Problem F. Safe Road

Solution for the first group of tests Since the value of n does not exceed 20, we can solve this group of tests by brute force in $O(2^n)$. We enumerate all possible strings of length n consisting of 0s and 1s. Let the torch with index i be turned on if there is a 1 at the i-th position of the string. Then we check whether the turned-on torches cover the entire road. If yes, then this is one of the valid solutions. From all valid options, we choose the one where Pankrat spends the least emeralds.

Solution for the second group of tests We solve the problem using dynamic programming. Let $dp[i]$ be the minimum number of emeralds that need to be spent to illuminate the road from the first to the i-th block by placing a torch on block i. The base of dynamic programming are cases when we need to illuminate blocks from the first to the k-th – for this, one torch is sufficient, so the base is easy to determine. For the transition, we find the minimum among the values $dp[j]$ on the segment from $i-2k-1$ to $i-1$. This can be done by a linear pass through the segment, which gives a transition complexity of $O(k)$. As a result, we get an overall running time of $O(nk)$. The answer will be the minimum value of $dp[i]$ on the segment $[\max(0, n-k-1), n-1]$.

Solution for the third group of tests We improve the previous solution. The minimum on a segment can be found using multisets. We go through the array dp, maintaining the needed segment $[i-2k-1, i-1]$ for the current i. Deletion and addition operations occur in $O(\log k)$, so the overall running time will be $O(n \log k)$.

Complete solution Note that the segment on which we need to find the minimum always has a fixed length. Then the transition task in dynamics reduces to the task of finding the minimum on a sliding window of fixed length. This task is well-known and is solved using a double-ended queue, which allows getting the minimum in constant time. Thus, the final complexity of the solution is $O(n)$. You can also write a segment tree from below. This solution, although it works in $O(\log k)$, with careful implementation gets the complete solution.

Problem G. Chord Filling

All free points on the circle are divided into groups by the already drawn chords — in each group, any two points can be connected by a chord that does not intersect any of the existing chords.

Obviously, in a group consisting of m vertices, you can draw $\left\lfloor \frac{m}{2} \right\rfloor$ chords — for example, by dividing all points into pairs of adjacent points and connecting them.

From there, it's just a matter of implementing this carefully.

The first subtask ($k = 0$) has an obvious solution.

In the second subtask ($k = 1$), all free points are divided into two groups, and it's easy to connect them with the maximum number of chords.

In the third subtask ($k = 2$), the points are divided into no more than four groups, so each group can still be handled manually.

According to the author, the simplest way to solve the full problem is to recursively divide it into two smaller subproblems using the "divide and conquer" principle: each drawn chord splits the considered points into two groups, and each group is processed recursively. The recursion ends when there are no drawn chords in the current group — then, just connect those points with the maximum number of non-intersecting chords.

Problem H. Chase

To solve the first subtask, it's enough to note that if $t_1 < a$, then either Timur will reach Alyosha by himself, or it will be impossible to catch him. If $t_1 > a$, then Timur cannot be caught, because if he is in a city v (where $v < n$), the distance between the two players cannot decrease, and if he is in city n, then the game ends the next day.

To solve the second subtask, notice that the given graph resembles a tree, and we can use the fact that

the cheapest path to a city is also the shortest one. This means we can traverse all the graph's vertices using depth-first search (DFS) and store the cost and the distance from Alyosha's starting city in each of them.

To solve the third subtask, we need to try all possible paths Alyosha can take. For that, we can use backtracking.

To solve the fourth subtask, we need to efficiently store the path costs from the starting city to all others for every possible path length. First, note that the path length cannot exceed $n-1$, because otherwise it would mean we visited some city more than once. Accordingly, let's define an array `d[cnt][v]`, where `cnt` is the number of days, and `v` is the city. Initially, `d[0][a] = 0`, and all other values are set to infinity. Next, we need to fill in all values optimally. Since the constraints are small, we can use either Dijkstra's algorithm for sparse graphs or the Bellman-Ford algorithm.

Problem I. Profit

Note that we can run a binary search on the value

$$P_{i_1 i_K} = \frac{A_{i_1} + A_{i_2} + \cdots + A_{i_K}}{B_{i_1} + B_{i_2} + \cdots + B_{i_K}}.$$

During each step we check whether there is a path in the tree that satisfies the inequality $P_{i_1 i_K} \geq X$, where X is the current probe value of the binary search. Rewriting the inequality gives

$$A_{i_1} + A_{i_2} + \cdots + A_{i_K} \geq X(B_{i_1} + B_{i_2} + \cdots + B_{i_K}) \Rightarrow$$

$$\Rightarrow (A_{i_1} - XB_{i_1}) + (A_{i_2} - XB_{i_2}) + \cdots + (A_{i_K} - XB_{i_K}) \geq 0.$$

Hence the task reduces to finding a simple path in the tree whose total weight under the transformed edge metric $(A_i - XB_i)$ is non-negative. This feasibility check can be carried out efficiently, for example with subtree dynamic programming or centroid decomposition.

Because the binary search is performed over real (floating-point) numbers, it needs about $O(60)$ iterations. The check at each iteration runs in $O(n)$, leading to an overall complexity of $O(60 \cdot n)$.

Problem J. Alien Prison Break

First, let's restate the problem a bit. For every query we have to find the shortest *safe* route from point A to point B. A route is *safe* if it never passes strictly through the segment determined by any two towers, and never enters the interior of the triangle determined by any three towers. In other words, the *safe* route must stay outside the *convex hull* of all towers mentioned in that query.

Subtask 1

In this subtask, it was guaranteed that all towers are located in the third quadrant of the plane, while points A and B are in the first quadrant. This means the answer is simply the length of the segment AB.

Subtask 2

Since in this subtask $L = R$, there are exactly 2 towers on the plane in each query. Let's denote them as points C and D. Then, there are two cases — the segment AB either strictly intersects CD, or it doesn't. In the first case, the answer is the minimum of $|AC| + |CB|$ and $|AD| + |DB|$. In the second case, the answer is simply $|AB|$. One also had to consider the edge case where AB lies strictly inside CD.

Subtask 3

In this subtask, $R - L \leq 1$, meaning the number of towers on the plane does not exceed 4. The idea here was to carefully go through all possible cases.

Subtask 4

In all queries, all cloning vectors were used, and point A was the same as point B. This means we needed to somehow precompute the convex hull of all 2^n tower positions in advance, and then, for each query,

check whether point A lies strictly inside this convex hull. So how can we construct it? Notice that when we clone towers, we're effectively performing a special case of the *Minkowski sum* with a two-sided polygon — this polygon is our cloning vector set. So, if we duplicate all vectors in the opposite direction, sort them by polar angle, and find the "lowest-leftmost" tower, we can build the desired convex hull.

However, there's a small issue — the lowest-leftmost tower depends on the *starting* tower, which is different in every query. This is easily handled if we assume that the starting tower is always located at the origin $(0, 0)$.

Given the actual coordinates of the starting tower S in a query, we can simply translate points A and B by subtracting S, i.e., use $A - S$ and $B - S$, where the subtraction is done coordinate-wise.

Now that we have the convex hull of all 2^n tower positions, we can check whether point A lies strictly inside it in $O(\log(n))$ time using a simple binary search.

Subtask 5 coordinatecom In this subtask, everything is the same as in *Subtask 4*, except that the points A and B in the queries are not the same. First, check whether these points lie strictly inside the convex hull. If they do, the answer is immediately -1. Otherwise, we need to be able to find tangents from a point to the convex hull. There are several well-known algorithms that can do this in $O(\log(n))$ time. Once we've found the tangent lines from points A and B in a given query, there are two cases: the segment AB strictly intersects the convex hull and if it does not. To check whether AB intersects the hull, use the following:

- Assume that point A lies strictly outside the hull. Let the rays AA_1 and AA_2 be the tangents from A to the polygon (where A_1 and A_2 are the contact points with the polygon). Then the segment AB strictly intersects the convex hull if and only if segments AB and A_1A_2 strictly intersect.

If AB does not intersect the convex hull, the answer is simply the Euclidean distance $|AB|$. Otherwise, we need to find the shortest path that: Starts at point A, follows one of the tangents to the hull, travels along the polygon edges from one tangent point to the other, and finally goes to point B. There are exactly four such routes. Their lengths can be computed using prefix sums over the polygon's side lengths.

There are also many special cases to consider when points A and B lie on the polygon's boundary, which must be handled carefully.

Subtask 6

In this subtask the point A is always the same as B, but L and R vary from query to query. Thus, we need an efficient way to maintain the current convex hull of all vectors whose indices lie in $[L, R]$.

Let's process the queries offline with Mo's algorithm. Keep a treap whose keys are the vectors themselves. Inserting or deleting a vector takes $O(\log(n))$. The treap keeps all vectors sorted by polar angle, so the coordinates of the i-th hull vertex can also be obtained in $O(\log(n))$ by summing the first i vectors coordinate-wise.

Maintaining the up-to-date convex hull therefore costs $O(q * \sqrt{n} * \log(n))$.

To decide, for each query, whether the point A lies inside the hull, use the binary-search test from *Subtask 4*, adding another $O(q * \log^2(n))$ overall.

Subtask 7

For this subtask it suffices, per query, to rebuild the convex hull and compute the tangents in $O(n)$.

Subtasks 8–9 (full solution)

To obtain the full solution, we combine the approaches from Subtasks 5 and 6. Since we can determine the coordinates of any vertex of the convex hull in $O(\log(n))$, we can also compute tangents in $O(\log^2(n))$. Additionally, we need to compute distances between convex hull vertices, which is also handled using a treap.

The overall time complexity is $O(q * \sqrt{n} * \log(n) + q * \log^2(n))$, with a large constant due to the treap, which is enough for a 90-point solution.

To get the full 100 points, the treap must be replaced with a Fenwick tree (Binary Indexed Tree).