2022-2023 ICPC Latin American Regional Programming Contest — Unofficial editorial

This is the unofficial editorial for the 2022-2023 ICPC Latin American Regional Programming Contest. For any questions/suggestions, you can reach out to me personally through Codeforces or leave your comments in the blog of the editorial. Thank you!

A. Asking for Money

Let's fix a person p and call a and b to the people p will ask for money. It's easy to see that p loses money if a, b and p are asked for money before p requests a or b to pay.

This is equivalent to checking if, starting the process in any person, we can visit the three people without taking into account the edges that start in p. By fixing the starting vertex and performing a BFS/DFS, we can check if this happens. Time complexity to perform this process for all the possible values of p is $\mathcal{O}(n^3)$. This is too slow.

To improve our algorithm, we can consider reversing the edges of the graph. Now we need to check if there is a vertex that is reachable from p, a and b. This means we can run our BFS/DFS starting from a, b and p, and then check if there is a vertex that was visited by all of them. Given that for a fixed value of p we only transverse the graph 3 times, the final time complexity is $\mathcal{O}(n^2)$.

B. Board Game

Instead of iterating over the players, let's compute for each token, the minimum id of a player that can get it

Let's assume that for a given token, we want to check if there is any player with id in $[l, r]$ that can get it. If there is at least one valid player in the range, it's easy to see that the one maximizing $A_i \cdot X + B_i$ will be one of them.

Let's build a segment tree over the players. On each node, we store the lines of all the players with id contained in the node's range. Now, for a given token (X, Y), we can start descending the tree. Let's assume we are currently standing in one particular node of the tree, and let's call V to the maximum value of $A_i \cdot X + B_i$ for this node.

- If $Y \geq V$, then none of the players in the node's range can take this token. We can ignore all of them.
- If $Y < V$, we know there is at least one player in this range that will take it. Let's check using recursion if there is at least one in the left child of the node. If we couldn't find it, then move to the right child.

It's easy to see that in each level of the segment tree we visit at most 2 nodes, so we check $\mathcal{O}(\log n)$ nodes in total.

The only thing remaining is to be able to quickly check the maximum value in one node for a given token. To do that, we can use convex hull trick on each node to be able to answer queries of maximum value of $A_i \cdot X + B_i$ in $\mathcal{O}(\log n)$.

Final time complexity: $\mathcal{O}(n \cdot \log^2 n)$. This can also be improved to $\mathcal{O}(n \cdot \log n)$ by processing tokens in increasing order of X, but it was not necessary to get AC. Alternatively, you can also use Li Chao tree to solve the problem.

C. City Folding

Let's consider the process backwards (i.e. starting from the last fold).

If H is in the upper half of the paper, then in the previous step, it was in the half that was folded on top of the other one. Otherwise, it was on the other part. By using this observation we can compute, for each step, if we should fold or not the part in which H is.

Now let's start the process from the beginning from P. We have some cases:

- If in this moment P should be part of the half that will be folded, then if P is in the left half we push L to the answer, or R otherwise.
- On the contrary, if P shouldn't be on the folded half, we push R if P is in the left half, or "L" otherwise.

Don't forget that whenever we fold one part on top of the other, the order of its elements is reversed.

D. Daily Trips

This was the easiest problem in the contest. The solution is to simulate the process described in the statement. It can be done in several ways.

E. Empty Squares

This problem has many solutions.

It can be proven that the answer is at most 3. This allows us to solve it in $\mathcal{O}(1)$ by some annoying case analysis, or even in $\mathcal{O}(n^2)$ by brute force with some observations.

Let me introduce a different solution that doesn't require any case analysis. We will solve the problem by using dynamic programming. In particular, $f(i, a, b)$ denotes if we can cover two squares of sizes $1 \times a$ and $1 \times b$ respectively, if we only have tiles of sizes $1 \times 1, 1 \times 2, \ldots, 1 \times i$.

- If $a = 0$ and $b = 0$, then $f(i, a, b) = \mathit{true}$.

- We can skip the tile of size $1 \times i$, transitioning to $f(i-1, a, b)$.
- If $i \neq k$, we can try to use the tile of size $1 \times i$ to cover either the first square ($f(i-1, a-i, b)$) or the second one ($f(i-1, a, b-i)$). Be careful to check that $i \leq a$ and $i \leq b$ respectively.

The solution then is to iterate over the values of a and b such that a is at most the size of the first part, and b is at most the size of the second part. If $f(n, a, b) = \textit{true}$, then we can cover $k + a + b$ squares. By taking the maximum value we can obtain over all possible values of a and b (denote it by v), we can note that the answer is $n - v$.

The number of states in our dp is $\mathcal{O}(n^3)$. In order to save some memory, we can notice that $f(i, a, b)$ only have transitions to $f(i-1, x, y)$ for some values of x and y. We can keep the dp values only for the previous value of i. By doing this, we reduce memory usage to $\mathcal{O}(n^2)$.

This looks too slow. But we can notice that $a + b <= n$, so the total amount of states is actually at most $\frac{n^3}{4}$. This is fast enough for $n \leq 1000$.

F. Favorite Tree

Let's assume the root of T_2 is always vertex 1. We are going to iterate over the root R of T_1, and we will assume it is matched with the root of T_2.

Now we need to solve the problem for rooted trees, assuming that the root of both trees will always be included. In particular, let's denote as $f(i, j)$ as a boolean value that denotes if we can remove some nodes of the subtree of i in T_1, so that the remaining nodes form a tree that is isomorphic to the subtree of j in T_2. We want to find out the value of $f(R, 1)$.

We are going to denote the set of children of vertex i in T_1 as $ch_1(i)$. In the same way, we define $ch_2(i)$ to represent children of i in T_2. Let's see how to compute $f(i, j)$.

Let's create a bipartite graph, where the left part is $ch_1(i)$ and the right one is $ch_2(j)$. Given $x \in ch_1(i)$ and $y \in ch_2(j)$, we add an edge between x and y if $f(x, y) = \textit{true}$ (we check this value with recursion). Notice that we reduced the problem to finding a maximum matching. If we can match every

vertex of the right part, then $f(i, j) = \textit{true}$. To compute this, we can use Hopcroft-Karp's algorithm

Let's analyze the time complexity of our solution. Notice that if we fix two nodes v_1 and v_2 in T_1 and T_2 respectively, we will add one edge to the bipartite graph while computing $f(i, j)$ only when i is the parent of v_1 and j is the parent of v_2 (and this happens at most once for a fixed root). This means that the total edges in all the matchings required to compute $f(R, 1)$ is $\mathcal{O}(n^2)$.

The time complexity of running Hopcroft-Karp's algorithm is $\mathcal{O}(m \cdot \sqrt{n})$, where m is the amount of edges. In our case, $m = n^2$, so the cost of computing $f(R, 1)$ for a fixed value of R is $\mathcal{O}(n^2 \cdot \sqrt{n})$. Given that we iterate over all the possible values of R, we conclude that the final complexity of our solution is $\mathcal{O}(n^3 \cdot \sqrt{n})$, which clearly fits in the time limit.

G. Gravitational Wave Detector

Let's denote the two polygons as P and Q respectively. We will solve the case in which the middle point coincides with a minor plant. The other two cases can be handled in a really similar way.

Let's consider a particular minor plant c, the answer is Y if there exists two points $a \in P$ and $b \in Q$, such that:

$$\frac{a + b}{2} = c$$

We can rewrite this as:

$$(a + b) = 2 * c$$

Given that P and Q are convex, the points that can be represented as a sum of $a \in P$ and $b \in Q$ also form a convex polygon with $\mathcal{O}(n + m)$ vertices, and we can construct it in linear time. This is called the Minkowski Sum of P and Q.

So, after computing the Minkowski sum of P and Q, we have reduced the problem to be able to check if c lies inside a convex polygon. This can be done with binary search in $\mathcal{O}(\log n)$.

Final time complexity of the solution is $(n + q \cdot \log n)$.

H. Horse Race

Let's analyse the i-th small race:

- If the j-th horse took part of it, it means it's final place should be at least W_i.
- If the j-th horse didn't take part of it, it's final place can't be exactly W_i.

By considering these conditions, we can compute for each horse, which are the valid positions for it in the answer. After we do this, we can construct a bipartite graph where left part represents horses and right part represents positions. We add an edge between i and j if the i-th horse can be placed in the j-th position.

It's easy to see that we need to find a perfect matching in this graph. Given that $N \leq 300$, any of the well known matching algorithms will fit in the time limit.

I. Italian Calzone & Pasta Corner

Let's iterate over the first dish we take. We can maintain a set of the dishes that can be visited in the next step (i.e. neighbours of visited cells). From all of the candidates in the set, it's always better to take the smallest one. We repeat this procedure until we run out of candidates. The cost of doing this process for a starting dish is $\mathcal{O}(n^2 \cdot \log n)$.

Given that we iterate over $\mathcal{O}(n^2)$ starting cells, the total complexity is $\mathcal{O}(n^4 \cdot \log n)$. If the solution is coded without a big constant factor, you can make it fit in the time limit.

In order to optimize our approach, let's iterate the starting dish in increasing order of value. We can notice that if we visited a cell when we executed the process in a previous candidate, it's useless to start all over

again with this value as the starting position: during that previous step, we visited all the positions we would visit now, plus some others too (so, the answer was bigger).

Although this optimization improves the performance of the algorithm, it doesn't actually improves its time complexity. There are cases in which it will perform $\mathcal{O}(n^4 \cdot \log n)$ operations, but the constant factor is smaller (and for $n \leq 100$ it runs fast enough).

J. Joining a Marathon

Let's first think about how to detect which photos are trash.

We can rewrite the equation of the position of the i-th runner as $S_i * t - T_i \cdot S_i$. This is the equation of a line, which we will denote as f_i, and we will denote as $f_i(v)$ as the result of evaluating the i-th line in $t = v$.

For a given photo, if we sort all the runners by the value of $f_i(U)$, we can check if there exists an index j such that $A \leq f_j(U) \leq B$ by using binary search. There is a problem: we clearly can't sort all the lines for each photo. Instead, we will process the photos in increasing order of U, while keeping the sorted list of runners in the meantime.

Let's take two lines f_i and f_j such that $S_i > S_j$. We will denote the value of t such that $f_i(t) = f_j(t)$ as $x_{i,j}$. It's not hard to see that for every value $U \in (-\infty, x_{i,j}]$, the i-th runner will be before the j-th one in the sorted list of lines. If we consider that $U \in (x_{i,j}, \infty)$, then the relative order between them should be swapped.

Initially, we can sort the lines assuming that $U = -\infty$. We will maintain in a set the values of $x_{i,j}$ for the lines i and j that are located in consecutive positions in the current list. Now, let's start processing the photos by increasing value of U. When processing a new photo, we should swap consecutive lines such while $x_{i,j} \leq U$ (and update the corresponding values of the affected positions in the set). After we finish updating all the necessary positions, it's easy to see that now the lines are sorted by the value of $f_i(U)$, so we can apply the binary search we mentioned before.

We can notice that when two consecutive values i and j are swapped, we will never swap them again, so the amount of times we do a swap is $\mathcal{O}(n^2)$.

This means that the complexity of computing the photos that are trash is $\mathcal{O}((n^2 + m) \cdot \log n)$. Let's remove from the list all the photos that are not trash, since they won't affect any query.

Now let's process the queries offline. Instead of counting the amount of trash photos for each query, we will compute for which queries each photo is trash.

It turns out we can use the exact same algorithm to keep the sorted array of queries by the current value of U while iterating over the photos again. After updating the order of queries for a given photo, we need to add one to the value of all the queries such that $f_i(U) < A$ or $f_i(U) > B$. Given that the queries are ordered by $f_i(U)$, all the positions in which we add one are located in a prefix or a suffix of the list.

We need a data structure that allows us to perform two operations:

- Swap two consecutive positions.
- Add one to a range of positions.

There are several ways to do this. For example, you can use an implicit treap, a lazy segment tree or even a Fenwick tree.

The final complexity of the solution is $\mathcal{O}((n^2 + q^2 + m) \cdot \log n)$.

K. Kind Baker

This problem probably has many different solutions. I'm going to describe the approach I used.

First, let's notice that if we use the machine n times, we can have at most 2^n different different types of pieces. So we know that the value of n has to be at least $\lceil \log_2 k \rceil$. Let's show that we can actually achieve this.

First, if $k = 1$, just print 0. Otherwise, let's fill the first column and the first row with all the n colors. Now we have 2 different types, so we can subtract 2 from the value of k.

Given that $k \leq 4000$, notice that $n \leq 12$. Also, notice that $2^{\frac{n}{2}} \leq 100$. Let's denote $k_1 = \lfloor \frac{n}{2} \rfloor$ and $k_2 = \lceil \frac{n}{2} \rceil$.

For each value $i \in [0, k_1)$, iterate over all the rows from 2 to 100, and if the id of the current row has the i-th bit set, add the color i to the entire row. You can do the same with columns for the other k_2 values, but using columns instead of rows.

It's not hard to notice that by doing this, we generate exactly 2^n different types of pieces. Also, thanks to the fact that we used every color in each of the cells of the first row and column, every color forms a connected component. So this solution is correct.

The only remaining part is to make the amount of different types exactly equal to k. This can be done in several ways. For example, you can perform the described process only until you detect that you already have k types, or you can generate all the 2^n types and then clear some cells until the amount is reduced to exactly k.

Time complexity is $\mathcal{O}(k)$.

L. Lazy Printing

It can be proven that when performing an instruction, it's always optimal to print the maximal amount of letters possible. I will show how to compute this value. During the explanation, $\text{lcp}(i, j)$ is the length of the longest common prefix between the suffixes of T that start in i and j respectively.

Let's denote the next position to be printed by the machine as i. We will iterate over all possible lengths (between 1 and D) of the string that the machine can use now. For length k we can write $k + \text{lcp}(i, i + k)$ letters with this operation.

The value of $\text{lcp}(i, i + k)$ can be calculated in several ways in $\mathcal{O}(\log n)$ or $\mathcal{O}(1)$ (for example hashing or suffix array), so the time complexity is represented by the amount of times we compute $\text{lcp}(i, i + k)$ for some values of i and k.

At first glance, it looks like our algorithm's complexity is $\mathcal{O}(n \cdot D)$, which would be too slow, but it turns out this is not true. When we compute the maximum value for a starting position, we perform D calls to the function lcp, but the key fact is to realize that this value we get is at least D. This means that after doing D calls, the value of i will increase by at least D.

From this observation we can conclude that the final complexity is $\mathcal{O}(n)$ or $\mathcal{O}(n \cdot \log n)$, depending on how you compute lcp.

M. Maze in Bolt

This problem can be solved by performing a BFS/DFS. Let's define which are going to be our states.

Given that rotating the nut horizontally C times is the same as not rotating it at all, we can represent the current state with two values: the row of the screw where the nut is located, and how many times we rotated the nut horizontally (between 0 and $C - 1$). Notice that the amount of states is $R \cdot C$, and this is at most 10^5.

We can start the BFS/DFS from all valid positions in the first row at the same time, and check if we can reach any position in the last row by performing the operations described in the statement.

Checking if a state is valid can be done in several ways, but the easiest one is to iterate each position of the nut naively, because the constraints allow us to do that. Time complexity is $\mathcal{O}(R \cdot C^2)$.

Don't forget to also try to reverse the string S and try the BFS/DFS again (representing the flip operation).