.

Problem M. Short Question

As a prerequisite, let's compute the value $\sum_{i=1}^{N} \sum_{j=1}^{N} |p_i - p_j|$. By sorting a sequence p, you can get rid of the absolute value and simply compute $2 \sum_{i=1}^{N} \sum_{j=1}^{i-1} (p_i - p_j)$. Each i is added for $i - 1$ times and subtracted for $(N - i)$ times, so the answer is $2 \sum_{i=1}^{N} (2i - N - 1)p_i$. Let's denote this value as a function of sequence p as $value(p)$.

Note that $min(a, b) = a + b - max(a, b)$, and $max(|p_i - p_j|, |q_i - q_j|)$ can be interpreted as a Chebyshev distance between two point (p_i, q_i) and (p_j, q_j). It is known that the Chebyshev distance is equivalent to the Manhattan distance if the point set are rotated by 45 degrees. In other words, $max(|p_i - p_j|, |q_i - q_j|) = \frac{1}{2}(|p_i + q_i - p_j - q_j| + |p_i - q_i - p_j + q_j|)$. If we create a auxiliary sequence $a_i = p_i + q_i, b_i = p_i - q_i$, this value simply becomes $\frac{1}{2}(value(a) + value(b))$.

In conclusion, the answer is $value(p) + value(q) - \frac{1}{2}(value(a) + value(b))$. The time complexity of this solution is $O(N \log N)$.