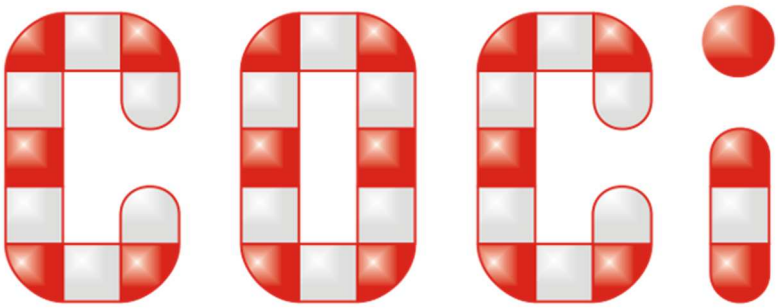

# COCI 2016/2017

Round #2, November 5th, 2016

## Solutions

| **Task Go** | **Author: Branimir Filipović** |
|---|---|

In order to calculate $E_i$, the number of Pokémon of type $P_i$ that Mirko can evolve, we must implement the algorithm given in the following pseudocode:

$E_i = 0$
while $M_i >= K_i$ :
    $M_i = M_i - K_i + 2$
    $E_i = E_i + 1$

Alternatively, we can use the following formula:

$$E_i = max(\left\lfloor \frac{(M_i - 2)}{(K_i - 2)} \right\rfloor, 0)$$

Why this formula is suitable is left as an exercise for the reader.

Now that we have calculated all $E_i$, all that's left is to output the sum of all $E_i$ and the first Pokémon $P_i$ that has the largest $E_i$.

*Pseudocode (written in Python 3.x):*

```
N = int(input())

maximum = total = 0

for i in range(N):
      pokemon = input()
      Ki, Mi = map(int, input().split())

      evolve = 0
      while Mi - Ki >= 0:
            evolve += 1
            Mi -= Ki
            Mi += 2

      total += evolve

      if evolve > maximum:
            maximum = evolve
            which = pokemon

print(total)
print(which)
```

**Necessary skills:** basic mathematic operations, for loops, algorithm for finding the maximum
**Category:** ad-hoc

| **Task Tavan** | **Author: Stjepan Požgaj** |
|---|---|

For 60% of points, the ink is spilled over only one letter, so it is sufficient to alphabetically sort the letters that could replace it and use the $X^{th}$ one. This is a good example of a task where a simple approach can get you a large number of points.

There are multiple solutions where it is possible to obtain all points. One of them is to convert the number X - 1 to a number in a numerical system where the base is K. For easier implementation, we pad the number with leading zeros so that the total number of digits is M. Let the digits of the new number be $a_1$, $a_2$, $a_3$ ,..., $a_m$, respectively. Then the $i^{th}$ unknown letter must be replaced by the $a_i^{th}$ letter from the sorted order of letters that could potentially replace the $i^{th}$ letter (the letters in the sorted order are 0-indexed).

The time complexity of this solution is O(M * K lg K).

**Necessary skills:** strings
**Category:** ad-hoc

| **Task Nizin** | **Author: Ivan Paljak** |
|---|---|

Let's first analyze the suboptimal solutions from the SCORING section.

For 30% of points, the task could have been solved using an exhaustive search. In other words, we could have modified the array in every possible way and, in the end, output the minimal number of moves that lead to the solution. Since we can make (*N - 1*) different moves on an array of length *N*, and because after each move the length of the array decreases by 1, we conclude that the complexity of this solution is O(*n!*).

Let's first observe the first and last member of the array. Since we wish to transform the array to a palindrome, the first and the last member must be equal in the end. If they are not equal at the given moment, obviously at least one of them must be joined with its neighbour. This kind of thinking leads us to a recursive formulation of the solution. Let *f(l, r)* denote the minimal number of moves it takes for elements A[l], A[l+1], …, A[r - 1], A[r] to transform to a palindrome. Given previous remarks,

$$l > r,\ f(l,r) = 0$$
$$l <= r,\ f(l,r) = \min(1 + f(l+1, r),\ 1 + f(l, r - 1),\ \mathbf{f(l+1, r-1)})$$

where the part written in bold is taken into consideration only if it holds A[l] = A[r]. Let's notice that in each step of the algorithm, the value A[l] is equal to the sum of all its predecessors, and A[r] is equal to the sum of all its successors, whereas the elements in the middle are not modified. Using a dynamic programming approach, more precisely the

memoization technique, the complexity of such a solution is O(n^2), which is enough for 60% of points. The implementation of this algorithm can be found in the file nizin_n2.cpp.

In order to obtain all points, we need to use the fact that the numbers in the input are positive. As in the previous paragraph, we approach the task from the outside within. When we focus on an interval [l, r], we differentiate between a couple of cases:

If it holds A[l] = A[r], then we don't need to join the outer elements, instead we continue solving the interval [l+1,r-1].

If it holds A[l] < A[r], then we surely can't profit from joining elements A[r] and A[r - 1] because their sum is necessarily larger than A[l] that can't be left unjoined. Therefore, we will join elements A[l] and A[l+1] and continue solving the interval [l + 1, r]. We analogously solve the case where A[l] > A[r].

Since we will decrease the interval for at least 1 in each step of the algorithm, we can conclude that the complexity of the algorithm is O(n), sufficient for all points.

**Necessary skills:** complexity analysis, breaking the problem into cases.
**Category:** Ad-hoc

| **Task Prosječni** | **Author: Mislav Balunović** |
|---|---|

The task can be solved in a lot of different ways.

One of the possible solutions is the following:

- In the first row, we write down the numbers $1, 2, \ldots, n - 1, \frac{n * (n-1)}{2}$
- Each following row but the last is obtained by adding $\frac{n * (n-1)}{2}$ to each number from the previous row
- The last row is obtained in the following way:
  For each column, if the numbers $a_1$, …, $a_{n-1}$ are the numbers written in that column so far, we write the number $n * a_{n-1} - (a_1 + a_2 + \ldots + a_{n-1})$ in the $n^{th}$ row in that column. By doing this, we achieved that the average of that column is equal to the next to last number in the column.

The only thing left is to make sure that the average of the last row is in that row. We leave this as an exercise for the reader to prove that the average is equal to the number in the last row and the next to last column.

**Necessary skils:** arithmetics, combinatorics
**Category:** mathematics

| **Task Zamjene** | **Author: Mislav Balunović** |
|---|---|

We will use the union-find data structure in solving this task.

Let's first describe the solution that isn't fast enough, but serves as a motivation for the actual solution.

What data must be remembered in each component?
Each component corresponds to a set of positions.
Let's denote the mutiset of all values contained in the array *p* on positions contained in component K with $P_K$, and the multiset of all values contained in the sorted array *q* on positions contained in component K with $Q_K$.
When joining components A and B into a new component C, we see that $P_C = P_A \cup P_B$ and $Q_C = Q_A \cup Q_B$.

It is crucial to notice that the array can be sorted if and only if $P_K = Q_K$ holds for each component.

After this, we can notice that it is not necessary to remember the exact multisets of numbers, but only their hash values. If number 1 is contained $c_1$ times in multiset S, number 2 $c_2$ times, … n $c_n$ times, then we define the hash vaule of set S as:

$$h(S) = c_1 * H + c_2 * H^2 + \ldots + c_{n-1} * H^{n-1}$$

When joining components, we simply add the hash values of the original components.

But, we still haven't mentioned how to answer query number 4.
We actually want to know the number of pairs such that it holds:

$$P_A + P_B = Q_A + Q_B \text{ (where now P and Q denote the hash values)}$$
$$\Leftrightarrow$$
$$P_A - Q_A = -(P_B - Q_B)$$

Now we know how to answer the query by maintaining map M where M[d] tells us how many nodes there are with the component's difference P - Q being exactly d.

The time complexity of this solution is O(Q lg N).
For implementation details, consult the official solution.

**Necessary skills:** union-find, hash
**Category:** graph theory

| **Task Burza** | **Author: Domagoj Bradač** |
|---|---|

We will root the tree in node 1. To begin with, let's notice that, after the $i^{th}$ move, the coin will be located in a node of depth *i*. It is clear that it is optimal for Daniel to label a node of depth *i* in the $i^{th}$ move. Now we can reformulate the task: given a set of nodes where none of the

nodes is a root and the nodes are of different depths, and there is no such node of depth *k* where none of its predecessors is labeled, does such a set exist?

The nodes of depth *k* will be called *leaves*. We can remove all nodes which don't have a *leaf* in its subtree (this includes the nodes of depth larger than *k*), because Daniel surely wins if the coin is located in such a node. Now the *leaves* are indeed leaves in the given tree. From now on, we will only observe trees obtained after removing unnecessary nodes.

Let's analyze the following simple algorithm: in the $i^{th}$ move, we will label a node of depth *i* and remove all nodes in its subtree. By doing this, we have removed at least *k - i + 1* nodes in the $i^{th}$ step, which means that, in *k* steps, we have removed at least $\frac{k(k+1)}{2}$ nodes. This means that, in the case $n \leq \frac{k(k+1)}{2}$, we know Daniel can surely win.

This is our motivation to find an even better bound, and we will use the following algorithm to do so: initially, for each node *v*, we will define *f(v)* as the minimal depth at which a descendant of *v* has more than one child. In the $i^{th}$ move, we will label node *v* of depth *i* with the minimal value *f(v)* from the tree and remove all its descendants.

Using the induction principle, we will prove that, by using this algorithm, Daniel wins if it holds $k^2 \geq n$. For the sake of induction, we will not observe a single tree, but multiple trees, which will actually represent the subtrees of the original tree. We will denote with *d* the minimal number for which it holds that, after *d* moves, there exists a non-removed node *v* for which it holds *f(v) = d*. If a number *d* for which this holds doesn't exist, it means that we have made *k* moves total, and in each move removed at least *k* nodes (we count the descendants, but also the predecessors of that node, that are all mutually distinct). But, that means that in each of *d* moves, we removed at least *2k - d* nodes (the worst case is for *f(v) = d*), and then we removed *d* nodes to depth *d*, and at least *2(k - d)* below depth *d*, because the tree *branches* at that depth.

Let's see what we have left: a new forest of trees that has at most $n_2 \leq n - d * (2k - d)$ nodes, and the new length to which the coin musn't propagate is $k_2 = k - d$. Now, from the condition $k^2 \geq n$, it follows ${k_2}^2 \geq n_2$, so the claim holds by the induction assumption.

We have shown that for $k^2 \geq n$ Daniel always wins, which means that we are left with solving the case when this doesn't hold. But, then $k < 20$, so we don't need to find a polynomial solution!

Let's denote the nodes in the order in which they appear in the *dfs* traversal of the tree. Now each node in the tree *covers* an interval of leaves, or, in other words, these are the nodes located in its subtree. We will solve the task using a dynamic programming approach. The state will be represented with a number and a bitmask *k* bits in size. Let *dp[T][mask]* denote whether we can *cover* the first *T* leaves by labeling only nodes of depth written in *mask*. The transition is simple, in a given moment we can choose any node out of at most *k* where the first leaf in the tree is labeled with T. Let's notice that, even though for each position *T* the transition can be *k* different nodes, each node will appear only in one position in the transition.

The total complexity is $O(2^{\sqrt{n}} * n)$.

**Necessary skills:** bitmasks, dynamic programming
**Category:** ad-hoc, dynamic programming