Problem A. Alice and Bob

Long way

Let's try to find the d_v = number of Alice moves − Bob moves, which the one token on vertex v will give (we will call this thing "`value`" of the game for this token). The value of the whole game is then the sum of values of all tokens.

Our invariants will be the following:

If the value of the game is > 0, then Alice will always win.

If the value of the game is $= 0$, then the second player will always win.

If the value of the game is < 0, then Bob will always win.

Our values should satisfy the following properties.

If the value of the game is > 0, then on the turn of Alice, she should be able to move to the state of the game with value ≤ 0, and on the turn of Bob, all moves of Bob should lead to the states with value > 0.

If the value of the game is $= 0$, then on the turn of Alice, all her moves should lead to the values > 0, and on the turn of Bob, all moves of Bob should lead to the states with value < 0.

The case when the value of the game is < 0 is symmetric to the case with > 0.

So we can understand that if there is an edge from white vertex $v \to u$, where $d_u < 0$, then it is better for Alice to never move a token from vertex v to vertex u.

Also, if there is an edge from white vertex $v \to u$, where $d_u \geq 0$, when d_v should be $\geq d_u + 1$ for Alice.

So we can understand (and prove that all properties are satisfied by not hard induction), that if vertex v is white, $d_v = \max(0, \max(d_{u_i} + 1))$.

Similarly, if vertex v is black, $d_v = \min(0, \min(d_{u_i} - 1))$.

Short way

This game is a partisan game, so let's try to analyze it using surreal numbers, d_v will be equal to the surreal number, corresponding to the game, which contains only one token, on vertex v. It appears that all these values are integers.

If vertex v is white and there are are several possible outgoing edges $v \to u_1, v \to u_2, \ldots, v \to u_k$, then d_v is equal to $\{d_{u_1}, d_{u_2}, \ldots, d_{u_k}|\}$, which is equal to $\max(0, \max(d_{u_i} + 1))$ for integers.

Similarly, if vertex v is black, $d_v = \min(0, \min(d_{u_i} - 1))$.

Summary

The set of tokens on vertices $v_1, v_2, \ldots, v_k$ is good for Alice is $\sum d_{v_i} > 0$.

All values are in the range $-n \ldots n$, so using a simple knapsack DP we can find the required answer in $\mathcal{O}(n^3)$.

Problem B. Brackets

Let's assume that all brackets initially are equal to ')'.

And then we will try to set the opening bracket greedily, in the order $1, 2, \ldots, 2n$. We will try to set i-th bracket to '(' if it is possible.

If we assume that i-th bracket should be '(', $t_i = 1$, otherwise $t_i = -1$.

Then, if $t_i + t_{i+1} + \ldots + t_{2n} > 0$ for some i, it is not possible to set i-th bracket to '('. Because we assume that we set everything correctly for brackets $1, 2, \ldots, i-1$ (and their pairs), and all other brackets are set to ')', so this value for suffix is the smallest possible, and if it is > 0 it means that on this suffix number of opening brackets will always be larger than the number of closing brackets, so it can't be a correct bracket sequence.

Otherwise, if the answer exists, we always can set i-th bracket to '('.

To make this solution fast enough, we can use the segment tree data structure.

Total complexity is $\mathcal{O}(n \log n)$.

Problem C. Circles

We can formulate this problem as a linear programming problem.

$x_i + x_{i-1} \le s_i$

$x_1 + x_2 + \ldots + x_n \to \max$

We can build a dual problem.

$y_i + y_{i-1} \ge 1$.

$\sum y_i \cdot x_i \to \min$

All optimal solutions to this problem are half-integer, $y_i \in \{0, 0.5, 1\}$.

As there are only three possible states of each variable, we can use dynamic programming, where we will maintain the state of the first variable and the state of the i-th variable, in the transition we should check that the sum of two adjacent variables is at least 1, and in the end, we should take the min among all states where the sum of the first and the last variables is at least 1.

Note that to find these values for all prefixes, we don't need to recalculate DP completely, we can just process the transition to the next layer.

The total complexity is $\mathcal{O}(n)$.

Problem D. Deja Vu

We will solve this problem offline.

Let's move on the array from the left to the right, and maintain the set of active queries (query j is active if $i \ge l_j$).

For each query, we will maintain four values:

a_i, b_i, c_i: the smallest value of element on which some LIS of length $1, 2, 3$ ends.

Note that for fixed element, its value is fixed on some segments of queries.

For the fixed segment of queries $l \ldots r$, for each active query i on this segment we need to recalculate these values for the fixed value x of the current element.

Process X

- If $a_i \ge x$, $a_i = x$.
- Otherwise, if $b_i \ge x$, $b_i = x$.
- Otherwise, if $c_i \ge x$, $c_i = x$.
- Otherwise, we found the answer to the i-th query! We should say that it is no longer active and remember the answer to it.

"`If` $a_i \ge x$`,` $a_i = x$", there is a well-known data structure for such queries.. Segment tree beats! In fact, we can modernize it to solve the problem in $\mathcal{O}(n \log n)$.

At first, let's note that if the query is not active we can assume that $a_i = b_i = c_i = -\inf$, and forbid to ban not active queries, and when the query is becoming active set $a_i = b_i = c_i = \inf$, anyway, these are just not hard implementation aspects.

In this problem, as usual, we will maintain the maximum a_i and the second maximum a_i.

Then, you should split on the segment, and you will obtain some vertices of the segment tree, and for each such vertex, you should proceed to the next iteration of the process X either for all queries on this vertex or for all queries besides the queries with maximum a_i.

To proceed to the next iteration, we should maintain the maximum b_i and second maximum b_i for all i, besides i's with the maximum a_i, and also these values for all is with maximum a_i.

Of course, it is not hard to recalculate these values.

And using these values you can continue splitting the current vertex into some vertices, to achieve the situation where among the correct set of a_i (one of the two possible sets), you should proceed to the next iteration of X either for all queries or for all queries besides the queries with maximum b_i.

Similarly, we can maintain the information about c_i for a proper set of b_i.

And then, you can find naively all **interesting** c_i, (where $c_i \neq -\inf$, as we discussed before), such that $c_i < x$, and ban these queries, and set $c_i = x$ to all other queries.

(Of course, you don't need to hardcode all these cases for all sets of a and b), you can maintain structs for values about c_i, and struct for values about b_i, which maintain two previous structures, and similarly for a_i.

Also, you need to write lazy propagation to update these values, but it is not a hard implementation detail too.

It is not hard to estimate the time complexity of this solution, each time when you split inside the vertex, the number of different a_i/b_i in this segment decrease, so the total complexity is $\mathcal{O}((n+q)\log(n+q))$

Problem E. Easiest Sum

Let's try to solve the following problem: for fixed x find the minimum number of coins that you should spend to make the sum on all subsegments $\leq x$ (we will call this value $c(x)$).

We will introduce the variable p_i: the number of coins that you spent on the prefix $1, 2, \ldots, i$.

Obviously, $p_{i+1} \geq p_i$.

Also, $p_j - p_{i-1} \geq$ `Sum on the segment` $i \ldots j$ `-` x

All constraints have $i < j$.

So we can set the proper values greedily for $i = 1, 2, \ldots, n$, by setting $p_i = p_{i-1}$ and relaxing it with $p_{j-1}+$ `Sum on the segment` $j \ldots i$ `-` x.

We can formulate it as the longest path in the DAG, we will convert it to the shortest path problem for simplicity.

- $i \to j$ has the weight x `- sum on the segment` $[i; j)$.
- $i \to (i+1)$ has the weight 0.

The shortest path on this graph is equal to the shortest path in the graph $-\sum a_i$:

- $i \to j$ has the weight x.
- $i \to (i+1)$ has the weight a_i.

Note that $c(x+1) - c(x)$ is monotone and $\leq n$, so if for each k we will find the smallest x such that $c(x+1) - c(x) = k$, we will be able to find the answer with some formulas.

$c(x+1) - c(x) = k$, when the shortest path in this graph passes through exactly k edges of type x.

If for each k, we will find the shortest path length t_k in this graph that passes through exactly k edges of type x (ignoring the weight x), then for fixed x we will be able to find minimum $c(x)$ as the smallest

$xk + t_k$, and it won't be very hard to find the required values for each k, (for example, using the fact, that t_k is convex too).

How to find t_k? Note that each edge of $i \to j$ is, in fact, some subsegment of the initial array, so the shortest path is, in fact, equal to $\sum a_i$ - largest sum of k disjoint subsegments.

So we just need to find is the largest sum of k disjoint subsegments for each k!

This problem is well-known (even for segment queries `:)`), we can find it by optimizing MinCostMaxFlow using segment tree (each time find the subsegment with the largest sum and invert it).

Also, some other solutions, optimizing the problem itself (by picking optimal position for each coin each time), but in fact, most of them appear to be the same as this solution.

The total complexity is $\mathcal{O}(n \log n)$.

Problem F. Funny Salesman

Note that the highest bit will present in the optimal path the maximum number of times.

Let's find connected components in the graph, where we will leave only edges without the highest bit.

OR on the path between two vertices will contain the highest bit if and only if these two vertices are from the different connected components.

If the sizes of the connected components are $a_1 \le a_2 \le \ldots \le a_k$, if $a_k \le a_1 + a_2 + \ldots + a_{k-1} + 1$, then we can reoder all vertices in such way, that highest bit will present in all adjacent indices.

Otherwise, $\ge a_k - (a_1 + a_2 + \ldots + a_{k-1} + 1)$ adjacent vertices on this path (note that they all will be from the k-th connected component) won't contain the highest bit.

We can proceed to the largest component, our current state will be the set of vertices and the number of elements k that we need to choose, initially our state will be the complete tree and $k = |V|$.

If $|V| - 1 - \max(0, a_k - (a_1 + a_2 + \ldots + a_{k-1} + 1)) \ge k - 1$, then we can add the current `highest bit` $\cdot(k-1)$ to the answer. Otherwise, we should add $(|V| - 1 - \max(0, a_k - (a_1 + a_2 + \ldots + a_{k-1} + 1))) \cdot$ `highest bit` to the answer and proceed recursively to the k-th component with corresponding set of vertices and new k equal to $\max(0, a_k - (a_1 + a_2 + \ldots + a_{k-1})))$.

Problem G. Graph Coloring

For vertex v, let s_v will be the set of outgoing colors from it.

For each edge $u \to v$, we can set to it any color, which exists in s_u but don't exists in s_v.

We can't find the color to this edge if and only if s_u is a subset of s_v.

But if $s_1, s_2, \ldots, s_n$ will be different sets of 7 elements among all subsets of the 14 colors, there will be no bad pairs.

The number of possible sets is $\binom{14}{7} \ge 3000$.

Bonus 1: prove that it is not possible to color tournament with 3000 vertices into < 12 colors.

Bonus 2: find some tournament with 3000 vertices which is impossible to color into 13 colors.

Problem H. Hidden Graph

Let's add vertices one by one and maintain the induced subgraph on the first i vertices.

Let's assume that we've already added vertices $1, 2, \ldots, i$, and know the induced subgraph on them, and now we want to add vertex $i + 1$.

`Any induced subgraph contains a vertex with a degree at most` k.

It means that we can color the graph into $k + 1$ colors (take the vertex with the minimum degree, delete it, color the remaining graph, insert the vertex back (and pick some suitable color for it)).

$k+1$ colors $\rightarrow$ $k+1$ independent sets.

Let's pick one of these independent sets, and find all edges from i to it. For this purpose, we can ask this set + vertex i, if i is not connected to any vertices from this set, it will return $-1-1$, otherwise, we will find some edge $i \rightarrow v$, we can delete v from this set, and ask again...

Using these queries, you will find each edge exactly one time. Also, for each vertex, there will be $\leq (k+1)$ additional queries (because each independent set will return $-1-1$ exactly one time).

The number of edges is $\leq nk$.

The total number of queries is $\leq nk + n(k+1) \leq 2nk + n$.

Problem I. Insects

Essentially, the problem essentially asks us to compute the size of a minimum vertex cover in a bipartite graph. Konig's theorem states the following:

Theorem 1 (Konig). In a bipartite graph, the size of a minimum vertex cover equals the size of a maximum matching.

Now, we have some dynamic left part of some bipartite graph, and we need to be able to find the max matching after each update.

We will solve this problem offline, using Divide and Conquer. To do it, we present proof of Theorem 1 and several lemmas.

Proof of Theorem 1. Let's take a maximum matching M. For each edge in M, one of the vertices should be in some vertex cover. So any vertex cover has size at least $|M|$.

We prove there exists a vertex cover of size $|M|$. Let's create a directed graph with the same edge set, where edges are directed right-to-left if it is in the matching, and left-to-right otherwise. Let's do DFS from all vertices of the left part not covered by M. We call Z as the set of all visited vertices. Let (L, R) be a bipartition of the graph. We'll denote $L_+ = L \cap Z, L_- = L \cap (G - Z), R_+ = R \cap Z, R_- = R \cap (G - Z)$.

Notice that $L_- \cup R_+$ is a vertex cover. This is true since $L_+ \cup R_-$ is an independent set: If there exists some edge $e \in L_+ \cup R_-$, $e \notin M$ by definition, and then $e \cap R$ is reachable from $L \cap Z$, contradiction.

Now we show $|L_- \cup R_+| = |M|$. By definition, $L_- \subseteq V(M)$. Also, $R_+ \subseteq V(M)$. Suppose there exists some vertex in R_+ not in M. Then the path that reaches that vertex is an augmenting path. Finally, observe that for all edges in M, either both endpoint is in Z or both are not in Z.

Corollary 2. The set of vertices $L_+ \cup R_-$ is the maximum independent set.

Lemma 3. If we will leave only vertices $Z = L_+ \cup R_+$ in the graph, then any maximum matching on this graph will cover vertices of R_+.

Proof: From the proof of Theorem 1, $R_+ \subseteq V(M)$, and the edges covering R_+ belongs to L_+, so there is a maximum matching of size $|R_+|$.

Lemma 4. If we will leave only vertices $Z = L_- \cup R_-$ in the graph, then any maximum matching on this graph will cover vertices of L_-.

Proof: From the proof of Theorem 1, $L_- \subseteq V(M)$, and the edges covering L_- belongs to R_-, so there is a maximum matching of size $|L_-|$.

We will assume that the left part is indexed in the queries order, so the first k vertices are white ants $1, 2, \ldots, k$.

Lemma 5. The answer for each query is simply an intersection of lexicographically smallest maximum matching (with respect to index in L) with a prefix of L.

Proof. Easy to see from the Kuhn algorithm or from the Transversal matroid.

Definition (Best Matching). Best matching is the lexicographically smallest maximum matching with respect to index in L.

Lemma 6. Let A_k be the first k vertices of the left part. Consider a subgraph of vertices $A_k \cup R$. Let A_-, A_+, R_-, R_+ be the partition of $A_k \cup R$ as defined in Theorem 1 (here we will write as A_- instead of L_-). Then for any best matching M_+ of $A_+ \cup R_+$, there exists some best matching in $A_k \cup R$ that covers M_+.

Proof. Note that we can't find any matching better than M_+ in $A_+ \cup R$: $A_+ \cup R_-$ is an independent set. Thus it suffices to show that M_+ can be extended. Take any maximum matching from $A_- \cup R_-$: By Lemma 4 any maximum matching is best matching in $A_- \cup R_-$. If we combine them we get a maximum matching of $A_k \cup R$, since $|L_-| + |R_+| = |M|$. And since the options are independent (in a sense that edges passing $A_- \cup R_+$ can be ignored without changing the lexicography) We can simply take the union to find the best matching in $A_k \cup R$.

Now, we will finally discuss the algorithm.

We will build the recursive function $f(L, R, x)$: add the best matching for $L \cup R$ to the answer, if we know that the first $|L| - x$ vertices are already considered, and we are only interested in the next x vertices (initial run is $f(L, R, |L|)$). The vertices in L is always sorted by the index.

Let's assume that x is even for simplicity. We will first try to find a matching of first $\frac{x}{2}$ vertices (excluding the ones already considered). Let's take the first $|L| - x + \frac{x}{2}$ of L, and call this set A. Let's find A_+, A_-, R_+, R_-.

Note that, we can independently find a best matching of $A_+ \cup R_+$ and $A_- \cup R_-$ (see Lemma 6). So we can run $f(A_+, R_+, t)$, t is equal to size of the intersection of A_+ and the last x vertices of L. Here we can note that t is $\le \frac{x}{2}$!

After this, we can delete A_+ and R_+ from the current graph.

Let's reuse L as $L - A_+$ and R as $R - R_+ = R_-$. Here, let's find L_+, L_-, R_-, R_+ again. Then, let's run $f(L_+, R_+, t)$, t is equal to size of the intersection of L_+ and the last $\frac{x}{2}$ vertices of L. Here we again can note that t is $\le \frac{x}{2}$!

Finally, we can add **any** max matching on $L_- \cup R_-$ to the answer, by Lemma 6.

Time complexity is equal to the $\mathcal{O}(\log \cdot (M(G) + D(G))$ (because there are log layers, as x each time is divided by two, and all vertices on all layers are non-intersecting), where $M(G)$ is the time complexity of maximum matching, and $D(G)$ is the time complexity of finding L_+, L_-, R_+, R_-.

Fast M(G)

We can find $M(G)$ very fast using a simple greedy algorithm!

Sort all (left part and right part) points by x, maintain all possible y of points from the left part in the set, and when you meet some point from the right part, take the closest by y point that you can delete (using lower bounds in the set).

Time complexity is $\mathcal{O}(n \log n)$.

Fast D(G)

We can optimize BFS using simple data structures!

For example, in the segment tree, for points of the left part sorted by x, you can store their y.

And then using it you can easily find all unvisited vertices, which can be reachable from the current vertex, and visit them!

Time complexity is $\mathcal{O}(n \log n)$.

Total

The total time complexity is $\mathcal{O}(n \log^2 n)$.

Problem J. Joining Points

Let's fix the chosen arc for the first color. After that, for each color, at least one arc won't be available, and the problem on the circle will be transformed to the problem on the line.

For each color, you will have either one possible arc $a \to b$, or two possible $a \to b \to c$ ($a \to b$ or $b \to c$), or zero possible arcs (but then you can skip this chosen arc for the first color).

Note that $a \to b$ and $b \to c$ are intersecting, so if we will find the largest possible number of non-intersecting subsegment and the number of ways to choose them, then if the largest possible number is equal to n, we can add the number of ways to the answer.