Editorial

Tasks, test data and solutions were prepared by: Marin Kišić, Josip Klepec, Ivan Lazarić, Daniel Paleka, Ivan Paljak, Stjepan Požgaj and Paula Vidas. Implementation examples are given in attached source codes.

Task: Trol

Suggested by: Ivan Paljak
Necessary skills: proving conjectures, math, interval sums of periodic arrays

At first, this task might seem a bit frightening (for the first task) since it requires you to solve a bunch of queries with very large constraints and Marin's changes on the array elements seem relatively complex. Since this is the first problem called *trol*, you might suspect that an elegant solution is hiding somewhere.

But, let's start from the beginning. In the test data worth a total of 10 points the queries operate on elements that were not changed by Marin. Therefore, it was enough to output the value of $l+(l+1)+\ldots+(r-1)+r$ for each query.

For additional 20 points, the constraints were such that it was possible to simulate the process explained in the task description. In other words, you could simply loop over the numbers from l to r and simulate Marin's changes for each number. The problem of finding the sum of digits of a certain number is well known and well covered (**hint:** How do obtain the last digit of a number? How to delete the last digit?). This solution is implemented in the source file `trol_brute.cpp`.

In order to obtain full score on this problem, you needed to make a certain observation. This observation could be made in two ways which we will depict by sharing a couple of truthful anecdotes that occurred during the making of this round.

First way

The task author first told this problem to Marin. Marin is an experienced competitive programmer and he knows that the suggestion for the easiest COCI problem must not be too difficult. Therefore, he conjectured that something fishy must be going on with the array after changes. In a matter of minutes, he implemented the aforementioned solution (worth 30 points) and realized that the array is periodic. More precisely, he found out that $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 1, 2, 3, 4, 5, 6, 7, 8, 9, \ldots\}$. Marin didn't fully understand why is that the case, but he experimentally ~~proved~~ shown that his conjecture holds.

Second way

The task author then spoke to Stjepan. Stjepan really did recently receive his maths degree so it wasn't long before he remembered something he learned way back in primary school: "The number is divisible by 9 if and only if the sum of its digits is divisible by 9". He also noted that a stronger formulation also holds, that he remainder we get when dividing a number by 9 must equal the remainder we get when we divide the sum of its digits by 9. Since that remainder doesn't change when we apply Marin's operation and each positive digit has a distinct remainder, he concluded that we could simply replace each number with a corresponding digit. Then, it is easy to see that the array is periodic in the same way Marin did.

Whether your thought process follows Marin or Stjepan, one thing is certain, once you made this observation the task became easy. Suppose we divide the numbers from the query in consecutive blocks of size 9. It is easy to see that each block has the same sum which equals $1 + 2 + 3 \cdots + 9 = 45$. There is a total of $\lfloor \frac{r-l+1}{9} \rfloor$ complete blocks. We still need to add the elements from the final, incomplete block. The number elements in that block is $(r - l + 1) \mathtt{~mod~} 9$ and the first number is l. Therefore, we can answer each query in $\mathcal{O}(1)$.

Task: Lutrija

Suggested by: Marin Kišić
Necessary skills: math, primality testing, bfs/dfs

For 14 points, it was enough to first check whether $|A - B|$ is prime. If it is, then the array could simply be $\{A, B\}$. If it isn't prime, we can check for every $2 \leq x \leq 1000$ if x is prime and $|A - x|$ is prime and $|x - B|$ is prime. If all three holds, then the sought array is $\{A, x, B\}$. Otherwise, there is no solution.

To obtain the full score it was enough to note that the absolute difference between two prime numbers greater than 2 must be an even number. Since the absolute difference between each successive pair must be prime, we can conclude that the only numbers that could appear in the array are $A - 2$, A, $A + 2$, $B - 2$, B, $B + 2$ and 2. Out of those seven numbers we will leave only the prime ones. Now, if we represent each number as a node in a graph and connect the two nodes if their absolute difference is prime, the task boils down to finding a path between nodes A and B. This is a standard problem which we can solve using BFS.

Task: Džumbus

Suggested by: Vedran Kurdija
Necessary skills: trees, dynamic programming, complexity analysis

In the first and second subtask we can observe that the pairs of friends form a single chain. We will solve the first subtask using dynamic programming which in its state has a position in the chain P, the amount of džumbus we have at our disposal K and a binary flag B which denotes whether person at position P exchanged any solutions. The value $dp[P][K][B]$ reflects on only the first P people in the chain and it will denote the largest number of people that can exchange solutions with optimal distribution of leftover drink when it is also known (via B) whether person at position B has already done so. The general idea behind the transition is the following:

$$dp[P][K][0] = \max\{dp[P-1][K][0], dp[P-1][K][1]\}$$

$$dp[P][K][1] = \max \begin{cases} dp[P-1][K - D_{L[P]}][1] + 1 \\ dp[P-1][K - D_{L[P]} - D_{L[P-1]}] + 2 \end{cases}$$

The answer to query S_i equals to $\max\{dp[L[N]][S_i][0], dp[L[N]][S_i][1]\}$. The time complexity of this solution is $\mathcal{O}(N \cdot \max S_i)$.

Since there is no additional constraint on value S_i in the second subtask, the dp solution from the first subtask would not fit the time and memory constraints. Luckily, we can think about the same thing in a different light. We will again store the position P in our state and a binary flag B which have the same meaning as before. But now, the amount of džumus will be the value of our dp and the number of people that have exchanged the solutions will be moved to the state. The transitions are left as an exercise to the reader.

The answer to the query S_i is the largest R such that $\min\{dp[L[N]][R][0], dp[N][R][1]\} \leq S_i$. The complexity of this solution is $\mathcal{O}(N^2)$.

In the third subtask we don't have a chain, but a forest. A forest can be reduced to a single tree by introducing a dummy root node which we treat as a person with infinite D_i. In that way, we are sure that person does not affect the results.

We use similar dynamic programming solution as in the second subtask, except we are do it on a rooted tree. The value of $dp[P][R][B]$ is the smallest amount of džumbus we need to make R people that are in P's subtree to exchange solutions when it is also known whether P exchanged the solutions (bit B). When calculating the dp values for subtree P, we assume we already have all the dp values calculated for the entire subtree. We will also have an auxiliary dp_prefix which in its state holds the number of people in a subtree of P which have exchanged solutions at least once, as well as number K which denotes that we have taken into consideration the first K children of P thus far. Then, it is obvious that $dp[P][R][B] = dp\_prefix[R][B][\text{number of children of P}]$. Let $C[i]$ denote the i-th child of node P. The transitions are therefore:

$$dp\_prefix[R][0][K] = \min_{A+B=R}\{dp\_prefix[A][0][K-1] + \min\{dp[C[K]][B][0], dp[C[K]][B][1]\}\}$$

$$dp\_prefix[R][1][K] = \min_{A+B=R}\begin{cases} dp\_prefix[A-1][0][K-1] + D_P + dp[C[K]][B][1] \\ dp\_prefix[A-1][0][K-1] + D_P + dp[C[K]][B-1][0] + D_{C[K]} \\ dp\_prefix[A][1][K-1] + D_P + dp[C[K]][B][1] \\ dp\_prefix[A][1][K-1] + D_P + dp[C[K]][B-1][0] + D_{C[K]} \\ dp\_prefix[A][1][K-1] + dp[C[K]][B][0] \end{cases}$$

(Implementation details and corner cases should be carefully analysed by the reader)

When we add up all the states across all of the nodes, we get a total of $\mathcal{O}(N^2)$ states in the dp_prefix and every transition is calculated in at most $\mathcal{O}(N)$ steps. Therefore, the complexity can be bounded by $\mathcal{O}(N^3)$.

I know what you're thinking. There is no way this can be optimized any further.

To solve the fourth and final subtask you should note that, with careful implementation, the complexity of our solutions is in fact $\mathcal{O}(N^2)$. Obviously, it is impossible that a set of people exchanges more solutions than the size of that set. Therefore, when calculating dp_prefix, it is not necessary to check all options for A and B, just those which are possible (A cannot be larger than the sum of subtree sizes for already processed children increased by 1 and B cannot be larger than the subtree size of the current child). We will prove that the total complexity of dp calculations for each subtree is at most $\mathcal{O}(\text{veličina podstabla}^2)$. The proof inductive on the tree depth. The claim obviously holds for leafs. Let's fix a node and suppose we have calculated dp values for its children with the mentioned complexity. Suppose that node has x children with subtree sizes $Y_1, Y_2, ...Y_X$. The total complexity for calculating dp values of that node and its subtree can be expressed as:

$$\begin{aligned} \mathcal{O}\left(\sum_{i=1}^{X} Y_i^2 + \sum_{i=1}^{X}(1 + \sum_{j=1}^{i-1} Y_j)\cdot Y_i\right) &= \mathcal{O}\left(\sum_{i=1}^{X} Y_i^2 + \sum_{i=1}^{X} Y_i + \sum_{i=1}^{X}\sum_{j=1}^{i-1} Y_i Y_j\right) \\ &= \mathcal{O}\left(1 + \sum_{i=1}^{X} Y_i^2 + 2\sum_{i=1}^{X} Y_i + 2\sum_{i=1}^{X}\sum_{j=1}^{i-1} Y_i Y_j\right) \\ &= \mathcal{O}\left((1 + \sum_{i=1}^{X} Y_i)^2\right) \\ &= \mathcal{O}(\text{subtree size}^2) \end{aligned}$$

The total time complexity of the whole solution is therefore $\mathcal{O}(N^2)$.

Task: Trobojnica

Suggested by: Daniel Paleka
Necessary skills: constructive algorithms, mathematical induction, amortized analysis

In order to score 10% of the points, we need to find the necessary and sufficient conditions for the patriotic triangulation to exist.

Claim 1: In each triangulation with $N \geq 4$ there are at least two "ears", i.e., triangles that share two sides with the input polygon.

Sketch of the proof: Induction. Claim holds for a square. Let a diagonal divide the polygon into two smaller polygons. If one of them is is a triangle, that is an ear. Otherwise, we make an inductive argument.

Let there be a, b and c sides with colors 1, 2 and 3 respectively. It obviously holds that $a + b + c = N$. It is also clear that $max\{a, b, c\} < n$ because otherwise no ear would exist.

Claim 2: If a patriotic triangulation exists, numbers $\{a, b, c\}$ have the same parity.

Sketch of the proof: Double counting the number of edges colored 1 across the triangles:

$$2 \cdot (\text{number of diagonals colored } 1) + a = N - 3,$$

From which we conclude that a has the same parity as $N - 3$. The same holds for b and c by analogy.

The aforementioned conditions are also sufficient; this is easiest to prove by a constructive algorithm which solves this task.

In short, we will inductively build ears using the sides of two different colors without changing (prove it!) the equal parities of numbers $\{a, b, c\}$ in a new polygon which is missing a side. We also need to watch out whether the new polygon is monochrome.

One nice implementation is as follows:

For each vertex of a polygon we will remember its successor in the current polygon and the edge color which connects them. Searching for a spot to build a new ear is, in fact, simple. We will just circle around and build an ear on a vertex whose predecessor and successor edges are of the same colors and at least one of those colors is the one that is most frequent among the current polygon sides. If each time we start from the place where we've built our last ear, it can be proven that the complexity is an amortized $\mathcal{O}(N)$.

Implementations with time complexity of $\mathcal{O}(NlogN)$ should also receive full score on this problem.

Task: Zoo

Suggested by: Ivan Paljak
Necessary skills: bfs/dfs, determining the depth of tree

It is easy to determine which type of animal traversed the area last. If the starting and ending square contain `T`, it was obviously a tiger, otherwise it must have been a bull. Also, observe that, if a minimal number of animals traversed the rectangular area then no two animals of the same kind went one after the other.

We will think about the problem "backwards". Let's find all the squares that could have been traversed by the last animal before it left the rectangular area. Those squares form a connected component (in four directions) of the same type of footprints. Since all of those squares could have been traversed by the previous animal, we can change the value of those squares to the footprint of another animal type and add 1 to the solution. We should repeat this process until all only a single type of footprint remains. Naive implementations of this algorithm should score 45 points.

Let's now represent each connected component of the same footprint type as a node in a graph. Two nodes are connected if their corresponding components are neighbours, meaning that there exist at least two neighbouring squares in the matrix such that one belongs to the first and the other belongs to the second component. This graph is actually a tree. If we root the tree in a component where the upper-left square belongs, then the number of iterations of the previous algorhtm corresponds to the depth of that tree. The tree can be built in $\mathcal{O}(R \cdot S)$ and the number of its nodes is bounded by the same expression.