International Collegiate Programming Contest

Latin American Regional Contests

March 18, 2023

Editorial

Problem A – Asking for Money

Author: Pablo Blanc, Argentina

For someone to lose money (say K), he must be reached by the scam and the people he would ask money from (X_K and Y_K) must be reached first. Since each person can take time to ask other people for money, the important thing is that X_K and Y_K are reached by someone other than K.

We consider the graph given by the inverted edges, that is, $X_i \to i$ and $Y_i \to i$ for every i. We fix K and remove $X_K \to K$ and $Y_K \to K$ (X_K and Y_K are not contacted by K), then we see which nodes can be reached starting at K, X_K, and Y_K. If there is any node that can be reached from the 3 nodes (K, X_K, and Y_K), then K can lose money. This holds since if the scam starts with that person, it could reach X_K and Y_K, and then K, then he would pay \$1 but he would get nothing by asking X_K and Y_K for money.

For each person, we go through the graph 3 times so the solution is $O(N^2)$.

Problem B – Board Game

Author: Enrique Junchaya, Perú

For every token, we can find which player took it with a binary search. The predicate of such a binary search is true for a number i if a player j with $j \leq i$ took the token. Hence, the answer is just the first index for which the predicate is true.

In order to test a predicate for an index i, we could do convex hull trick or keep a Li Chao Tree with the first i lines. Then, for a point (x_j, y_j), we just have to query the maximum of the lines in the abscissa x_j and compare it to y_j. Thereby, each predicate test costs $O(P \lg P)$ and doing the binary search for a point costs $O(P \lg^2 P)$.

Of course, repeating this for every point would TLE, but we do not need to do that. We can use the same line container structure to answer the predicates for all the points using parallel binary search. Overall, we just need to rebuild this data structure $O(\lg P)$ times. With this, the total insertions in the line container cost $O(P \lg^2 P)$. Also, answering all queries costs $O(T \lg^2 P)$ since we must do $O(T \lg P)$ of them. Therefore, the total time complexity of this solution is $O((P + T) \lg^2 P)$.

Another approach is to build a Persistent Li Chao Tree. Each line insertion creates a new *"snapshot"* of the tree. Then for each token, we can binary search in which snapshot a query on the tree results in a value higher than the y_j value for the given x_j. The complexity for this solution is $O(T \lg^2 P)$.

Problem C – City Folding

Author: Mário da Silva, Brasil

In any given stage, let S_H denote the stack's current height and A_H the height of Amelia's house. If Amelia is currently on the lower half ($A_H \leq \frac{S_H}{2}$), then she was at this same height in the previous stage (before the last folding). Otherwise, she was at height $S_H + 1 - A_H$. Knowing this, we can go backward from H and find Amelia's height at each stage. Then, starting from the beginning, at each step we can determine the folding direction by taking into account Amelia's current position (from left to right), current height, and future height.

Problem D – Daily Trips

Author: Mário da Silva, Brasil

Just simulate: keep track of the number of umbrellas at Bella's home and workplace.

```python
#!/usr/bin/env python3

(N, at_home, at_work) = map(int, input().split())

for _ in range(N):
    morning_rain, night_rain = input().split()
    bring_work, bring_home = 'N', 'N'
    if morning_rain == 'Y' or at_work == 0:
        bring_work = 'Y'
        at_home -= 1
        at_work += 1
    if night_rain == 'Y' or at_home == 0:
        bring_home = 'Y'
        at_work -= 1
        at_home += 1
    print(f'{bring_work} {bring_home}')
```

Problem E – Empty Squares

Author: Pablo Blanc, Argentina

The idea is to observe that when the holes are large we will be able to completely cover the board. Then the cases with small holes remain and can be solved by hand or by brute force.

Let $A = E$ be the size of the first hole and $B = N - K - E$ be the size of the second hole.

We begin with the case $A = B = K$. In this case if $K \geq 5$ we can cover the hole of size A with the tiles $1 \times (K-1)$ and 1×1, and the hole of size B with the tiles $1 \times (K-2)$ and 1×2, the answer is 0. It is easy to solve the remaining small cases by hand:

- If $K = 4$ the answer is 2.
- If $K = 3$ the answer is 3.
- If $K = 2$ the answer is 3.
- If $K = 1$ the answer is 2.

If the three values A, B, and K are not equal, we have $A \neq K$ or $B \neq K$, let's say that the latter occurs. Let's cover the hole of size B with $1 \times B$. Let's show that this is always optimal. If B is larger than A it is clear that the tile $1 \times B$ would not help us to cover the hole of size A. If B is less than or equal to A and we use $1 \times B$ to cover the hole of size A we can change this tile to the other hole (the one of size B) and move what is in it to where we were using the $1 \times B$ tile so we get a solution as good as the other. We conclude that we can assume that we have covered the hole of size B with the tile $1 \times B$.

Now we can assume that we have used the tiles $1 \times K$ and $1 \times B$ and only remains to cover the hole of size A. If $A \geq 5$ we consider three possible ways to cover the hole of size A:

- with the tile $1 \times A$
- with the tiles $1 \times (A-1)$ and 1×1
- with the tiles $1 \times (A-2)$ and 1×2

These three possibilities involve different tiles, since only 2 tiles have already been used, one of the 3 possibilities will be available, and the answer is 0. For the case $A \leq 4$ we can only use the 1×1, 1×2, 1×3 and 1×4 tiles, those still available. We conclude by testing all cases by brute force.

Problem F – Favorite Tree

Author: Daniel Bossle, Brasil

The expected solution is a dp: for each subtree from the real tree and from the goal tree, check whether the goal subtree is contained in the real subtree. A suggestion for notating a subtree is as the pair (root, parent). You can also fix the root for one of the trees.

The recursive dp step is to check whether a goal subtree is contained in a real subtree, knowing, from previous dp steps, which children of the goal subtree's root are contained in which children of the real subtree's root. This is a bipartite matching problem: on one side you have the goal children, on the other, you have the real children, you want to match each goal child to a different real child, and you know which of those pairings are valid (which you should model as edges).

The complexity is thus $O(\sum_{n<N} \sum_{m<M} \text{matching}(\deg(n), \deg(m)))$, which seems like $O(N^{4.5})$, but there cannot be many nodes with high degree, thus non-optimal matching algorithms should also pass. Also, the worst case for the matching is not when the degree is the maximum N, as that means that the children are all leaves and thus the bipartite graph is complete. For trickier matchings, the children must have a few nodes, which cut down N by a constant factor.

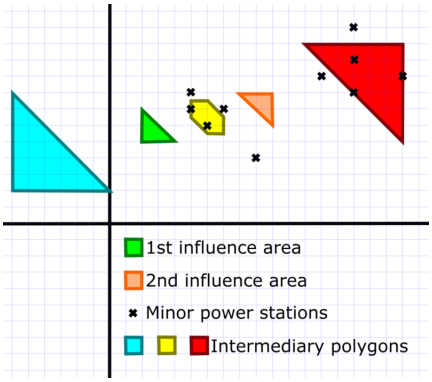
Problem G – Gravitational Wave Detector

Author: Mário da Silva, Brasil

First, calculate the three intermediary polygons that describe the suitable areas for the 3rd GWD station, one for each case:

- The middle GWD station is powered by the 1st main power plant;
- The middle GWD station is powered by the 2nd main power plant;
- The middle GWD station is powered by the minor power station;

To find each of these polygons, calculate the Minkowski sum / use rotating calipers on the two input polygons, in a different way each time. Next, for each such polygon, list which minor power stations it contains. You can do this in $O(NlogN)$ for each polygon, by sorting the candidate points.



Problem H – Horse Race

Author: Daniel Bossle, Brasil

The input can be summarized into a matching problem, between horse names and their positions. Each small race constrains that matching in two ways: To explain them, consider a small race where A B C was won by 3rd. The first constraint is that A B and C each cannot have placed above 3rd. The second constraint is that the 3rd horse must have been among A B C, which can be modeled as no other horse could've been 3rd. That is:

	1	2	3	4	5
A	X	X			
B	X	X			
C	X	X			
D			X		
E			X		

Similarly, the given sample input can be translated into the matrix

	1	2	3	4	5
A			O		O
B				O	O
C		O	O		O
D				O	O
E	O		O		O

There is also a linear solution to the problem: you can consider the subraces starting with the one with the worst winner. Any horse participating in those subraces will not influence races with better winners, and so can be assigned any available position at or below the winner. You also need to make sure to assign one of the horses that participated in *all* races with winner W to position W.

For the sample input, this means: b and d must be 4th or below, with one of them being 4th. We can choose that b came 4th and d 5th. a b c d and c b must be 2nd or below, with c or b being 2nd. We already assigned b and d, so c must be 2nd, leaving a to be 3rd. Finally, the horses left are only e, which gets the remaining position, 1st.

Problem I – Italian Calzone & Pasta Corner

Author: Daniel Bossle, Brasil

The expected solution is a simple $O((N \times M)^2)$ one, starting at each cell and simulating the best path that can be achieved from there.

This simulation can be done with a priority queue, taking always the earliest dish available, with an extra $log(N \times M)$ factor to the time complexity.

Or, you can take dishes out of order, with the only restriction that you cannot go from a dish to an adjacent earlier one. This allows a linear DFS or BFS to work. In other words, you can model the grid as a directed graph, with an edge between each pair of adjacent dishes going from the earlier dish to the later one, and you want to find how many nodes can be reached from each starting one.

Problem J – Joining a Marathon

Author: Giovanna Kobus Conrado, Brasil

We will sweep through time, simulating the run without Johnny, keeping a record of all runners in order and swapping adjacent runners whenever they intersect. This can easily be done by creating the R^2 updates by seeing when any two runners will be at the same place. There will also be updates for each photo. For each photo, it is easy to check whether some runner lies in the desired range by binary searching in the ordered list of runners.

From that first sweep, we may remove all photos that are already not trash. From this point on, the first set of runners is irrelevant.

Now a second sweep will be performed, but instead of the original runners, it will be Johnny throughout all queries. We will keep a segment tree that keeps, during the sweep, how many (previously trash) photos the runner that is currently at the i-th position appears in. This segment tree must handle range sums and swapping two adjacent values (which can be done with two point-queries and two point-sum updates).

Now, for every photo, we can again binary search on the range of runners that will be in such photo, and range sum one to all runners in that range. Whenever one runner passes the other, their values may be swapped. By the end of all photos, each value in the segment tree may be used to generate the solution of the corresponding query.

The complexity for the first sweep is $O((R^2 + P)\log(R + P))$ and for the second sweep is $O((Q^2 + P)\log(Q + P))$. Because of the heavier constants on the second sweep, as the log comes from segment tree updates and not only sorting, the second sweep will dominate the complexity.

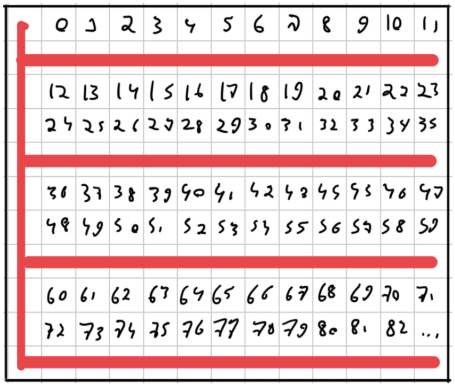
Problem K – Kind Baker

Author: Giovanna Kobus Conrado, Brasil

First, it is easy to see that $\lceil log_2 k \rceil$ toppings will be required, so the point is how to form exactly the number of combinations necessary while ensuring the forms remain connected.

Ignoring the connectivity constraint for a bit, it would be easy to list all numbers from 0 to $k-1$ in some slices of the cake and for every number x put topping i if the i-th bit of x is on.

We will keep the same idea, but add an extra area so that the region each topping is on is connected, like in the figure below.



Where the red part all toppings share, and each number is included in the toppings if the respective bit is one.

This, in a 100×100 grid, can comfortably support numbers up to the upper limits of k. Other patterns can also be used to ensure that the pieces are all connected, for instance, dividing the grid into many 3×3 subgrids, and painting on each 3×3 grid the leftmost and top cells.

The only extra detail is that since the region containing all toppings counts as a combination of toppings itself, the numbers listed must be from 0 to $k-2$. $k = 1$ requires no toppings but for most implementations it won't even need to be handled as a corner case.

Problem L – Lazy Printing

Author: Célio Passos, Brasil

Consider that the answer to the problem is K, i.e. the string T can be split into K substrings $T_1, T_2, \ldots, T_K$ such that $T = T_1 T_2 \ldots T_K$ and each T_i is a string with period at most D.

First, since any substring of a string with a period at most D also has a period at most D (in particular, this is true for substrings of T_2), we can assume T_1 is the largest prefix of T which has a period at most D. We can therefore solve it in $O(NK)$: simply keep finding the largest prefix that satisfies the restriction (this is a classical problem that can be solved with prefix function/KMP algorithm in $O(N)$ until the whole string is eaten. Since $K \leq \lceil \frac{N}{D} \rceil$, we can use this strategy when $D \geq \sqrt{N}$.

Now, suppose $D < \sqrt{N}$. Let's for each d ($1 \leq d \leq D$) compute the length of the largest suffix with period d of each prefix of the string S. For a fixed d, this can be easily solved by running a sliding window (of length d) over the string S in $O(N)$. This precalculation will then take $O(N\sqrt{N})$ running time in total. As we only care for the largest suffix with period at most D, we don't need to store the lengths for each d, only the largest of them (thus using only $O(N)$ space in total, which improves the constant factor by a lot). After that, the problem can be easily solved in linear time with dynamic programming.

Alternative solutions include:

- using a suffix array to calculate the longest common prefix of the suffixes of T and check how far to the right one can go with a fixed period, this works in $O(N \log N)$; and
- a $O(N)$ approach where the prefix function computation is modified to stop as soon as it reaches a prefix with period greater than D.

Problem M – Maze in Bolt

Author: Welton Cardoso, Brasil

The problem consists of, given a matrix P of 0's and 1's that describes a maze and a string S with 0's and 1's that describes the nut's internal pattern, check that it is possible to access the upper end of this matrix and go through it in such a way that it is possible to reach the lower end of it performing the movements allowed in S. This path must take into account the string S as a guide, that is, along the path, for each position j ($0 \leq j <$—S—) in S that has 1, you must have 0 in cell $P_{i,j}$ with i ($0 \leq i < R$) being the current row of traversal.

This problem is a variation of the classic maze problem where we want to find an exit given that the labyrinth was accessed by a certain entrance. Some algorithms known to solve this type of problem are the breadth-first search (BFS) and depth-first search (DFS). However, some important points are worth mentioning:

1. It is not necessary to perform a BFS or DFS for every position of S that has 1. A movement of S, either clockwise or counterclockwise, will shift all 1's at the same time, so just make sure that a next BFS/DFS state is reachable if in this new state all 1's of S are in a cell of P with 0.

2. Remember that P and S are, respectively, representations of a circular maze and a circular nut. Therefore, next position of $j =$ —S—-1 will be $j = 0$ and previous position of $j = 0$ will be $j =$ —S—-1.

3. Problem statement informs that: when the bolt and the nut are separated, the nut can be flipped. On a quick read, it might seem that this inversion makes no difference since it can be substituted by clockwise and counterclockwise movements. It might work for some patterns, but for others not, for example: 001011 → flip → 110100, see that it is not possible to reach 110100 with clockwise movements or counterclockwise. Thus, in some cases, it is necessary to actually flip the nut and check if there is a solution with the nut flipped.

The complexity of the BFS/DFS at worst is $\mathcal{O}(V + E)$, where V is the number of vertices and E is the number of edges in the graph. Therefore, we have that $V = R \cdot C$ (number of cells in matrix P) and $E = 4 \cdot R \cdot C \cdot$—$S$— (for each vertex we will have a maximum of 4 edges and, for each edge, it will be necessary to check if the positions of S are ok in relation to the current line of the matrix P). Therefore, the final complexity will be $\mathcal{O}(R \cdot C^2)$ which is sufficient for the given bounds for the problem.