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: ad-hoc, basic data structures

More formally, the task statement asks us to divide the input data into two arithmetic sequences.

To obtain 20 points, it was enough to try all possible combinations and check whether some of them satisfies all conditions. The time complexity of this solution is $\mathcal{O}(N \cdot 2^N)$.

For additional 30 points, it was enough to observe that the smallest number in the input must also be the smallest number in one of the arithmetic sequences. Now, let's fix the second smallest number in that sequence. Now we have also fixed the difference between two successive elements of that sequence, meaning that we can easily calculate what number could be next in the sequence. We will add new numbers to that sequence, one by one, and each time check whether the remaining numbers form another arithmetic sequence. The time complexity of this solution is $\mathcal{O}(N^3)$.

As Bob the Builder would say: "Can we fix it? – Yes, we can¡'

We can note that the two smallest numbers of one arithmetic sequence could only be fixed in three different ways. If we observe the three smallest numbers in the input (denoting them with $A \leq B \leq C$), we can easily conclude that one arithmetic sequence must start with either (A, B), (A, C) or (B, C).

This observation leads us to an algorithm of time complexity $\mathcal{O}(N^2)$.

Finally, to obtain all points, we can speed up the idea from above. More precisely, we will speed up the check whether all elements that were not included in first sequence form another arithmetic sequence. We will keep two (multi)sets around. The first set will contain all remaining numbers, and the other will contain the differences of neighbouring elements from first set. When we remove the number from the first set, i.e. append it to the first sequence, we need to remove the difference between it and its neighbours from the second set. We also need to add the difference between the neighbours of removed element because they have now become neighbouring elements. The check whether the remaining number form an arithmetic sequence boils down to checking whether the smallest element from the second set is equal to the largest element in that same set. All operations on the aforementioned sets are supported by the STL collection `std::set` in C++. Naturally, other supported languages (with exception of C) also offer similar built-in collections.

The time complexity is $\mathcal{O}(N \log N)$.

Task: Lampice

Suggested by: Tonko Sabolčec
Necessary skills: hashing, binary search, centroid decomposition of a tree

The first subtask can be solved in multiple ways. We will describe the solution using hashing which will be useful for the final subtask. Let's root the tree in a node R. We are interested in all palindromic segments with one end in node R. For every node X, we will calculate two values:

$$down_x = x_0 B^{k-1} + x_1 B^{k-2} + \cdots + x_{k-1} B^0$$
$$up_x = x_0 B^0 + x_1 B^1 + \cdots + x_{k-1} B^{k-1}$$

Where x_0, x_1, …, x_{k-1} is an array of colors on the path from R to X ($color(R) = x_0$, $color(X) = x_{k-1}$) and B is the value of the hash basis. These two values can be determined using a single DFS traversal of our tree. The path from R to X is a palindromic segment if $down_X = up_X$ holds. If we root the tree in every node and apply this procedure, we will cover all cases. The complexity of this algorithm is $\mathcal{O}(N^2)$.

The second subtask is a classic, well-known problem of finding the longest palindromic substring which can be solved using Manacher's algorithm.

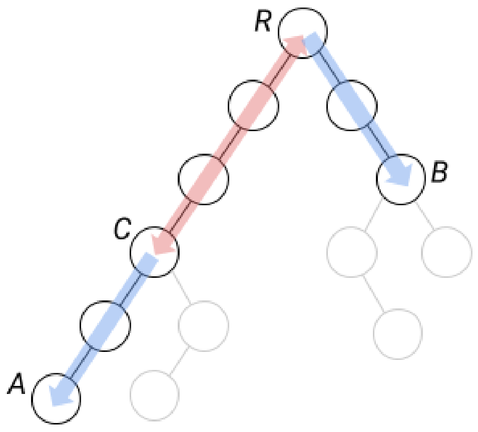
The solution for the third subtask is quite similar to the solution of the second subtask. The only difference is that we must apply Manacher's algorithm on a path between every pair of leaves.

We will begin our solution for the whole problem with the following observation: If there exists a palindromic segment of length $K > 2$, then there exists a palindromic segment of length $K - 2$. Therefore, we can use binary search on even and odd lengths to find the longest palindromic segment.

But, how do we check if a palindromic segment of fixed length exists in a given tree? To answer that question, we will first solve an easier version of that problem – we will check if there is a palindromic segment of a fixed length that contains the root of our tree, node R.

In a rooted tree, a path between two nodes A and B consists of two tree branches, where the length of path on one branch is greater or equal to the length on the other branch. We will divide the path between A and B into three parts: path from A to C, path from C to the root and path from the root to C, where C is chosen such that the lengths $A - C$ and $R - B$ are equal. Note that C is uniquely defined by the node A, since we only care about palindromes of fixed size. In order to check whether the path from A to B is a palindromic segment, it is enough to check the following:

- (1) Path from R to C is a palindrome (depicted in red)
- (2) Colors from C to A are the same as colors from R to B (depicted blue).



For each node of the tree we will calculate the values $down_X$ and up_X in advance in the same manner as it was described in the solution of the first subtask. The first check is simply done by comparing $down_C$ and up_C. The second check can be done by comparing the values:

- $down_B$
- (3) $down_A - down_{par(C)} \cdot B^{dep(A)-dep(C)}$, where $par(C)$ is the parent of node C and $dep(X)$ is the depth of node X.

Now we know how to efficiently check whether path from A to B is a palindromic segment, but that is still not enough because there are $\mathcal{O}(N^2)$ such pairs (A, B). Let's try to observe the problem from another angle: If we fix A, is there (at least) one B that satisfies all the conditions?. Let S_B be a set of hashes in which we will insert values $down_B$ of all nodes B of our tree. Then, for a fixed A, we can simply check condition (1) and we can check condition (3) by finding the corresponding value in S_B. We can assume that the complexity of all operations with S_B is constant if we use a hash table (`std::unordered_set` in C++, for example). The complexity of the whole check is therefore $\mathcal{O}(n)$.

Note: during implementation you need to make sure that the lowest common ancestor of of nodes in S_B and A is the root node R.

Now that we know whether a palindromic segment of certain length which passes through the root R exists, we can use centroid decomposition. If we perform this check for every decomposed subtree with the centroid as a root, we will cover all the cases. With centroid decomposition and a binary search, we end up with an algorithm of complexity $\mathcal{O}(n \log^2 n)$.

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 $a \,\&\, b = a$.

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 $a \,\&\, b = a$. 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 $a \,\&\, (b-1) = a$ 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.