



COCI 2016/2017

Round #1, October 15th, 2016

Solutions

Task Tarifa	Authors: Nikola Dmitrović and Nikola Pintarić
--------------------	--

How many megabytes would Pero have available in the $N + 1$ month if he didn't spend any? The answer is $(N + 1) * X$ megabytes. How many megabytes did Pero spend in the first N months since activating his plan? The answer is $P_1 + P_2 + \dots + P_n$. The number of megabytes that Pero has available in the $N + 1$ month of using the plan is obtained by subtracting the spent megabytes from the total possible number of megabytes.

Necessary skills: loops

Category: ad-hoc

Task Jetpack	Author: Branimir Filipović
---------------------	-----------------------------------

In order to solve this task, we need to find any path Barry can take from the initial position to any position in the last column.

We can do this by using BFS or DFS algorithm, after which we need to construct the path. Finally, all that's left is to format the path according to the task. For implementation details, consult the official solution.

Necessary skills: matrix operations, BFS/DFS

Category: ad-hoc, simulation

Task Cezar	Author: Stjepan Požgaj
-------------------	-------------------------------

First, we need to sort the words in the order which corresponds to the permutation. Now we see that, after encoding, a word must be lexicographically smaller than all the words that come after it.

Word A is smaller than word B if one of the two conditions is met:

1. A is a prefix of B
2. A and B differ in the i^{th} letter and A_i is lexicographically smaller than B_i (A_i is the i^{th} letter of A, B_i is the i^{th} letter of B)

To ensure that A that comes after B is not smaller than B because of the first condition, it is sufficient to observe all pairs and check if A is prefix of B.

The second condition is checked by constructing a directed graph consisting of 26 nodes, where each node represents one letter of the English alphabet.

The graph contains a directed edge $B_i \rightarrow A_i$ if and only if there exist words A and B such that A comes before B and they differ for the first time in the i^{th} letter.

In other words, if there exists a directed edge $B_i \rightarrow A_i$, then the letter that will be replaced with A_i must be lexicographically smaller than the letter which will be used to replace B_i .

Obviously, the solution will not exist if there is a cycle in the graph.

Given the fact that the number of nodes is quite small, a cycle can be detected in various ways. One of the simpler ways is using the Floyd–Warshall algorithm that performs in the time complexity $O(V^3)$, where V is the number of nodes.

If the graph doesn't have cycles, then the graph is DAG (directed acyclic graph), which means that its nodes can be topologically sorted.

For each node X, it holds that all nodes which are accessible from X in a topologically sorted array come before it.

From the aforementioned conditions, we can see that the letter that is represented by the first node in a topologically sorted array must be assigned the letter 'a', the second node the letter 'b', and so on.

Necessary skills: topological sort, DFS / Floyd-Warshall

Category: graphs

Task Mag	Author: Daniel Paleka
-----------------	------------------------------

If the amount of magic in the node with the minimal magic is 1, it is easily shown that the solution is a path of length 1, that exact node.

If there are nodes of magic 1 in the tree (in the rest of the text, *one-magic node*), it can be shown that a path of minimal magic (in the rest of the text, *optimal path*) comes in one of the two forms:

- a) a path consisting of k one-magic nodes, for some k
- b) a path consisting of $2k + 1$ nodes where only the central node is of magic 2, all the rest are one-magic nodes.

The proof reduces to breaking down the assumed minimal optimal path to two suitably-selected parts. Then one of the parts must also be optimal, except in the specified cases, which contradicts the fact that it is minimal.

In solving this task, we will use the weaker claim: the optimal path consists of at most one node that is not a one-magic node. Let's call a path through node v where all other nodes are one-magic nodes *v-good* path.

It is sufficient to find the longest good path for each node, because adding one-magic nodes to the end of the path decreases its magic.

In order to do this, we will use dynamic programming in two DFS traversals over the tree. Let's root the tree and, in the first DFS traversal, for each node v dynamically calculate:

- a) the longest *v-good* path that starts in v and is contained in a subtree of v ,
- b) the same thing for each *truncated subtree* of v that consists of subtrees of a prefix of its children,
- c) the same thing for each *truncated subtree* of v that consists of subtrees of a suffix of its children.

Now, using dynamic programming for prefixes and suffixes of children, we can determine the longest *v-good* path contained in the subtree of v that doesn't contain u , for each child u of node v .

(We take the maximum of the prefix to the child before u and the suffix from the child after u .)

The previous dynamic programming approach is used in the second DFS traversal. When we're in node u , using the dynamic programming approach for its parent v , we can calculate the longest *u-good* path that contains v . We must take into consideration the paths that go through v that do not contain any of its children, which is solved using an additional dynamic programming approach.

When we have the longest *v-good* path in its subtree and the longest *v-good* path for its parent, for a node v , by traversing the children of v , we can easily obtain the lengths of the two longest *v-good* paths that start in v , and whose union is the required *v-good* path. Now, we just compare the amount of magic of the longest *v-good* paths for all nodes v and take the smallest one.

Necessary skills: dynamic programming on trees, depth-first search, extremal principle

Category: dynamic programming, graphs

Task Kralj	Author: Domagoj Bradač
-------------------	-------------------------------

In the description of this solution, all labels are considered cyclical, which means that label $n+1$ is actually 1, $n+2$ is actually 2, and so on.

For any order of sending the elves, a position k exists such that none of elves has taken a walk from the k^{th} to the $k+1^{th}$ dwarf, because the k^{th} dwarf was busy. This is pretty obvious, because this holds for the dwarf that got its opponent last. However, a much stronger claim also holds: there always exists at least one position k such that **for each** order of sending the elves, none of them has taken a walk between the k^{th} and the $k+1^{th}$. Let's define R_i as the

number of dwarves whose assigned opponent has a label less than or equal to i , and let $P_i = R_i - i$. Notice that it must hold $P_n = 0$.

Let's take the smallest number out of all P_i , and say that it is obtained for position m . We will show that it is impossible for an elf to take a walk from position m to position $m+1$. Let's assume the contrary. This means that a sequence of positions exists $a, a+1, a+2, \dots, m-1, m$ where it holds that the number of elves whose opponent is located on some of these positions is larger than the number of these positions. However, it holds that the difference between the number of elves whose opponent is located at one of these positions and the number of these positions is precisely equal to $P_m - P_a$. This holds, regardless whether a is smaller or greater than m because $P_n = 0$. This difference cannot be positive, because of the selection of position m .

Now we can "cut through" the circle. In other words, observe it as an array of length n that starts at position $m+1$. Now we can solve the task using a greedy algorithm. Traversing from the beginning to the end of the array, we will push the strength of an elf into a *set* when we reach the position of its assigned opponent. We will select a dwarf's opponent the weakest elf from the *set* that can beat him, and if such doesn't exist, the weakest elf in the *set*. We will then remove the elf we selected from the *set*. The order of removal of elves from the *set* is the order in which they will enter the hall.

For solving the subtask worth 40% of points, it was sufficient to implement a greedy algorithm with the beginning of the array in the first position, but it was possible to solve it using binary search.

Necessary skills: greedy algorithms

Category: ad-hoc, greedy algorithms

Task Vještica	Author: Dominik Gleich
----------------------	-------------------------------

In order to successfully solve this task, it was necessary to notice the following: if we construct a prefix tree for a set of words, it is surely optimal to begin constructing the prefix tree so that all letters in common to all words are used as the initial chain of that subtree. In example, if we have words *aaab*, *baab* and *cab*, it is optimal to use letters *abc* to create a common prefix of all words, and after that, we are left with words *aa*, *ab*, *c*. Now that we don't have any more letters that are in all three words, we need to keep constructing the tree somehow.

Since we know that the remaining suffixes of the three words will not be in the same subtree, because they don't have any letters in common, we select two subsets of words for which we say that from now on are being built in **different** subtrees. In our case, it is optimal to select (*aa*, *ab*) as one subset, and (*c*) as the other. The subset division is performed in 2^n

ways, if there are n words in the set. Each word can be either in one or the other set in which we divide the words into.

It is also necessary to notice that, even though two words are in the same set, it doesn't mean that they will continue being built in the same subtree for at least one step, but it might be the case that they become divided immediately, recursively.

This kind of thinking leads us to solving the task using dynamic programming, where a state is described with the set of words (a bitmask), and we have 2^n such states. This dynamic programming approach will calculate the size of the smallest tree constructed from these words.

Since we traverse through all subsets of the state in each state, the total complexity is $O(3^n)$. For implementation details, consult the official solution.

Necessary skills: programming, recursion

Category: dynamic programming