Division B Final Contest problems analysis

Gleb Evstropov
evstropovgleb@gmail.com

March 15, 2023

1 Tree Height

Construct the tree from the input data. Determine which node is the root, i.e. has $p(v) = -1$.

Run any graph traversal from the root node. You can use either DFS or BFS, because there is exactly path between any two nodes of a tree. If you run DFS, do not forget keep track of the length of the current path by adding one more function parameter. Or you can store this value in a dedicated array and compute before you call DFS.

Both solutions work in $O(n)$ time.

2 Ladder

Define $d(i)$ as the maximum value of a path that starts at the first step and ends at step i.

1. $d(1) = a_1$
2. $d(2) = a_1 + a_2$
3. $d(i) = \max(d(i-1), d(i-2)) + a_i$
4. $d(n)$ is the answer

The above solution works in $O(n)$ time.

3 Knight 3

Construct a graph with one node for each board cell. Add an edge from node u to node v if a knight can get from the cell corresponding to u to the cell corresponding to v in one move.

Let (r_1, c_1) be the cell for node u and (r_2, c_2) be the cell for node v. Let $x = |r_1 - r_2|$ and $y = |c_1 - c_2|$. A knight can move from (r_1, c_1) to (r_2, c_2) in one move if and only if $\max(x, y) = 2$ and $\min(x, y) = 1$.

Then run BFS to find the shortest path. Restore that path and print it.

4 Edges on Some Shortest Path

For each edge uv you need to identify whether it belongs to at least one shortest path from a to b.

1. Run BFS to find the shortest distance from node a to all other nodes of the graph.
2. Run BFS to find the shortest distance from node b to all other nodes of the graph.
3. Edge uv belongs to at least one shortest path from a to b if and only if $\rho(a, u) + 1 + \rho(v, b) = \rho(a, b)$ or $\rho(a, v) + 1 + \rho(u, b) = \rho(a, b)$.

This solution runs in $O(n + m)$ time.

5 Colored Graph

The tricky part is to come up with a proper way to encode the state. For each path P that starts at node 1 we care about its other end v and the color of the last edge uv.

Thus, the actual graph we will use to find the shortest path is the graph of states $d(v, c)$ — what is the minimum possible length of the path that starts at node 1, ends at node v, and has the last edge (one leading to v) of color c.

Each edge uv of color c of the original graph will generate four directed arcs in the new graph. The edge uv of color c will lead from states $(u, (c + 1) mod 3)$ and $(u, (c + 2) mod 3)$ to state (v, c), and vice versa from v to u.

There are three starting states $(1, 0)$, $(1, 1)$ and $(1, 2)$, all have distance 0. Run BFS to find the shortest path, push all the starting states into the queue in the beginning. The answer is the minimum of three values $d(n, 0)$, $d(n, 1)$ and $d(n, 2)$.

The complexity of this solution is $O(n + m)$.

6 Colored Graph But Cooler

Apply exactly the same transformation to the given graph as in the previous task. Then run Dijkstra instead of BFS.

The complexity of this solution is $O(Dijkstra(n, m))$. That is $O(m \log n)$ for the heap or set based implementation that I suggest you to use for this task.

7 Socks

Interpret pairs of socks given in the input as graph edges. One can easily show that every two socks u and v belonging to the same connected component of this graph should have the same color after the repainting.

What is the optimal way to change the colors now? For each connected component the problem can be solved independently. The minimum number of changes means the maximum number of nodes will keep the original color. Thus, for each connected component we should use the color that is the most frequent in this particular component.

To count the most frequent color in one component you need to make a list of all vertices in this component. Then do one of the following.

- Use hash map $cnt(color)$. Consider all the vertices one by one and increase $cnt(c(v))$ by 1. Then pick the maximum value in the hash map.
- You can use an array $cnt(color)$ instead of a hash map. Consider all the vertices one by one and increase $cnt(c(v))$ by 1. After each increment update the maximum value with the current $cnt(c(v))$. At the end go through all the vertices once again and set $cnt(c(v))$ to 0 for all the colors that appear in this component. Thus the whole array $cnt(x)$ will be set to zero values again. That is a more complex way to find out the largest color, but it will work several times (up to ten times, actually) faster, than the first method.

The solution is $O(n + m)$.

8 Magnetic Triangles

1. Replace each triangle i with a new vertex u_i, connected to three endpoints of this triangle.
2. Find articulation point in the graph.
3. Triangle i is a bridge if and only if vertex u_i is an articulation point.

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

9 Lazy Coordinator

Observation 1. Consider the pool of proposals at any moment of time t. All of these proposals have the same expected remaining waiting time.

Observation 2. Consider the pool of proposals at any moment of time t. The remaining waiting time for each of these proposals doesn't depend on the preceding events. It only depend on the upcoming events.

Observation 3. Consider the pool of proposals at any moment of time t. Its size is known and doesn't depend on which proposal were used and which remain in the pool.

Now we consider all the events in the order of decreasing t, so we already know all that is going to happen later when we consider some event.

- Keep only two variables, the expected remaining waiting time x for each proposal in the pool, and the size of the pool k.
- If we move y moments of time back, we increase x by y.
- If we consider an event where coordinator picks one proposal from the pool, we increase the size of the pool by k. Note that this is not a mistake, we should **increase** the pool size, and not decrease, as we move back in time. After that you should multiply x by $\frac{k-1}{k}$ (assume the just used proposal has waiting time equal to 0).
- If we consider an event where the new proposal appears, use the current value of x as the answer for this proposal. Then decrease k by 1.

The complexity is $O(n)$.

10 Vanya and Jackets

First lets consider $O(n^3)/O(n^2)$ dynamic programming $d(i, j)$ — is it possible to follow all the requirements during first i days, if on the i-th day Vasya used jacket j. This dynamic programming can be computed straightforward in $O(n^3)$ time or in $O(n^2)$ if we precompute "first *true* after current position" and "last *true* before this position" for every pair (i, j).

Now, we can make an observation, that for each level no more than three jackets do matter. Indeed:

- if there are no jackets suitable on this day, the answer is just "No";
- if there is exactly one jacket we just need to check whether it's possible not to wear it on the previous day;
- if there are exactly two jackets we check if the first one is necessary on the previous day and that makes the second one necessary on this day;
- if there are three or more jackets suitable for the current day, none of them is crucial (i.e. every one can be left unused).

So leave no more than three jackets for each of the days in $O(n \log n)$ time and solve dynamic programming in $O(n)$.

11 Super M

1. Ari should teleport to one of the attacked cities (it doesn't worth going to a city that is not attacked since then she should go to one of the attacked cities).

2. The nodes visited by Ari will determine a sub-tree T of the original tree, this tree is unique and is determined by all the paths from two attacked cities.

3. If Ari had to return to the city from where she started, then the total distance would be $2e$, where e is the number of edges of T, that is because she goes through each edge forward and backward.

4. If Ari does not have to return to the starting city (the root of T), then the total distance is $2e - L$, where L is the distance of the farthest node from the root.

5. In order to get a minimum total distance, Ari should chose one diameter of the tree, and teleport to one of its leaves.

The problem is now transformed in finding the diameter of a tree that contains the smallest index for one of its leaves. Note that all diameters pass through the center of the tree, so we can find all the farthest nodes from the center.

12 Apline Problem

Consider two $O(nm)$ solutions.

First solution is to compute $d(v, k)$ for each node v and path length x. $d(v, k)$ equals to the maximum possible attraction of a path that ends at v and consists of exactly k tracks.

Second solution is to compute $d(v, x)$ for each node v and path attraction x. $d(v, x)$ equals to the maximum possible length of a path that ends at v and has attraction of at least x.

Let $T = \max(n, h_s)$. The key observation is that each path either its length or its attraction doesn't exceed $\sqrt{T}$.

We can apply first solution for all values of $k \leq \sqrt{T}$, thus solving the task for all short paths. Then we can apply the second solution for all value of $x \leq \sqrt{T}$, thus solving the task for all paths with small value of attraction.

The overall time complexity is $O(m \cdot \sqrt{\max(n, h_s)})$.

13 The Sting

Split all bets in three groups by the outcome being bet on.

For each bet i we can think of the following transformation. Instead of gaining b_i in case the bet losses and losing a_i in case the bet wins, we can say Ostap gets b_i at the moment he accepts this bet and then pays $a_i + b_i$ if this bet wins.

Now we solve an independent task for each of the potential outcomes. We compute $d(r, x)$ — what is the maximum possible total b_i of all bets with the outcome r, if their total value of $a_i + b_i$ is no greater than x. In other words, what is the maximum possible total profit for Ostap, if he can risk of loosing no more than x. This dynamic programming is similar to a notable knapsack problem.

Now, for each value of x we consider Ostap's worst-case profit to be $d(D, x)+d(W, x)+d(L, x) - x$.

The complexity is $O(n^2C)$, where $a_i, b_i \leq C$.

14 FoodBerry

First, let's learn how to solve problems for a fixed set of orders. I.e. check whether we need to use Super Large Warehouse at all.

Create a bipartite graph. Left part consists of nodes representing orders. Right part consists of nodes representing warehouses. Each warehouse has two nodes, one for d deliveries by car and the other for $c - d$ deliveries by foot. An edge connects an order-node with a delivery-node if the order is within a range of this method of delivery from this warehouse.

The most convenient way to solve the task would be to use flows. Create an arc of capacity 1 from the source to each order node, an arc of capacity $c - d$ from each foot delivery node to the sink, and an arc of capacity c from each car delivery node to the sink.

We can use bipartite matching as well. However, creating d vertices for each warehouse would make the graph too large. Instead, we can start by creating only one vertex for car delivery of each warehouse and one vertex for foot delivery of each warehouse. Then, if at some moment of work of Kuhn's algorithm all nodes of some type of delivery for some warehouse are used, add a new vertex of this type to the graph unless the maximum allowed quantity (d or $c-d$) is already reached. That is a useful technique for some other problems. However, for this particular one using maximum flow algorithms would be a way more convenient solution.

How do we solve the original task now? Just add nodes to the left part of the graph (i.e. add orders) one by one, and keep finding maximum flow or maximum matching using the previous result. That means, do not run the search from scratch, just run one more DFS.

The overall time complexity is $O(nm \cdot \min(n, m))$. Or simply cubic.

15 XOR and Favourite Number

- Compute prefix XORs $p_i = a_1 \oplus a_2 \oplus \ldots \oplus a_i$.
- XOR of all values from l to r inclusive can be found as $p_{l-1} \oplus p_r$.

- For each query (i, j) one needs to find the total number of pairs (l, r) such that $i \le l \le r \le j$ and $p_{l-1} \oplus p_r$.
- Let we have some current segment borders (i, j) and values $cnt(x)$ equal to the number of positions t such that $i \le t \le j$ and $p_t = x$. Let us also have the answer to the problem's question for segment (i, j).
- If we move each of the borders one step to the left or to the right, we can easily recompute all the values. For example, if we move j one step to the right, the answer increases by $cnt(p_{j+1} \oplus k)$ and $cnt(p_{j+1})$ increases by 1.
- Now we have all the prerequisites to use standard Mo's technique for processing segment queries.

The time complexity of this solution is $O((n + m)\sqrt{n})$.

16 Second Shortest Path

In this problem, we need to find a value y equal to the minimum possible length of a path from s to t, which is greater than the shortest path from s to t.

One of the possible solutions goes as follows.

- Run Dijkstra's algorithm.
- For each node v keep track of two minimum values of $d(v)$ and $d'(v)$, instead of just one in the traditional algorithm.
- Keep two versions of node v in heap or set, one for the shortest distance (minimum value of $d(v)$) and one for the second shortest distance ($d'(v)$ — second minimum value of $d(v)$).
- Process both copies of v in the heap or set as usual. Extract v and try to update all its neighbours with value $d(v) + w(vu)$ or $d'(v) + w(vu)$ depending on which copy you process. Note that both updates should consider and try to update both minimum and second minimum values.

The complexity is $O(m \log n)$.