Problem A. Everyone Loves Playing Games

When dealing with operation $\oplus$ and we want to know what result may be, we usually consider the problem digit by digit in the binary notation and calculate the linear basis.

By observing, we can see that this is not a game theory problem in fact. The process is equivalent to that Acesrc makes his choice in his own linear space, and Roundgod make his choice after Acesrc finishing. Due to the transparency of information, Acesrc knows what Roundgod's best move is when he makes his decision on each digit.

Firstly, let's transform this problem into a problem of a single person instead of two to make best use of the linear basis. We have to choose A or B for each digit in this problem. To deal with it, let's pretend choosing A and add $A \oplus B$ as a new vector into the linear basis, which make it becomes a problem of choosing or not choosing.

Let's consider bits from high to low. There are three boolean conditions in each digit.

1. Whether this digit is 0 or 1.
2. Whether Acesrc can deal with this digit.
3. Whether Roundgod can deal with this digit.

To put it into a simplified way, let's encode these three conditions into a Boolean triple $(\cdot, \cdot, \cdot)$. Now we can see that one can do nothing to $(0, 0, 0)$ and $(1, 0, 0)$; Also, $(0, 0, 1)$ and $(1, 1, 0)$ is meaningless because doing operations simply make the result worse. In $(1, 0, 1)$, Roundgod will choose to do $\oplus$. Similarly, Acesrc will do $\oplus$ when it is $(0, 1, 0)$.

So all we have to do is dealing with $(0, 1, 1)$ and $(1, 1, 1)$, which is quite easy since Acesrc can choose the optimal for himself and keep track of the linear basis. Acesrc can predict what Roundgod will do and its influence on the linear basis. For example, Roundgod will definitely choose to turn 1 into 0 if possible, ignoring its future influence.

Problem B. Gifted Composer

Considering the time interval that there exists a repetition of length x, it must start at time x and end in some moment and won't show up again.

The basic idea is to find the end time for each length x with a binary search.

We only have to judge whether $S[1, n - x] = S[X + 1, n]$ if we want to check whether $S[1, n]$ has a repetition of x. We can do hashing to the final string and then answer each check in o(1).

So the total time complexity is $O(nlogn)$.

Problem C. An Unsure Catch

Obviously, the answer won't change after all of them are on a cycle.

Let's consider every modify operations, they will form a connected component. Now we need to discuss whether the cycle in the component is natural or man-made.

By observing, we can see that those man-made cycle must be self-loops and we don't care their exact position as long as it is coming from operations of decomposing cycles.

Now let's consider the natural cycle on a graph of n nodes and n edges (we usually think of it as a tree and a cycle). We can see that the change of the contribution when decomposing the next cycle edge is putting the original cycle head to the cycle tail with those tree nodes connected with it.

So we can do each modify operation in O(size of the subtree) and calculate the optimal contribution to the natural cycle in O(number of nodes in the component).

We can do greedy on how to choose the component by a sort. Note that this sort process can be a radix sort. Therefore, the time complexity for calculating concerning each natural cycle is $O(n)$.

We don't really care which natural cycle is optimal but only the length of it, there are at most $O(n^{1.5})$ kinds.

So basically, let's enumerate all possible lengths and calculate the contribution in $O(n)$. Take a cycle of largest contribution away and do greedy in the left part.

Problem D. String Theory

The problem asks the number of index pairs (l, r), such that $S[l..r] = A^k$ for some nonempty string A.

We enumerate the length of A. For length x, we may find out all maximal substrings $S_x(1), S_x(2), \cdots$ in S such that x is a period of these strings. For example, if 'S = abcabcabdabd', then there are two such substrings for $x = 3$: 'abcabcab' and 'abdabd'. Every substring of $S_x(i)$ of length kx is splendid.

Also note that the overlapping part of every two such maximal substrings is shorter than x. We may divide the original string into segments of length x. For every two adjacent segments, if they are identical, then they belong to the same maximal substrings; otherwise, they belong to different maximal substrings. In the latter case, one shall further compute their longest common prefix and suffix to decide how long the substrings can extend, both forward and backward. One may use suffix array with sparse table to achieve overall $O(n \log n)$ time complexity.

Note that the case $k = 1$ may require special treatment.

Problem E. Road Construction

The idea of this problem comes from JOI2011-2012, Day2, Constellation.

We have to find two spanning trees for two distinct sets on the plane and there is no intersection.

Let's give a conclusion first. We build a convex hull for all points from two sets. On the convex hull, for each set, there should be 0 or 1 continuous part, or there will be no solution otherwise.

It is easy to see that without this condition there is no solution, now let's give a construct way to prove this condition is enough.

Let's define a subprogram `WORK(A,B,C)` that point A is from one set X and point B, C (B, C are connected before by a direct or indirect way) are from the other set Y, aiming at connecting all points from X in the triangle ABC to A and all points from Y to B, C.

We can define such `WORK(A,B,C)` in this way:

1. If there is some points from X, let's pick a point S from it and connect A with S and then we run `WORK(S,B,C)`, `WORK(B,A,S)`, `WORK(C,A,S)` recursively.
2. If there is no point from X, which means all points from triangle ABC is from Y, we should find a proper way to connect all of them to B, C. One easy way is that let's sort them by the decreasing order of the distance to BC and then connect them one by one.

Now, if points on the whole convex hull are from one set X, then let's connect them one by one first. Then we can find a point S from the other set Y. At last, we run `WORK(S,ConvexHull(i),ConvexHull(i+1))` on the convex hull.

The other situation is there is one part from X and another part from Y on the convex hull. As usual, let's connect them one by one separately first. Then we find a supporting line between the first point A from X and the first point from Y in the clockwise order to divide it to two convex polygons. The rest part is to build a triangulation base on A and B on these two polygons and then run `WORK` for each triangle you get.

Bonus: It is obvious that we can reach a $O(n \log n)$ algorithm on something such as the random data. And we can also use the sweepline technique to improve our Special Judge to $O(n \log n)$. However, building a

point set with no three points are on the same line is quite annoying. So any easy way to construct this problem including Special Judge and Testdatas to a higher level?

Problem F. Girlfriend

The problem asks the following question: given an edge-weighted undirected graph. For each query, you are given two vertices u, v, among all simple paths between u, v, you should find the minimum possible weight of the second heaviest edge.

Note that to find the heaviest not the second heaviest edge, a minimum spanning tree simply works. Here, we may do something similar: for each query u, v, we add the edge from the lightest one to the heaviest one and merge the connected components, until u, v are adjacent in the original graph. The weight of the last added edge is the answer.

Now we solve this problem algorithmically. The main technique we use is the so-called *small-to-large trick*. Note that, we have a set of edges $E = \{u_i v_i\}_{i=1}^{m}$ and a set of queries $Q = \{u_i v_i\}_{i=1}^{q}$. Define $A_u = \{v_i : uv_i \in E\}$, and $Q_u = \{v_i : uv_i \in Q\}$. Also we define the weight of vertex u as $w_u = |A_u| + |Q_u|$. What we should do is to perform a series of merge operations like Kruskal's MST. To merge vertex u and v ($w_u \le w_v$), we always insert all elements in A_u and Q_u into A_v and Q_v, respectively. Merging two vertices may yield answers to some queries. How do we detect these queries? For each $x \in A_u$ if $x \in Q_v$, then we obtain the answer to query (x, v), and for each $x \in Q_u$ if $x \in A_v$, then we obtain the answer to (u, x). Due to the small-to-large property every element is visited at most $O(\log n)$ times.

The last thing to do is to map all these abstract data structures to concrete ones. If we use BBST we achieve $O(n \log^2 n)$ time and if we use hash table we may achieve $O(n \log n)$. When implementing, you should pay special attention to the indices in your data structures: keep in mind whether they are the original indices of the vertices or the indices of the representatives of union-find sets.

Problem G. Blackjack

The optimal strategy for Calabash is keeping drawing cards until the total points is greater than a.

Let's enumerate the last card he draws and suppose it is x. If he wins, which means the sum of points is larger than a but no larger than b, then the sum of points should be in the interval $(a - x, min(a, b - x))$ before he draws the last one.

We define $dp[i][j]$ as the possibility of drawing i cards and the sum is j without considering x we enumerate:

$$f[k][i][j] = f[k-1][i][j] + f[k-1][i-1][j-x_k] * i/(n-i+1)$$

$$dp[i][j] = f[n-1][i][j]$$

We know that we draw i cards before and the sum should be of $(a-x, \min(a, b-x))$, then the contribution to the answer should be $dp[i][j]/(n-i)$.

After enumerating a card, the time complexity for computing contributions to answer is $O(n^2)$. But the dynamic programming process is $O(n^3)$, which makes the time complexity become $O(n^4)$.

However, we can solve it easily by applying the way when solving the reversible knapsack problem. we calculate $f[i][j]$ as the possibility of drawing i cards and the sum is j without considering all cards. Then we can eliminate the contribution of any card in $O(n^2)$ and get the dp value in $O(n^2)$.

So the time complexity becomes $O(n^3)$.

Problem H. Yet Another Geometry Problem

Basically, we must see that a solution of $O(nq)$ can pass this problem.

Let's sort them by x-coordinate and enumerate the left endpoint as the left border. Then we update the answer with the right endpoint as the right border. Then update the top and the bottom. Since we need to enumerate left and right segments, we are solving a query with $O(n^2)$.

Now let's try to improve it to $O(n)$. We still enumerate the left segment, for the right part, we can only reach the maximum between two neighboring segments.

Considering the monotonicity of the top and bottom endpoints, when we enumerate the left segment from right to left, the useful length of all right segments won't increase while the distance to the left segment may increase, which means the decision point won't move to right. With maintaining the position of the decision point, we can answer each query in $O(n)$ now.

Problem I. A Math Problem

This problem can be solved in three ways.

First of all, we can actually solve it by Math. We know that topology is a structure on the set. Base on the definition, we can transform the problem into calculating the number of topology on the set of n elements. I don't know much about Math so I can't elaborate it further. Please do case study to $m = 2, 3, 4, 5, 6$. Please refer to some mathematical books to learn more.

Secondly, getting enough elements by brute-force, we can use the Berlekemp Messy Algorithm to solve the K_{th} element of the linear recurrence.

Now let's provide a more beautiful way to solve it. Consider fans of each team as a set, we need to calculate the number of ways when satisfying the given conditions. By simple deducting, we can prove the following properties:

1.There must be a set equal to the combination of all sets.

2.There must be a set equal to the intersection of all sets.

3.There is no equal set.

4.The biggest set must contains all elements.

5.The smallest set must contain no element.

Without considering the full and the empty set, there are at most 4 kinds of set, which means there are at most $2^4 = 16$ types of fans. Let's enumerate whether these types are existed and check the legality. Now for each case, we transform the problem into the number of ways of delegating n items to k colors, which can be solved by combinatoric number.

Problem J. Gaokao

By observation ,we can see the answer is 2^p, p is the number of 1 in $n-1$'s binary notation. This can be proved by Lucas Theorem.

Problem K. Data Structure

This problem can be solved in $O(n^{1.5})$, though some $O(n^{1.5}\sqrt{\log n})$ or $O(n^{1.5}\log n)$ solutions might also be acceptable. We only describe an $O(n^{1.5})$ solution here.

To achieve $O(n^{1.5})$ time complexity, we break operation 1 into two cases: $x > S$ or $x \leq S$, where $S = \Theta(\sqrt{n})$.

1. When $x > S$, the problem can be done by performing additions on n/x intervals, if we relabel the vertices in BFS order, such that the indices for vertices of the same depth are contiguous. Note that the time budget for each operation if $O(\sqrt{n})$, so we need a data structure (which is left as an exercise) that supports range addition in $O(1)$ time and querying the value of an element in $O(\sqrt{n})$ time.

But, how to find these n/x intervals? We may build a forest F according to the following rule: for each vertex u, connect it up to such a vertex v, if exists:

- v is one level deeper than u;

- the DFS order of v is minimum, but greater than u;

Note that the leaves in the original tree are the roots in F.

The leftmost k level descendants of u is simply the k level ancestor of u in F. We can use heavy-light decomposition to find the $O(\sqrt{n})$ k level ancestors of a vertex, because the additional $O(\log n)$ cost is dominated by $O(\sqrt{n})$. The rightmost k level descendant can be found likewise.

2. When $x \leq S$, for each modulus x and remainder y, store the vertices whose depth is y modulo x in DFS order. Hence each operation can be done by a single range addition. When querying the weight of a vertex, because there might be $O(\sqrt{n})$ moduli, we need to perform single element queries in the $O(\sqrt{n})$ sequences, each taking constant time. The design of such data structure is again left as an exercise.

Exercise:

1. Design a data structure maintaining an array of elements that supports $O(1)$ time range addition and $O(\sqrt{n})$ time single element query.

2. Design a data structure maintaining an array of elements that supports $O(\sqrt{n})$ time range addition and $O(1)$ time single element query.

Problem L. Landlord

Extend the edges into infinite straight lines and remove duplicated ones. The plane is thus divided into at most 5×5 connected regions. For two adjacent regions, if there was originally no edge between them, then unite these two components (you may use union-find). The answer is the number of sets in union-find.

Problem M. Travel Dream

Let's solve it with some randomized solution. First of all, let's color all nodes into 0 or 1 randomly. Supposing the largest cycle is $(v_1, v_2, ..., v_k)$, then the possibility that there are exactly $k/2$ continuous nodes are 0 and the rest $k - k/2$ nodes are 1 is $k/2^k$.

Now for components of 0 and 1, we need to calculate the longest simple path distance between each node pair (x, y) with the length of $k/2 - 1$ and $k - k/2 - 1$ saperately, stored by $f[0][u][v], f[1][u][v]$. We have $k/2 - 1, k - k/2 - 1 \leq 4$ obviously.

Then we enumerate two edges $e_1(u_1, v_1)$ and $e_2(u_2, v_2)$ crossing these two components. We now can update the answer with $f[0][u_1][u_2] + e_1 + e_2 + f_1[v_1][v_2]$.

After each random, we have the possibility of $k/2^k$ to reach the best answer. If we run it for T times, we have the possibility of $(1 - k/2^k)^T$ without having the best answer.

When $k = 10$, we set T as 1000, $(1 - k/2^k)^T$ reach 5×10^{-5}, which is small enough to ignore.

Then we need to do case study to find the longest simple path between any two nodes with the length of $1, 2, 3, 4$:

- Length=1: Enumerate edges and update the answer directly.
- Length=2: Enumerate two edges (u_1, v_1) and (u_2, v_2) and test if they share a common point. If so, update the corresponding answer.
- Length=3: Enumerate two edge (u_1, v_1) and (u_2, v_2) and test if they share a common point. If not, check if there is an edge between (u_1, u_2), (v_1, v_2), (u_1, v_2) and (u_2, v_1) and update the corresponding answer (the rest 2 nodes in u_1, v_1, u_2, v_2).
- Length=4: With the method in length=2, we calculate the longest 3 simple path with length of 2 and number those passed nodes. Then we eumerate two edge (u_1, v_1) and (u_2, v_2) and judge if they share a common point. If not, we get the longest simple path with length of 2 without passing replicated

nodes between (u_1, u_2), (v_1, v_2), (u_1, v_2) and (u_2, v_1) and update the corresponding answer (the rest 2 nodes in u_1, v_1, u_2, v_2).

During the testing round, we've hack a bunch of random, brute-force or search algorithms which trying to pass the problem. If anyone still use some tricky and possible hacky way to pass it, please feel free to discuss with us.