Editorial

Tasks, test data and solutions were prepared by: Fabijan Bošnjak, Nikola Dmitrović, Marin Kišić, Josip Klepec, Daniel Paleka, Ivan Paljak, Tonko Sabolčec and Paula Vidas. Implementation examples are given in attached source code files.

Task: Preokret

Suggested by: Nikola Dmitrović
Necessary skills: loops, arrays

This task is made for beginners and consists of three parts, each slightly more difficult than the last.

In the first part you were asked to determine what was the final score of the game. It was enough to simply count how many times did the numbers 1 and 2 occur in the input. This can be done simply by looping through the input data and storing that information in two variables.

In the second part you were asked to determine the number of ties. Since you already know how many goals each team scored in every moment (from part 1), you can simply introduce another variable which should be increased whenever the number of scored goals was the same for both teams. You also needed to make sure to include the starting score (`0:0`).

In the third and final part you were asked to determine what was the largest turnover. Note that each turnover consists of three parts. First, one team is losing and starts to score goals, then the score comes to a tie and finally that team starts winning. Suppose we store the difference in goals between the two teams. In that case, we are simply looking for the largest interval in which the difference at the start of the interval was negative and it became positive at the end (or vice versa).

Task: Grudanje

Suggested by: Marin Kišić
Necessary skills: prefix sums, binary search

In order to score 14 points, it was enough to simply simulate the process described in the problem statement.

The time complexity of that solution is $\mathcal{O}(NQN) = \mathcal{O}(QN^2)$.

In order to score additional 14 points, you could use prefix sums to speed up the solution above. Suppose that for every letter you built an array $pref$ in which an element at position i stores the number of occurences of that letter from the beginning of the string to the i-th letter (inclusive). For example, for a word `abcaab` and a latter `'a'`, the array $pref$ would be $[1, 1, 1, 2, 3, 3]$. That array is useful because now we can determine in $\mathcal{O}(1)$ what is the number of occurrences of a certain letter in an interval from L-th to the R-th letter in our original word. The formula for that is $pref[R] - pref[L-1]$.

The idea is now to precompute this array for every letter, thus obtaining an algorithm with complexity $\mathcal{O}(N(N\Sigma * Q\Sigma)) = \mathcal{O}((N^2 + Q)\Sigma)$, where Σ denotes the size of our alphabet.

In order to score all points, you could observe the following: if a word is prefect after i-th snowball, it will also be perfect after every other snowball thrown after the i-th one. This property allows us to use binary search. The check inside binary search is done similarly to the algorithm above (using prefix sums).

Time complexity: $\mathcal{O}((N\Sigma + Q\Sigma)\log N) = \mathcal{O}((N\log N + Q)\Sigma)$.

Task: Drvca

Suggested by: Marin Kišić i Josip Klepec
Necessary skills:

Task: Lampice

Suggested by: Tonko Sabolčec
Necessary skills:

Task: Sob

Suggested by: Paula Vidas i Daniel Paleka
Necessary skills: matematika, pohlepni algoritmi

Let's consider an ordered pair (a, b) to be *good* if $a \mathbin{\&} b = a$.

We can solve the first subtask by matching $a \in A$ with $b \in B$ for which $b \bmod N = a$ holds.

The second subtask can be solved with the following algorithm. Let $i_1 > i_2 > ... > i_k$ be the positions of ones in a binary notation of N. We will match the smallest 2^{i_1} elements of A and B such that we match those a and b for which $a \equiv b \mod 2^{i_1}$ holds. Then we take the following 2^{i_2} elements and match those that are equal modulo 2^{i_2}, and so on. The proof of correctness is left as an exercise for the reader.

The third subtask can be solved in multiple ways. One of those ways is to build a bipartite graph in which each node correspond to one member of the sets A and B. Naturally, we add edges between all good pairs of nodes. The problem now boils down *bipartite matching* which could have been implemented by a standard $\mathcal{O}(NE)$ algorithm, where E denotes the number of edges and N denotes the number of nodes. Note that we can deduce the upper bound on $E < 3^{10} = 59049$.

The other way is by using the following greedy algorithm. We will traverse through the elements of A from largest to smallest and we will match the current element with the smallest unmatched element from B which forms a good pair with our current element in A.

If we run this algorithm on a couple of examples, we can observe the following. Suppose that the largest element of A, i.e. $N-1$, is matched with $b \in B$. Then, $N-1-t$ will also be matched with $b-t$ for each $t \in \{1, 2, ..., b-M\}$. After we remove those matched pairs, we are left with the same problem on smaller sets $A' = \{0, 1, ..., N-1-(b-M)-1\}$ and $B' = \{b+1, b+2, ..., M+N-1\}$. This solution can easily be implemented in $\mathcal{O}(N)$.

Proof of the former observation:
Let $a = N-1$ and let b, as before, be the smallest element of B for which $a \mathbin{\&} b = a$. With index i we will denote the i-th digit of weight 2^i. If $b = M$ there is nothing left to prove, so we will assume that $b > M$ and introduce $k = b - M$. Let i be the position of the smallest significant one in b. Obviously it must hold that $a_j = b_j = 0$ for all $j < i$. If $a_i == 0$, then $a \mathbin{\&} (b-1) = a$ would hold and b wouldn't be the smallest element in B that can be matched with a. Therefore, $a_i = b_i = 1$. Now it's obvious that $(a-t, b-t)$ is a good pair for $t \in \{1, 2, ..., 2^i\}$. If $k \leq 2^i$, we are done, otherwise we observe the next smallest significant one in b and go through the same procedure inductively. The only thing left to prove is that there is always a $b \in B$ which can be matched with a, but we will leave this part of the proof as an exercise for the reader.