Problem Tutorial: "Absorbing The Point"

The answer to the query is similar to the answer to the query "if the point P is inside or on the convex hull of a subsequence of the points from L to R (with the special case where all points are collinear and the convex hull is actually degenerated)".

The naive solution is to find the convex hull in $O(N^2)$ and check whether the point P is inside or on the shell. But this solution is too slow — the time complexity is $O(QN^2)$. If we will use more efficient algorithm to find the convex hull with complexity $O(N \log N)$, we get the time complexity of $O(QN \log N)$, but it is not enough as well.

We need something more smart. For every i, let find the smallest boundary of G_i so that the point P is in the convex hull of the substring from i to G_i. If there is no such substring, we will take $G_i = N + 1$. Note that the G_i array is nondecreasing, so we can find it by the method of two pointers.

We need an additional data structure that supports point insertion and extraction operations and can answer the question "whether a point P is inside a convex hull". Let's look at the case when the point P is in the convex hull of some set of points. For each line passing through P, there are two points that are on different sides of this line, or there is a point that is on this line. Let's now look at the case when the point P is not in the convex hull of some set of points. In this case, there is a line that passes through P such that all points from the set are on one side of it. Let's look at the points from this set sorted by the angle around P. If there is such a line that all the points on one side of it, we have an angle of more than 180 degrees between two adjacent points. If there is no such pair of the adjanced points — such a line does not exist.

Now we can answer questions about whether a point P is in the convex hull of a set of points, and support the removal and insertion of points by keeping the points in `std::set` sorted by angle.

We can compare points by angles if we look at which quadrant they are in and look at the sign of the cross product for points in the same quadrant. When we insert a point, the two points cease to be adjacent, and we get two new pairs of neighboring points, so we need to update the information about whether there is an angle greater than 180 degrees. Similarly, when we throw out a point, we throw out two pairs of neighbors and insert one. One exception to pay attention to is the case when a point coincides with P, then we can store information about the number of such points and not insert them into `std::set`. When using this structure we make two pointers and find the boundaries of G_i, so we can easily answer queries by comparing G_{L_i} and R_i.

The time complexity of this solution is $O(N \log N)$

Problem Tutorial: "Black and White Map"

Given a closed polyline such that no point of the plane is covered by more than two of its edges, find out a way to color some faces in black, such that out of any two neighbouring faces exactly one of them is black. The total area colored black should be minimum possible.

First we should build the resulting planar graph, find its faces and build dual graph, where two faces are connected by an edge if they share a common segment of positive legnth.

If there are no limitations on the shape of a polyline, the general approach to build planar graph will be:

1. Intersect all edges of polyline with each other.
2. Consider every edge, sort points along the corresponding line, add edges to planar graph.
3. Find faces of this planar graph with a standard algorithm that sorts all outgoing arcs by angle and then wraps faces one by one.
4. Total running time will be $O(n^3 \log n)$.

For this particular problem we are given, that no point is covered by more than two edges. That means we know that if we obtain a point as an intersection of two segments, there is no need to check whether it belongs to any other segment and reduces algorithm complexity to $O(n^2 \log n)$.

To check whether it's possible to color the graph we should check it's bipartite. If that is the case, we should pick the smaller part (by total area) to color it black.

Problem Tutorial: "Covering"

You are given a directed acyclic graph. The cost of each arc is defined as the number of outgoing arcs from its starting vertex. The cost of a path is defined as the sum of all individual costs of its arcs. We want to find a set of paths from node 1 to node n such that any arc belongs to at least one path and the total cost of all paths used is minimum.

First we will pay for each arc only once. If the number of outgoing arcs is equal to the number of incoming arcs for each node, the sum of all costs is obviously equal to the right answer. If that is not the case there will be some nodes with more incoming arcs than outgoint (call them sources) and with more outgoing arcs than incoming (sinks). However, we make an exception for nodes 1 and n, 1 is considered to be an infinite source, while n is an infinite sink.

Now we have to add some paths from sources to sinks such that for each node the number of incoming paths will be equal to the number of outgoing paths. This can be done by maximum flow minimum cost algorithms.

The total power of all sources and sinks (1 and n not included) won't exceed m, one iteration of shortest-path-augmentation algorithm (weighted Ford-Fulkerson method) will work in $O(nm)$ time if Ford-Bellman algorithm is used.

The overall running time is $O(nm^2)$ and that might be too tight for the given constraints. Instead of Ford-Bellman algorithms one can use Dinic's algorithm with Johnson's potential function and reduce complexity down to $O(m^2 \log m)$.

Problem Tutorial: "Discover And Transform"

Let's determine a codon that appears more than $N/2$ times, say it appears x times, while its complement appears y times. We want the most frequent codon in our interval to appear at least $(x - y/2)$ times more than its complement. It must not appear too many times, because then its complement becomes an element that appears more than $N/2$ times, fortunately in most cases we can find the exact interval in which the most common codon appears exactly $\lceil (x - y)/2 \rceil$ times more than the complement. The only case in which this is not possible is if N is an odd number and $x + y = N$ holds, in that case whatever interval we choose, there will be an element that appears more than $N/2$ times, so there is no solution.

If this is not the case, there is a corresponding interval, a corresponding interval that is also a prefix. Let's call $f(i)$ the difference between the number of occurrences of the most frequent element and its complement in the first and elements. It is clear that $f(0) = 0$ and $f(N) = x - y$ hold. Since $f(i)$ and $f(i+1)$ differ by at most 1, we can say that in some sense this function is continuous, i.e. there are indices for which it is 0 and $x - y$, and there are indices where it is anything in between, including $\lceil (x - y)/2 \rceil$. Therefore, there is a prefix that suits us, and we can easily find it.

We will use binary search to find i for which $f(i) = \lceil (x-y)/2 \rceil$ holds. We know $f(0) = 0$ and $f(N) = x-y$, we will check $f(mid)$. It will be valid that $\lceil (x - y)/2 \rceil$ will be located either between $f(l)$ and $f(mid)$ or between $f(mid)$ and $f(r)$ (perhaps in both places, but one is enough for us). This applies to the continuity of our function. Then we continue the binary search in the part that contains the value we are looking for.

Time/memory complexity: $O(\log N)$, number of queries: (at most) $2\lceil \log N \rceil + 3$.

Problem Tutorial: "Exploring the Maze"

Since all values are less than P, the sum maximization is equivalent to maximizing the length, and then the sequence of values (lexicographically).

Let c be a *centroid* vertex of the tree, that is, the one that minimizes the maximal distance to other vertices. One can show that for each vertex v any longest path starting at v passes through c.

Let us root the tree at c. We can now obtain a solution as follows: for each subtree of c find the largest path starting from c to the subtree (comparing lengths first, then sequences of values lexicographically). Now, for every vertex v the best path starts with going straight to c, and then proceeding to a subtree of c (different from where we have come from) with the best path for the subtree.

We can see that only two best paths for subtrees of c should be saved (since at most one of them if forbidden for each v).

The complexity is $O(n^2)$ since for each vertex v we have $O(1)$ candidate paths, and comparing two paths may takes $O(n)$ time.

We can improve on the comparison complexity in several ways. Note that to compare two sequences it suffices to find the largest common prefix, and compare the subsequent elements.

- Finding the longest common prefix can be done with polynomial hashing, and binary search on the length of the answer.
- A hash-free solution to this is to construct a "suffix-array-like" structure, that is, classify all substrings of length 2^k. With this, a binary descent approach allows to compare strings in $O(\log n)$ time.

A small setback is that all our sequences consist of two parts: paths going from a vertex towards c, and downward paths. This is a technical issue, and can be resolved without changing the solution much. The total complexity is $O(n \log n)$.

Problem Tutorial: "Favorite Restaurants"

Construct the condensation of the graph. The conditions imply that we pay x_u whenever we stay in the same SCC (strongly connected component), and y_u otherwise.

For a single SCC i, compute $s_{i,k}$ — the answer if we are restricted to stay in the SCC i. To compute $s_{i,k}$ we try all options for first vertex v (which will contribute y_v) summed up with $k-1$ smallest values of x_u among vertices u in the same CSS.

Now, compute $d_{i,k}$ — the complete answer if we have to start in the CSS i. Clearly, $d_{i,k} = \min(c_{i,k}, \min_{i \to j, 1 \le k' < k} c_{i,k'} + d_{j,k-k'})$, where the summation is over all CSS j reachable from i.

The total complexity is $O(n^3)$ (with good constant).

Problem Tutorial: "Game For One"

First, let's denote by $dp_{i,j}$ the maximal solution if after event j 1 is at position i. At the beginning, $dp_{i,0} = -\infty$ (a sufficiently small negative number) for every i on which 1 is not located, while $dp_{i,0} = 0$ is valid for i, which is the initial position of 1. Now we need to find the recursion. There will be two cases, for two different event types:

- 1 l r: $dp_{i,j} = max(dp_{l,j-1}, dp_{l+1,j-1}, \dots dp_{r,j-1})$, if $i \in [l, r]$. Otherwise $dp_{i,j} = dp_{i,j-1}$. Therefore, any position on the interval can appear from any other position on the interval, and it is optimal to take from the maximum.
- 2 x: $dp_{x,j} = dp_{x,j-1} + 1$, and $dp_{i,j} = dp_{i,j-1}$ if $i \neq x$. It increases the dp value when the 1 is right there, the other elements don't change.

The memory complexity: $O(NM)$, time complexity: $O(NM)$, that is not sufficient for the given constraints. The time complexity can reduced in the following way: we go through the events one by one and maintain an array of dp_i, for the events passed so far, which is the best solution, if 1 is currently at position i . We can maintain this sequence using a segment tree with lazy propagation, which has operations: "find the maximum on the interval" and "set the entire interval to some value". Event of type 1 is implemented

by the operation "set interval to value" applied to the result of "find the maximum on the interval", while event of type 2 is implemented by the operation "set interval [x,x] to the value of $dp_x + 1$".

Final memory complexity: $O(N)$, final time complexity: $O(M \log N + N)$.

Problem Tutorial: "Horizontals and Verticals"

Let $g_{x,y}$ be 1 if the cell (x, y) is originally black and 0 otherwise. Also let $a_{x,y}$ be 1 if we apply the operation to this cell, or 0 otherwise (clearly, it doesn't make sense to use the same cell twice).

If the sequence of operations is good, we must have the sum $g_{x,y} + \sum_{p \in N(x,y)} a_p$ even for each cell (x, y), where $N(x, y)$ is the set of cells in the same row and/or column with (x, y).

Finding a suitable set of values for $a_{x,y}$ can be reworded as a linear equation system modulo 2, with nm variables and equiations. However, straightforward Gaussian elimination takes $O((nm)^3)$ time and gets TL.

We can optimize it w times (where w is the length of a bitword and is either 32 or 64) by using bitsets for storing rows of the matrix, this is fast enough.

Problem Tutorial: "Incident Vertices"

For $P = 0$, all queries are known in advance, so we can solve them in any order. When P = 1 we have to answer the i-th query to find out the $i + 1$-th query, so we need to solve the problem *online*, that is, we answer the queries in the order in which they were given.

The path from A to B in the tree we can split the path from root to A, plus the path from root to B, minus the path from root to $LCA(A, B)$, minus the path from root to $parent(LCA(A, B))$. We need to find a way to get the sum of the $f(i)$ of all nodes on the path from the root to some node. Consider a Euler tour tree of the given tree and put the nodes in the array in the order in which we visited them in DFS, thus, each subtree will make up one subarray. During DFS, we can also remember subarray boundaries for each subtree.

So the queries are reduced to the queries for subarrays on the arrays S and T. Consider the subarray $L_1 \dots R_1$ of the array S and $L_2 \dots R_2$ on the array T. If we will label the position i in the array T with 1, if T_i is in the substring $L_1 \dots R_1$ of the array S, and 0 otherwise, the answer of query is the sum of the labels in the substring $L_2 \dots R_2$ of the array T.

We can use the persistent segment tree (or Fenwick tree in the offine case) to keep those arrays, and after the decomposition into 4 paths we are dealing with the prefixes of one big array, so we can sort them and reduce the total number of changes in our data structure to $O(N)$.

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

Problem Tutorial: "Journeys With Rent-a-car"

The solution for a city is equal to the solution of the city with the highest D_i among the cities it can visit. Consider a city i and the set S of all cities that cannot visit a city with D greater than its own. Let j be the city with the smallest D among the cities from the set S that can visit city i. The solution for i is the same as the solution for j. We will prove the previous statement by showing that i can visit j. If i can visit a city k with $D_k \geq D_j$, it can also visit j by going to k, transferring to a vehicle from that city, returning to i and going to j by the reverse route that would have taken from j to i (we know that on that path all $L_e \leq D_j \leq D_k$ because j cannot visit the city with larger D). If $D_k < D_j$ for all cities that i can visit, then there is a city in the set S that i can visit and has less D than city j, which is a contradiction.

Now we know that the solution for each city is equal to the solution for the smallest D-valued city in the set S that can visit that city, and we can solve the problem as follows. Let's process the cities in increasing order of their D values. When we process a city and add all the edge e that have L_e less than or equal to D_i. For each connected component, let's store the largest D for the nodes in it. If the largest D for

the nodes in the component of node i is equal to D_i, node i is in set S and its solution is the size of the connected component. It is also the solution for every node in this component that is not already in the component of some node from the set S. We can maintain these cities for each component in the sequence and switch them from the smaller to the larger component when the merging of the two components occurs.

In this way, we get a solution that works in time complexity $O(N \log N + M \log M)$.

Problem Tutorial: "Ksir"

First, note that if there are two adjacent peaceful vertices, then the game is a draw. Indeed, any of the players can maintain so that both of these vertices have a positive numbers of knights by redistributing, hence the game will never end.

If no two peaceful vertices are adjacent, then the game can be considered a direct sum of individual peaceful vertices. We can compute Grundy values for each vertex (assuming all its neighbours are aggressive). The values will never exceed 40.

The problem now is to compute the number of independent sets in the graph, and out of these count the number of sets with total Grundy value 0.

First, let us simply count the independent subsets. Divide the vertices into two parts A and B. Try all independent subsets I of A directly (in $O^*(2^{|A|})$); each of them confines vertices in B to a subset B_I. The answer is $\sum_I i(B_I)$, where $i(S)$ is the number of independent sets confined to S. This can be computed in $O(|A|2^{|A|} + |B|2^{|B|})$ with fast subset convolution (or, less efficiently, in $O(|A|2^{|A|} + 3^{|B|})$).

To account for the Grundy values we simply apply the previous approach for independent sets $I_A \subseteq A, I_B \subseteq B$ with $gr(I_A) = gr(I_B) = x$ for each x. Note that x does not exceed 63.

The resulting complexity is $O^*(2^{n/2})$ (or, alternatively, $O^*(1.529...^n)$).

Problem Tutorial: "Lexicographically Maximal Sequence"

Let's note the following:

Let $P[i]$ be the prefix bit xors of the sequence A (meaning $P[i] = A[1] \oplus A[2] \oplus \ldots \oplus A[i]$). Now if we perform the operation from the statements on the element i, we can notice that it replaces the values of $P[i-1]$ and $P[i]$. Then, if we start to count the new values $P'[i-1] = A[1] \oplus A[2] \oplus \ldots \oplus A[i-1] \oplus A[i] = P[i]$ and $P'[i] = A[1] \oplus A[2] \oplus \ldots \oplus A[i-1] \oplus A[i] \oplus A[i] = P[i-1]$, while all other values of the array $P[i]$ remain unchanged, because the effect on the elements $A[i-1]$ and $A[i+1]$ is shortened (or not affected at all).

Now let's note that we can permute the first $N-1$ prefix xors of this sequence in an arbitrary way, and how for a given sequence of prefix xors we can uniquely reconstruct the sequence $(A[i] = P[i] \oplus P[i-1])$, such strings that are obtained by permuting the prefix xors are precisely the only strings that can be obtained (whence it follows that the largest number of different strings that can be obtained is $(N-1)!$).

We can greedily choose the permutation of prefix xors so that the lexicographically largest sequence is obtained. At any moment, we go through the remaining prefix xors and simply choose the one that gives us the highest xor with the previously selected one (note that the last prefix xor cannot be changed) but the time complexity $O(N^2)$ is not enough.

Let's insert all the prefix xors into a binary trie. Namely, let's represent each number as the binary string of length 60 (with potential leading zeroes). Now we insert all those values into the trie, then it is possible to very quickly find the element that gives the highest xor with a given number x, by going down the trie and greedily choosing to go to the child with the opposite digit, if there is one. It is also possible to quickly drop a number from the structure.

I.e. we will just build the trie and go in order, always choosing from it the element that gives the highest possible xor with the previous value, and then we throw it out.

The final time complexity is $O(N \cdot \log \max A_i)$.

Problem Tutorial: "Matches"

We are given a $r \times c$ table, some cells have chips. Two cells are neighbouring if they share a common point, thus each cell has no more than 8 neighbours. Some chips located in neighbouring cells are connected by edges. No two edges intersect and it's possible to get from any chip to any ohter chip travelling only along this edges. Count the number of ways to remove one edge and add one new edge such that all the conditions listed above still hold.

Consider the case there are no bridges in this graph. Then, we can remove any edge and place it to at any place. We should only worry about intersection condition. Intersection might only be the case for two diagonal edges. This case is easy to deal with as we just keep track on the number of valid position for inserting new edges.

However, if we remove any bridge from this graph, we have to count only such ways to place edge back that make graph connected again.

To count this fast we would like to find biconnected components. This can be done by removing all bridges from the graph and coloring the remainin connected components. Now, we simply try all possible ways to insert new edges and keep track on which pair of biconnected components it connects.

Total running time is $O(r \cdot c)$ if we use hash-table to query precomputed result for pairs of components.