Fuzzy Pairs

Abutalib Namazov

December 2021

1 13 points

One can simply check for all possible sets of two numbers and see if they are in the same pair or not. This will give 5 points, but if you skip already found pairs, number of queries will not exceed 2^{14} which scores 13 points.

2 100 points

Let's create a sequence containing at most one number from each pair. We will call it *uni*. First we add 1 to *uni*, then from 2 to $2N$ when we add each number we will check if this number is from a new pair or already present in *uni*. If it is from a new pair, it will just stay in the sequence, otherwise we can binary search and find its couple. This way we can get 63 points. The solution can further be improved to 96 points by deleting the found pair from *uni* and to 100 points doing the above in random order. That is we need to random shuffle the numbers from 1 to $2N$ at the beginning.

LOTFIZADEH OLYMPIAD

Problem Analysis

Fuzzy Graph

Rashad Mammadov

December 2021

In the problem you are asked to construct a simple graph (an undirected graph without any self-loops or multiple-edges) with N nodes in which for all edges (u, v), the property $\gcd(degree[u], degree[v]) = 1$ holds.

1 Star graph

A simple construction would be a star graph. For example connecting nodes from 2 to N to node 1. This scores around 3.8 points.

2 Complete bipartite graph

It is not difficult to see that a complete bipartite graph can be constructed that holds the given conditions. For example, when N is odd ($N = 2K + 1$) you can put K nodes in one set and other $K + 1$ nodes in another set such that there is an edge between any two nodes from different sets. When N is even you can just eliminate one node and apply the same. This solution scores around 18.5 points.

3 Complete k-partite graph

From here on, one can observe the general idea of constructing a complete k-partite graph. See multipartite graph.

Now problem boils down to a partitioning problem, that is to find $n_1, n_2, \cdots, n_k$ such that the conditions below hold:

- $N = n_1 + n_2 + \cdots + n_k$, $n_i > 0$ for all i
- $\gcd(N - n_i, N - n_j) = 1$ for all $i \neq j$

Note that, $(N - n_i)$ is the degree of nodes in the i^{th} set. Also note that, the number of edges in this graph would be $\frac{N^2 - n_1^2 - n_2^2 - \cdots - n_k^2}{2}$. So, we are for maximizing this number or minimizing $n_1^2 + n_2^2 + \cdots + n_k^2$.

Despite the fact that N has exponentially many partitionings, one can find a good partitioning quite fast for $N \leq 1000$. We can recursively consider partitionings taking into consideration the 2^{nd} condition above and also the fact that we are for minimizing $n_1^2 + n_2^2 + \cdots + n_k^2$. This way we can eliminate many of them. This idea can score around 98 points, if efficiently implemented.

There is also another implementation of this idea with knapsack-dp which scores around 85 points.

In order to get 100 points, one needs to find the best partitioning. Indeed, jury's solution is based on that. This can be achieved by finding the best partitionings for the given values of N locally in your computer (which may take around 10-20 minutes) and then use those pre-calculated partitionings to construct the graph.

LOTFIZADEH OLYMPIAD

Problem Analysis

Fuzzy Uniqueness

Rahim Mammadli

December 2021

1 Subtask 1

For this subtask it is enough to go through each $(i, j), 1 \leq i \leq j \leq n$ and count the number of unique elements for contiguous subarray starting at i and ending at j. One can use map/unordered_map in $C++$ to solve this subtask. Expected time complexity : $O(n^3 * log(n))$ or $O(n^3)$

2 Subtask 2

Let us fix the starting point i, and find the uniqueness for each ending point j. As we sweep for j from left to right, if we come across a number we haven't seen before we add 1 to our uniqueness count. If we come across a number we have seen exactly once before, then we subtract 1 from our uniqueness count. While doing all this, we keep track of contiguous subarrays whose uniqueness is between l and r. Expected time complexity : $O(n^2 * log(n))$ or $O(n^2)$

3 Subtask 3

There is only $d = 500$ distinct elements, and we only need to count subarrays whose uniqueness is exactly 1. Let us fix the starting point i. Then for at most 500 numbers we will have ranges (x_i, y_i), where x_i denotes the index of first occurrence of number i that is greater than or equal to i, and y_i denotes the index of second occurrence of number i that is greater than or equal to i. Put $+1$ at each x_i, and -1 at each y_i. For given i, we are only interested in number of points j whose prefix sum , based on the given $+1$ and -1s, is exactly 1. This can be counted by a simple sweep in $O(d)$. The $+1$ and -1s can be kept in a *map* and updated at each index accordingly. This map will cost $O(n * log(n))$ in total. Expected time complexity : $O(n * d)$.

4 Subtask 4

Once again, we will sweep from left to right for starting indices i, and then count number of ending indices j for which the uniqueness count is 0 (as per given subtask). Let us denote the number at index m as a_m. Let us denote pv as the last occurrence of a_m to the left of m, that is $pv = \{max(l)$, such that $l < m$, and $a_l == a_m\}$. Similarly let us denote nx as the first occurrence of a_m to the right of m, that is $nx = \{min(l)$, such that $l > m$, and $a_l == a_m\}$. This number will result in adding $+1$ to some contiguous subarrays of the given array. This subarrays can be generalized as : subarrays whose starting point i is in range $[pv + 1, m]$, and ending point j is in range $[m, nx - 1]$. Let these ranges represent our updates. As sweep from left to right for starting indices i, when we reach $pv + 1$ we will add $+1$ to each ending index j, such that $m \leq j \leq nx - 1$, and when we reach $m + 1$ we will add -1 to each ending index j, such that $m \leq j \leq nx - 1$. While doing all of these we need to keep track of number of ending indices whose uniqueness count is 0. This can be done with range addition, range min lazy segment tree. Have an extra variable to keep track of number of indices equal to range min. Note that Subtask 3, can be solved in the same way, with keeping a few extra variables inside each node of the segment tree. Expected time complexity : $O(n * log(n))$.

5 Subtask 5

We will sweep from left to right for starting indices i, and then count number of ending indices j for which the uniqueness count is between l and r. We will use the aforementioned ranges and updates. So we have range addition by 1, subtraction by 1, and we want to keep track of ending indices whose uniqueness count is in between l and r. Divide the candidate ending indices into $sqrt(n)$ blocks each of size $sqrt(n)$, at each block keep n values. Let us denote this array of size $sqrt(n) * n$ as occ. $occ[i][j]$ stand for the number of ending indices in block i whose uniqueness value is currently j, we also have another array $lazy$, where $lazy[i]$ denotes the lazy sum added to the whole block i. With these structures the whole question can be solved in $O(n * sqrt(n))$ time, and $O(n * sqrt(n))$ memory. For a given range update, if only a part of a block is within the given update, go through the indices one by one, and apply the query. While doing that, update the count of candidate ending indices whose uniqueness value is between l and r, accordingly. For whole blocks that are inside the given range, go through them, add to their $lazy$ count, and use occ and $lazy$ arrays to update the count of candidate ending indices whose uniqueness value is between l and r, accordingly

Christmas Tree

Rahim Mammadli

December 2021

1 Subtask 1

For this subtask it is enough to go through each subset S, and each candidate socket node c, and calculate the best possible (S, c) pair. Bitmasks can be used to iterate over all possible subsets. Expected complexity : $O(2^n * n^2)$

2 Subtask 2

For a given c and S , the given formula is equal to $|S| * n - \sum_{x \in S} dist(x, c) - max_{x \in S}(h_x)$, where $dist$ denotes shortest distance between the given 2 nodes in the tree. Given this formula it is clear that, if we denote node y as the node with maximum h (heat) value among the ones we have chosen, e.g. $max_{x \in S}(h_x) = h_y$ and $y \in S$, then it is optimal to also include all the nodes z whose heat value is less than or equal to the one of y's in our chosen subset S. Indeed each added node will add n to our expression, and subtract at most $n - 1$ from our expression. So one can sweep over nodes' heat values, and pick all the nodes whose heat value is less than or equal to current value. We also have to find best possible socket node c to choose along with given S. In this subtask, this can be done by first, just doing bfs from each newly added node as we sweep over the heat values, and then going through all nodes and choosing the one with minimum sum of distances found by bfs. Expected complexity : $O(n^2)$

3 Subtask 3

From this subtask onwards, the general idea will always be to sweep over heat values, and choosing all nodes less than current heat value. Only thing that will change is the way we find socket node c. For this subtask the given tree represents "a path graph"(`https://en.wikipedia.org/wiki/Path_graph`). Let us pick one of the two degree 1 nodes as the root, and renumber the nodes as their distance from the root. Let m_i denote the new number of node i. In this case, given a chosen subset of nodes S the best choice for socket node c is the node corresponding to median of the set that contains all m_x, such that $x \in S$. This node and sum of distances from this particular node, can be kept track of with a simple set in $C++$. This is very similar to finding running median. One would need to keep track of two variables to keep track of sum from given socket node : $var1 = \sum_{y \in S, m_y < m_x} m_y$, and $var2 = \sum_{y \in S, m_y > m_x} m_y$. Expected complexity : $O(n * log(n))$

4 Subtask 4

Let us again sweep for heat value. Let us call a node "activated", if it has heat value less than or equal to current heat value that is being considered (e.g. it belongs to current subset S). It can be proven that as we sweep for heat values, if the optimal socket node in previous iteration was c_1, newly added node (as we increased the heat value exactly one new node was added) is cur , the new optimal socket node c_2 is somewhere on the simple path from c_1 to cur in the given tree. Let us assume $cur != c1$. Then let x denote the first node on the path from c_1 to cur. Let us cut the tree into two parts along the edge (c_1, x). Now it is clear that as long as the part of the tree not containing c_1 has more activated nodes, compared to the part of the tree containing c_1, we should move the socket node towards cur. Since the distance between any two nodes in this tree is $O(log(n))$, we can move the socket node one by one. And to check number of activated nodes in a particular subtree we can just use the binary structure given to us. Whenever a node is activated add $+1$ to each one of its parents. Note that knowing the count of activated nodes in subtrees is enough to figure out the number of the nodes in either side of the tree whenever we consider a cut, where cut is defined as above. Expected complexity : $O(n * log(n))$

5 Subtask 5

Let us again sweep for heat value. Now given, c_1 and cur as defined above, instead of moving one by one for socket node, we will binary search for it implicitly. Let us create the binary jump table for the given tree, e.g. create a two dimensional array, where $lca[i][j]$ denotes the 2^jth parent of node i. First of, if the current number of nodes is even, it can be proven that the previous c_1 is still one of the optimal nodes to choose as a socket. Now if the number of nodes is odd, then we know the new c_2 is somewhere on path between c_1 and cur. Let x again denote the first node on the path from c_1 to cur. Let us make a cut along the edge $(c1, x)$. Here let us denote number of activated nodes in the part of the cut containing c_1 as k. if c_1 is non optimal , then the number of activated nodes in the part of the cut not containing c_1 is exactly 1 more than the number of activated nodes in the part of the cut containing c_1, e.g. it is $k + 1$. This can be simply proved by contradiction, e.g. if it would'nt be $k + 1$ that means c_1 was not optimal socket node for previous iteration. This information will ease our algorithm. . Given c_1 and cur, and the fact that number of activated nodes currently is odd, and c_1 is currently non-optimal, the new c_2 is one of ;

a) The ancestor of c_1 with maximum depth who has more than k activated nodes in its subtree.

b) The ancestor of cur with maximum depth who has more than k activated nodes in its subtree.

Compute both and take the one with maximum depth. One would need dfs timestamps, to calculate "tin", "tout"s, and use fenwick tree/segment tree on top of it. As mentioned before binary search can be done implicitly using the lca binary jump table. Expected complexity : $O(n * log^2(n))$