International Collegiate Programming Contest (ICPC)
2024 Asia Dhaka Regional Contest
Onsite Round - Editorial

Hosted By
Department of Computer Science and Engineering
Daffodil International University

Supported by

Problem A. Are the Nodes Reachable?

Problem Setter: Raihat Zaman Neloy
Tester: Rumman Mahmud, Pritom Kundu, Nafis Sadique
Cetegory: Ad-hoc
Total Solved:
First to Solve:

We can split the problem is two subproblem.

When $ans < 32$**.** How do we solve it? First for each vertex calculate the list of vertex reachable from it. Also calculate the list of vertex from which it is reachable. Both can be done in $O(\frac{max(n)\cdot m}{64})$ using bitsets. Now assume that vertex a is reachable from vertex b. In that case vertices $b-1$ or $b+1$ will be reachable by adding an edge of cost 1. We can find all the vertices from which vertex a is reachable with 1 cost by simply binary shifting the bitset by 1 in both direction and apply the bitwise OR operator on all three (no change, left shift, right shift). If we do this again then we find the vertices 2 distance away and so on. We keep doing this for distances upto 31. The complexity to do this is $O(\frac{n^2*32}{64})$.

Now, lets split the bitsets into 64 sized buckets. Technically we could've used an unsigned long long array to perform the bitset operations in the first place. For each vertex we will have 33 bitset buckets. 32 of them are for the shifted bitsets from upstream and one for the downstream. Then for the query, we check if the first upstream bucket of V with 31 shift and the downstream bucket of U has any match. We can do that by simply performing a bitwise AND operation. If there is a match then we check 30 shift and keep reducing it. If no match is found then we move to the next bucket. At the end in this approach we will need $O(\frac{n}{64}+32)$ operations per query.

When $ans \geq 32$**.** We again keep a bucket of vertices reachable from a vertex. Vertices numbered from [1..64] are in one bucket, [65, 128] are in second buckets and so on. For each bucket we only keep the minimum id of the vertex reachable from it. Similarly we do it for the upstream vertices. For a query, we merge those two list of vertices (min and max of each bucket), sort them and find the smallest gap in consecutive vertices (one must be upstream while the other must be downstream). This takes $O(\frac{n*2}{64})$ per query.

Alternatively, we can do the same using 128 bit integers. The performance is comparable.

Problem B. Yet Another Crossover Episode

Problem Setter: Shahjalal Shohag
Tester: Rumman Mahmud, Pritom Kundu, Jubayer Rahman, Nafis Sadique
Cetegory: Dynamic Programming, Bit Manipulation
Total Solved:
First to Solve:

Problem C. Cut the Stick, Share You Must

Problem Setter: Rumman Mahmud
Tester: Shahjalal Shohag, Pritom Kundu, Nafis Sadique
Cetegory: Math, Number Theory
Total Solved:
First to Solve:

Problem D. CatGPT

Problem Setter: Anik Sarker
Tester: Rumman Mahmud, Pritom Kundu, Nafis Sadique
Cetegory: Graph, Offline Processing
Total Solved:
First to Solve:

There can be only 26 clowders. We can use this information. First, lets process all the event in the ascending order of R_i. We then process the cats from left to right. After processing each cat we keep in memory the most recent position we saw a cat of clowder C_i. Then we process the events ended at the current position. Because we know the last position for the clowder, we know that event will merge clowders that have their *last position* $> L_i$. We keep the list of merged clowders for the event in memory as well.

For the queries, we also process them in the ascending order of y_i. But this time we iterate over the events from left to right. Since we know for an event, the clowders that will be merged, we can generate a list of unordered (i, j) pairs that will be merged for an event. There can't be more than $\frac{25 \cdot 26}{2}$ pairs. We keep track of the last event they were merged, similar to the previous section. Finally for an event, we process the queries that ended in that event. From the list of the last position of the pairs, we can have a graph of 26 vertices. We merge vertices using union-find. Then we check the largest merged clowder and output their size.

Problem E: Quasi-binary Representations

Problem Setter: Pritom Kundu
Tester: Shahjalal Shohag, Pritom Kundu, Jubayer Rahman, Nafis Sadique
Cetegory: Dynamic Programming, Math
Total Solved:
First to Solve:

Problem F: Flowers

Problem Setter: Hasinur Rahman
Tester: Nafis Sadique
Cetegory: Hashing, Divide & Conquer
Total Solved:
First to Solve:

Problem G: Library Function vs Keyword

Problem Setter: Shahriar Manzur
Tester: Jubayer Rahman, Nafis Sadique, Rumman Mahmud
Cetegory: Ad-hoc
Total Solved:
First to Solve:

For $strlen(line)$, we simply need to find the position of the first null('\0') character. The number of characters before that is the answer. If no null characters exist then the answer would be the string length.

For $sizeof(line)$, cound the number of null in the string. Then the answer is $string\ length - number\ of\ null\ characters +$ 1.

Problem H: Hand Cricket

Problem Setter: Kazi Md Irshad
Tester: Pritom Kundu, Jubayer Rahman, Nafis Sadique

Cetegory: Data Structure, Math, Probability
Total Solved:
First to Solve:

Problem I: In Search of a Kind Person

Problem Setter: Md. Imran Bin Azad
Tester: Shahjalal Shohag, Rumman Mahmud, Raihat Zaman Neloy, Muhiminul Islam Osim
Cetegory: Ad-hoc
Total Solved:
First to Solve:

Problem J: The Taxman

Problem Setter: Aminul Haque
Tester: Rumman Mahmud, Muhiminul Islam Osim
Cetegory: Binary Search, Math
Total Solved:
First to Solve:

Problem K: Packet Transmission

Problem Setter: Ashraful Islam
Tester: Pritom Kundu, Nafis Sadique
Cetegory: Least Common Ancestor
Total Solved:
First to Solve:

Every query we receive can be classified in the following way.

- The query packets (source, destination pair) share a common path in the tree.
  - The packets are going in the same direction. In that case we need to assume one of the packets will never wait. So, the other packet will arrive at the first vertex of the common path, wait for the first packet to arrive and then go after it. Since an edge can't be used by multiple packets at the same time, the other packet will have to wait until the first packet crossed that edge. Keep doing that until they reach the end of common path and then they can go their own ways. If we think carefully, we can see that the other packet will have to wait an additional time totalling the maximum edge cost on the common path. However if the first packet comes early then the wait time reduces. Do the same the other way around and take the minimum of the two.
  - The packets are going in the different directions. So, we need both packets to start their journey. Eventually they might meet on the two sides of an edge where if one starts crossing, the other must wait. Make one of them wait and the other one go and calculate the time it would take for both them to reach their destinations. Do it both ways and take the minimum time. However if they don't meet at the edge then they don't have to stop and their actual time to reach both their destinations is the answer.
- The query packets use completely different paths. In that case the time it takes for both to reach their destination is the answer.

We can calculate all of these using Sparse Table to calculate Least Common Ancestors in a tree. Alternatively we can use heavy-light decomposition, but that may be very slow. There are lot of corner cases, so careful implementation is necessary.

Problem L: Unhappy Team

Problem Setter: Nafis Sadique
Tester: Shahjalal Shohag, Rumman Mahmud, Jubayer Rahman
Cetegory: Dynamic Programming
Total Solved:
First to Solve:

The trick is to select the unhappiness score of someone and count how many times that score appear in the top K values. We can keep a DP state ($bit\_mask, bigger\_scores\_count, selected\_score\_appeared$). Basically we will put people one after another, everytime calculating their unhappiness score. If the score is smaller than the selected score then we ignore it. If the score is bigger then $bigger\_scores\_count$ is incremented by one. If the score is the same is the selected score then we can consider it as bigger or mark $selected\_score\_appeared$ as $true$. This DP will have the complexity of $O(2^n \cdot n^3)$. With some other minor optimizations, this runs reasonably fast.