IOITC 2020 TST solutions

Day 1

1. Removing Leaves

Consider the reverse process. Here, we add nodes one by one. A vertex can be added only after its parent has been added. Let d_i denote the depth of node i. Let the i^{th} added node be x_i. We have to maximize $\sum_{i=1}^{n} x_i(n-i+1)$. Observe that there is an optimal solution in which, for any i:

1. If x_i is not a leaf, x_{i+1} is a child of x_i : Indeed, consider the first i, where this property doesn't hold true. Let $j > i+1$ be the smallest index where x_j is a child of $v = x_i$. Let H be the sum of depths of all nodes between $i+1$ and $j-1$ inclusive, and let k be the total number of nodes inbetween. If we shift x_j to the position $i+1$ and all the intermediate nodes one position to the right, the change in objective function is $(d_v+1)k - H$. If this value is greater than or equal to zero, we can shift x_j to position $i+1$ and continue. Else, say $H > (d_v+1)k$. Now, we can shift x_i to position $j-1$ and all intermediate nodes one position to the left. The change is $H - d_v k > 0$ and our solution wasn't optimal in the first place, a contradiction.
2. If x_i is a leaf, x_{i+1} is a sibling leaf of x_i or all its sibling leaves have already been added : The proof is similar to that of the first property.

Using property 1, we find a solution in which we add leaves. To add a leaf x, find the closest to root ancestor (say y) of x that hasn't been added yet, and add all nodes on the path from y to x in that order. So, in our DP state, we can maintain the set of leaves that have been added. For each set S of leaves, we will also maintain the bitmask of all nodes (including non-leaf nodes) that have been added, that is, the nodes that lie on the paths from leaves in S to the root. When we have the value of $\text{dp}(S)$, we will iterate over leaves x not in S, find the corresponding y value by traversing up from x and using the stored bitmask to check if a node has been added, and update the cost of $\text{dp}(S \cup \{x\})$ accordingly.

The complexity of the above algorithm would be $O(N2^{|L|})$, where L is the set of leaves. (note that in the process of iterating over x and traversing up to the root till you find an already added vertex, each node is visited at max once, so the complexity is $N2^{|L|}$, instead of $N^2 2^{|L|}$). If our tree is a rooted binary tree, $|L| \le \frac{N+1}{2}$, and this approach passes.

Note that we haven't used the property 2 yet. As we know that all sibling leaves will be removed one after the next, we can group all leaves with the same parent together. Let R be the set of nodes that are the parent of atleast one leaf node. In our DP state, instead of keeping the subset of leaves that have been added, we will keep the sibling groups (each group represented by their parent (in R)) that have been added. The complexity here would be $O(n2^{|R|})$.

Notice that $|R| \le \frac{N}{2}$. This is because $|R| \le |L|$ (as each node in R has atleast one leaf as a child) and $|R| + |L| \le N$. So, the complexity is $O(N2^{N/2})$.

2. Counting Intervals

Say two intervals $[L_1, R_1]$ and $[L_2, R_2]$ *cross* each other if exactly one of L_1 and R_1 lies in $[L_2, R_2]$. Call an interval $[L, R]$ *good* if no $[a_i, b_i]$ crosses $[L, R]$. We have to count the number of good intervals.

For each i, define f_i to be the smallest $j \geq i$ for which there exists a k with $b_k = j, a_k < i$. f_i can be computed by traversing from left to right and keeping the right ends of valid intervals in a multiset. Similarly, define g_i to be the largest $j \leq i$, for which there exists a k with $a_k = i, b_k > i$. g_i can be found in a similar fashion iterating in reverse.

Note that an interval $[i, j]$ is good if and only if $j < f_i$ and $i > g_j$. So, now we have to count for each i, the number of j in $[i, f_i)$ for which $g_j > i$. To do this, iterate on i from 1 to n, and keep a set of all $j \geq i$ with $g_j > i$ in a fenwick tree. When we reach i, we can just query the sum in $[i, f_i - 1]$.

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

3. Paint It Black

Define $C_{i,j}$ to be 0 for each (i, j) outside the given $N \times M$ matrix.

Let U be the set of rows i for which there exists a column j such that $C_{i,j} = 1$, $C_{i-1,j} = 0$. Note that there must be a rectangle whose upper side is row i, otherwise $C_{i,j} = 1$, $C_{i-1,j} = 0$ is not possible. Therefore, for a solution to exist, $|U| \leq K$.

Let D be the set of rows i for which there exists a column j such that $C_{i,j} = 1$, $C_{i+1,j} = 0$. There must be a rectangle whose lower side is row i. $|D| \leq K$

Let L be the set of columns j for which there exists a row i such that $C_{i,j} = 1$, $C_{i,j-1} = 0$. There must be a rectangle whose left side is column j. $|L| \leq K$

Let R be the set of columns j for which there exists a row i such that $C_{i,j} = 1$, $C_{i,j+1} = 0$. There must be a rectangle whose right side is column j. $|R| \leq K$

If there exists a solution, we can find a solution in which for every rectangle its upper side belongs to set U, lower side belongs to set D, left side belongs to set L and right side belongs to set R. If say there is a rectangle whose upper side (say row i) isn't in U, we can move its upper side up until it is in U. This is because for all $i \notin U$, for all j with $C_{i,j} = 1$, $C_{i-1,j}$ is also 1.

Precompute 2D sums in $O(NM)$ (that is, compute the sum of all cells to left and up of (i, j) for each (i, j)). Now, to check if a set of rectangles is valid, we can first check for each rectangle that each cell inside the rectangle has $C_{i,j} = 1$ using a 2D sum query, then find the area of union of all rectangles using line sweep in $O(K \log K)$ and check whether this area equals the total number of cells with $C_{i,j} = 1$.

The number of ways to choose the upper sides of the K rectangles is $\leq s_2(K, |U|)$, where $s_2(n, k)$ denotes the stirling number of second kind, that is, the number of ways to divide a set of size n into k non empty groups. If $|U| = K$, this equals $K! \leq 24$ (each row in U is assigned to one of the K rectangles).

In our case, this is maximized when $k = 4, |U| = 3$, when $s_2(K, |U|) = 36$. Thus, there are 36^4 possibilities for the set of rectangles.

So, the total number of operations is $\leq 36^4 \times K \log K$, which easily fits in the time limit.

Day 2

1. Buildings

Let $f(a,b) = \max(0, b-a)$ be the energy requirement function. Note that it satisfies the triangle inequality : $f(a,c) \le f(a,b) + f(b,c)$.

Instead of storing surplus energy for future needs, we can think about storing no surplus energy and going back in time to buy the required energy at the minimum possible cost.

So, if we are at a building i, about to jump to building j, and the minimum cost per unit of energy in buildings visited before or at i is x, we need to add $f(H_i, H_j) \times x$ to the cost.

As the cost of energy at building n doesn't matter, define it to be 0. There exists an optimal solution in which:

1. The costs are decreasing. Let i be the first building in the sequences of buildings, from which we jump to a building $j \neq n$ with a larger energy cost. Let k be the building we jump to, from building j. Then, jumping to building k directly from building i is not worse. This is because $P_i(f(H_i, H_j) + f(H_j, H_k)) \ge P_i f(H_i, H_k)$ using the triangle inequality.
2. After the first building, we never jump to a building with a smaller height. Assume we jump from a building $i \neq 1$ to a building j with $H_j \le H_i$. Let k be the building before i in the sequence of visited buildings. Then directly jumping from k to j is not worse. This is because $P_k f(H_k, H_i) + P_i f(H_i, H_j) \ge P_k f(H_k, H_j)$ as $H_j <= H_i$.

Define dp$[x]$ to be the minimum cost of reaching building n starting from building x if only jumping to higher buildings is allowed. Then the required answer is the minimum dp$[i]$ over all i with $H_i \le H_1$.

To compute the dp values, sort according to decreasing heights. Now, $\text{dp}[i] = \min\limits_{j: H_j > H_i} (\text{dp}[j] + P_i(H_j - H_i))$. This is now a standard convex hull problem. Note that you don't need complex data structures such as dynamic convex hull, as the slopes are monotonic.

Another solution is to notice that after sorting in decreasing order of height, it is optimal to jump to the nearest vertex to the left with a smaller energy cost. Then, it is a classic stack problem.

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

2. Almost Shortest Paths

Let $d(i)$ be the shortest path from vertex 1 to vertex i, $f(i)$ be the number of shortest paths, and $g(i)$ be the number of paths with length $d(i) + 1$.

First, note that any path of length $\le d(i) + 1$ has to be simple. This is because, if there was a cycle, its length must have been atleast 2, and hence the total length must have been $\ge d(i) + 2$. So, the required answer for vertex i is $f(i) + g(i)$.

Compute $d(i)$ for all i using BFS. Now, let $P(i)$ be the set of neighbors j of node i with $d(j) = d(i) - 1$. Similarly, let $Q(i)$ be the set of neighbors with $d(j) = d(i)$. We have following recurrences for f and g:

- $f(i) = \sum\limits_{j \in P(i)} f(j)$.
- $g(i) = \sum\limits_{j \in P(i)} g(j) + \sum\limits_{j \in Q(i)} f(j)$.

The total complexity is $O(n + m)$.

3. Pairing Trees

Claim : Given a set $P = \{p_1, p_2, \dots p_k\}$ of paths on a tree T such that any two paths have a common node. Then, there must be a vertex v in all the paths in P.

Proof : Root the tree at an arbitrary node r. Let l_i be the LCA (closest to root) node in the path p_i. Let $t = l_j$ be the deepest node among $l_1, l_2, \dots, l_k$. Then, each path must pass through t. Say $p_i = (u_i, v_i)$ doesn't pass through t. Then, neither of the paths (l_i, u_i) and (l_i, v_i) can intersect with p_j, a contradiction.

In this problem we are given a tree with $2n$ nodes and n paths. In in one operation, we can choose two paths and choose a new pairing from the 4 endpoints. We need to find the minimum number of operations after which any two paths intersect. From our claim above, we know that they pass through a common vertex (say v).

Note that, when we remove all edges adjacent to v, the endpoints of any path must be in different connected components. This means that no component has more than n nodes as then we can't completely pair the nodes inside this component with nodes outside it. So, v must be a centroid.

There are ≤ 2 centroids, so we can iterate over them. Now, we need to change the pairing so that each path passes through the centroid v in focus. Let $C_1, C_2, \dots, C_m$ be the components formed on removing all edges adjacent to v. Call a path *bad* if it has both endpoints in the same component and *good* otherwise. Let B be the total number of bad paths and b_i be the number of bad paths in component C_i. Clearly, we need atleast $\lceil B/2 \rceil$ operations, as in a single operation, we can make at most two paths good. There are two cases:

1. $2\max(b_i) \leq B$: First assume B is even. In this case, we can keep on choosing two bad paths one each from two of the components with the highest number of bad paths, and cross their endpoints to make them pass through v. The invariant $2\max(b_i) \leq \sum b_i$ is maintained, and this process terminates with no bad path remaining. If B is odd, we will be left with a single bad path in the end, which we can fix by performing an endpoint swap operation with a good path outside the component containing this bad path. This way uses $\lceil B/2 \rceil$ operations, which is clearly optimal.

2. $2\max(b_i) > B$: Here as well, we can choose the same strategy as in the previous case. But in the end we will be left with $h = 2\max(b_i) - B$ bad paths in a single component (say C_j) and all other paths good. Then we can fix these bad paths one by one by performing an endpoint swap operation with any good path with both endpoints outside C_j. Note that the number of good paths outside C_j would be at least $(2n - |C_j| - (|C_j| - 2h))/2 \geq h$. So, we can match the h bad paths in C_j to the good paths outside C_j. This is optimal because we can't have more than $\frac{B-h}{2}$ operations which decrease the number of bad paths by 2 in any solution. So, we can't get anything better than $\frac{B+h}{2}$, which is what we achieve here.

Day 3

1. The social Gathering

We will first ignore the m bad pairs and then subtract them in the end. Let f_k be the number of people i with $l_i \geq k$. Similarly, let g_k be the number of people with $r_i \geq k$, and h_k be the number of people with $l_i \geq k$ or $r_i \geq k$. The f and g values can be easily computed in $O(n)$.

Consider $1 \leq i < j \leq n$ with $j - i = k$. For p_i and p_j to be friends, we want $r_{p_i} \geq k$ or $l_{p_j} \geq k$. Using inclusion exclusion, the number of ways to choose p_i and p_j is $g_k n + n f_k - g_k f_k$. We also have to subtract the ways in which $p_i = p_j$, so we subtract h_k to get $n(f_k + g_k) - f_k g_k - h_k$. The number of ways to choose for the rest $n-2$ positions is $(n-2)!$ Also, for a fixed k, there are $n-k$ ways to choose i.

So, ignoring the m bad pairs, the answer would have been $(n-2)!\sum_{k=1}^{n-1}(n-k)(n(f_k+g_k) - f_k g_k - h_k)$.

Now, for each bad pair (i, j), we need to subtract the number of permutations in which i and j were friends. Let $k_1 = \max(r_i, l_j)$ and $k_2 = \max(l_i, r_j)$. If i was seated before j, they would have become friends in $(n-2)!\frac{(n-k_1)(n-k_1+1)}{2}$ of the permutations, as we want their positions to differ by $\leq k_1$. Similarly, if i was seated after j, they would have become friends in $(n-2)!\frac{(n-k_2)(n-k_2+1)}{2}$ of the permutations.

The total complexity would be $O(n + m)$

2. Beautiful Trees

If $k = 1$ and the given tree is a chain, we output $n - 1, 2^{n-2}$ (we must choose the 2 leaves, and for other nodes we have 2^{n-2} choices. In all other cases, we prove that the only optimal solution is to choose only all leaves.

First, say a leaf was not chosen. Clearly, adding it will only strictly increase the objective, as a leaf is not an intermediate node in any path. So, all leaves must be present.

Let T be the optimal set of chosen nodes. We have already proved that T will contain all the leaves. We'll now prove that there will be no non-leaf node in T. Assume a non leaf node $x \in T$. Let $S_1, S_2, \dots, S_m$ be the components obtained on removing node x. Let X_i be the set of nodes $y \in S_i \cap T$, such that no intermediate node on the path between x and y is in T. Then, the change in objective function, if we remove x from T would be :

$$\sum_{i<j}\sum_{u\in X_i}\sum_{v\in X_j}(d(u,x)+d(v,x))^k - \sum_{i=1}^{m}\sum_{u\in X_i} d(u,x)^k$$

Using $(d(u,x)+d(v,x))^k \geq d(u,x)^k + d(v,x)^k$ (with equality only in case of $k = 1$), the change is atleast equal to (in case of $k = 1$, equal to, and in case of $k > 1$, strictly greater than):

$$\sum_{i=1}^{m}\sum_{u\in X_i} d(u,x)^k\left(\Big(\sum_{j\neq i}|X_j|\Big) - 1\right)$$

Firstly, the above value is always non negative as $|X_j| \geq 1$ for all j and $m > 1$. Hence, if $k > 1$, the change must be greater than 0. So, we will assume $k = 1$. Now, first remove all non leaf nodes till only one non-leaf node is left. In the process we haven't reduced the objective as the change was non negative in each step. Now, again if we remove this only remaining non-leaf node, the change would be non negative. For the change to be 0, we want $\sum_{j \neq i} |X_j| = 1$ for each i. This is possible only when there are only two leaves, that is, when the tree is a chain.

So, now we know that the number of solutions is 1. What's left is to find the sum of k-th powers of pairwise distances between the leaves. To do this, root the tree at an arbitrary node non-leaf node r. Let $g(j, x)$ be the sum of j-th powers of distances between leaves in subtree of x to x. Let $f(j, x)$ be the sum of j-th powers of pairwise distances between leaves in subtree of node x. Also, let $H(x)$ be the set of children of node x. Then:

- $g(j, x) = \displaystyle\sum_{y \in H(x)} \sum_{i=0}^{j} \binom{j}{i} g(i, y)$. This is because, if the distance of a leaf l to y was t, now it has become $t + 1$. So, we have to add $(t+1)^j = \displaystyle\sum_{i=0}^{j} \binom{j}{i} t^i$.
- Now, we have to think about how to merge two subtrees. Let F_1, G_1 be the f, g functions for the first subtree and F_2, G_2 for the second. Then we can merge them as $F(j) = F_1(j) + F_2(j) + \displaystyle\sum_{i=0}^{j} \binom{j}{i} G_1(i) G_2(j-i)$. This is because if we consider two leaf nodes one in the first subtree with depth a from x and the other in the second subtree at depth b from x, we have to add $(a+b)^j = \sum \binom{j}{i} a^i b^{j-i}$
- We iterate on the children of node x one by one, and use the above merging procedure.

Each merge operation adds a complexity of $O(K^2)$. Since we merge once for each edge, the overall complexity would be $O(NK^2)$.

3. Hidden Vertex

Subtask 1:

Let the hidden node be h. For any node x, query($\{x\}$) gives $2d(x, h)$. So, $h = x$ iff query($\{x\}$) gives 0. So, we can just ask n queries here.

Subtask 2:

First find two endpoints(say a, b) of a diameter. It is well known that the farthest distance of a node x to some node in the tree equals $\max(d(x, a), d(x, b))$. So, the idea is to always include a, b in our query set so that the farthest distance remains fixed across queries. So, first query $\{a\}$ and $\{b\}$ to find distances from a and b. Let D be the maximum of these two distances. If any of these distances is 0, we have our answer. Else, initialize the set of potential hidden vertices as $S = \{1, 2, \ldots n\} \setminus \{a, b\}$. Now repeat while $|S| > 1$:

> Let Q be a set of $\lfloor |S|/2 \rfloor$ nodes from S. Query for $Q \cup \{a, b\}$. If the result is equal to D, replace S by Q, else replace S by $S \setminus Q$.

Clearly, in each iteration, if the size $|S| = m$, it becomes $\leq \lceil \frac{m}{2} \rceil$ after the query. So, the total number of queries is $\leq \lceil \log_2(n) \rceil + 2$.

Subtask 3:

Root the tree at an arbitrary vertex r. Do a depth first search to find the dfs start times. Now, query for $\{r\}$ to find the depth of the hidden node (say t). Let $S =$ the set of nodes at depth t, be the initial set of potential hidden vertices. Repeat while $|S| > 1$:

> Sort S according to start times. Let Q be the set of $\lfloor |S|/2 \rfloor$ nodes in S with the smallest start times. Let D be the distance between the leftmost and the rightmost node in Q(this is also the maximum distance between any two nodes in Q). Query for Q. If the result is $\leq D$, replace S by Q, else replace it by $S \setminus Q$.

If the hidden vertex was in Q, the smallest distance is 0, and the largest distance is $\leq D$. Otherwise, if the hidden vertex was in $S \setminus Q$, the smallest distance is > 0 and the largest distance is $\geq D$, making their sum $> D$. Clearly, this method uses $\leq \lceil \log_2(n) \rceil + 1$ queries.

Subtask 4:

Here, we try to modify the solution for subtask 3. Let $S = \{1, 2, \ldots, n\}$ be the set of potential hidden vertices. Repeat while $|S| > 1$:

> For some $u \in S$, let $f(u) = \max\{d(u, v) | v \in S\}$. Sort S according to the f values. Let Q be the first $\lfloor |S|/2 \rfloor$ vertices of S. Let D be the maximum possible value of $d(u, v)$ for some $u, v \in Q$. Query for Q. If the result is $\leq D$, replace S by Q, else replace S by $S \setminus Q$.

To prove correctness, we need to prove that if the hidden vertex was in Q, the result of the query must have been $\leq D$, else it would have been $> D$. If the hidden vertex was in Q, the shortest distance would have been 0 and the largest distance would have been $\leq D$ making their sum $\leq D$.

Now, let p, q be the nodes in Q with $d(p, q) = D$. It turns out that for any node $r \in S \setminus Q$, $\max(d(r, p), d(r, q)) \geq D$. This can be proved by considering the virtual tree formed on the nodes in S, and observing that Q is the set of $|S|/2$ nodes closest to the center (node / edge). So, if the hidden vertex was some $r \in S \setminus Q$, the largest distance would have been $\geq D$ and the smallest would have been ≥ 1,making their sum $> D$.

The total number of queries is clearly $\leq \lceil \log_2(n) \rceil$. In each iteration, the time complexity is $O(|S|^2)$. The overall time complexity therefore is $O(n^2 + (n/2)^2 + (n/4)^2 + \ldots) = O(n^2)$.