



## **COCI 2016/2017**

Round #6, February 4th, 2017

### **Solutions**

<b>Task H-indeks</b>	<b>Author: Adrian Satja Kurdija</b>
----------------------	-------------------------------------

When determining the h-index, it is obvious that we need to observe the papers with more citations. Therefore, we first sort the numbers by size, starting with the largest. We iterate over this array respectively (index  $K = 1, 2, 3, \dots$ ) and if the  $K^{\text{th}}$  element is larger than or equal to  $K$ , this means that  $K$  is a potential h-index (because the scientist has at least  $K$  papers with a citation score larger than or equal to  $K$ ). The task comes down to finding the largest such  $K$ , which is easily solved by storing the solution in an auxiliary variable and updating it while iterating the array.

**Necessary skills:** array sorting

**Category:** ad-hoc

<b>Task Telefoni</b>	<b>Author: Nikola Herceg</b>
----------------------	------------------------------

Notice that the series of ringing phones will always be a consecutive series from the one on the first table to the one at the  $p^{\text{th}}$  (prefix). Now we want to increase the prefix by adding a new phone. We can place the new phone on the table  $p+1, p+2, \dots, p+d$ , because if we put it to the right of table  $p+d$ , it will not ring. Obviously, it pays off the most to place the phone on table  $p+d$ , because this way the new prefix will be of maximal length. We implement this by iterating from left to right and keeping track of the location of the last ringing phone, and placing the new phone if the difference between the current position and the table with the last phone is equal to  $d$ . The complexity is  $O(n)$ .

**Necessary skills:** arrays

**Category:** greedy

<b>Task: Turnir</b>	<b>Author: Dominik Gleich</b>
---------------------	-------------------------------

First, we need to notice that the only relevant information for each number  $X$  is the number of numbers larger than it. This is relevant because only the larger numbers will continue to the next level, and all the other numbers will accompany the current number  $X$  to the top. This information is easily obtained by sorting the numbers.

Let's try to calculate the highest level  $X$  can reach. For the zeroth level, there obviously cannot be a single larger number than  $X$ , i.e., the number of numbers larger than it must be 0. In order to reach the first level, there must be at most  $n/2$  numbers larger than it, because otherwise, at least one number larger than  $X$  will 'fight' with it, which will result in  $X$  not passing to the next level. In the case when there are more than  $n/2$  numbers larger than  $X$ ,  $X$  should be at a level below the first one. Now, in order for  $X$  to be at the second level, the number of numbers larger than it must not exceed  $n/2 + n/4$ . Why?

It is easy to see that  $n/2$  numbers can be left on the level above at the other side of the tree, then we are left with  $n/2$  numbers with **X** in there. On the current second level, we make the same decision where we see whether we can leave all larger numbers in the second subtree sized  $n/4$ , in order for **X** to freely rise to the top of its subtree, to level 2. We follow this algorithm, subtracting the descending powers of 2 from the number of numbers larger than **X**. For implementation details, consult the official solution.

The total complexity of the algorithm is  $O(N \lg N)$ .

**Necessary skills:** arrays, recursive problem decomposition

**Category:** ad-hoc, mathematics

<b>Task Savršen</b>	<b>Author: Adrian Satja Kurdija</b>
---------------------	-------------------------------------

Instead of looking for the divisors of the given numbers (which is too slow), we can take a reverse approach, similar to the sieve of Eratosthenes: for each number  $d = 1, 2, 3, \dots, B$  check which number's divisor it is, i.e., iterate over its multiples between A and B and for each multiple V increase its sum of divisors:  $\text{sum}[V] += d$ . In the end, we iterate over the given array summing up the required absolute differences.

At first glance, it seems that this approach could be too slow, but it is empirically seen that it is not. Theoretically, we can deduce it this way: for each divisor d, we iterate over  $B/d$  multiples, which gives us the total complexity of  $B/1 + B/2 + B/3 + \dots + B/B$ , i.e.,  $B * (1 + 1/2 + 1/3 + \dots + 1/B)$ , which is approximately  $B \log B$ .

**Necessary skills:** the sieve of Eratosthenes

**Category:** number theory

<b>Task Sirni</b>	<b>Author: Domagoj Bradač</b>
-------------------	-------------------------------

In this task, we needed to find the minimum spanning tree in a graph where two nodes  $a$  and  $b$  are connected with weight  $\min(Pa \% Pb, Pb \% Pa)$ . This expression is equal to  $Pa \% Pb$  for  $Pa > Pb$  and vice-versa.

If two nodes exist with equal values, we can connect them with zero cost and observe them as a single node. Furthermore, we will assume that all node values are unique. For each node value  $Px$  we will iterate over all of its multiples  $k * Px \leq M$  where M denotes the largest possible node value. For each such multiple, we will find a node with the minimal value larger than or equal to x, and in the case when  $k = 1$ , the node with the minimal value strictly larger than  $Px$ . We denote this node with y, and add the corresponding edge to the list of edges we must process.

By doing this, we have at most  $O(M \log M)$  edges, which we can use to construct a minimum spanning tree using Kruskal's algorithm. If we naively sort the edges, we will obtain a solution worth 70% of total points, but since all edge weights are up to  $M$ , we can use *counting sort* and obtain an algorithm in the complexity of  $O(N + M \log M)$ .

We still have to prove the existence of a minimum spanning tree that holds only the edges that we singled out. Let's assume the contrary, that a tree exists that hold another edge, and has a smaller cost than any other tree that holds only the singled out edges. Let that edge be between nodes  $a$  and  $b$ . Let  $P_a < P_b$  i  $k * P_a \leq P_b < (k + 1) * P_a$ . Then, since we did not single out that edge, nodes  $c_1, c_2, \dots, c_n$  exist such that it holds  $k * P_a \leq P_{c_1} < P_{c_2} < \dots < P_{c_n} < P_b$ , that is  $P_a < P_{c_1} < P_{c_2} < \dots < P_{c_n} < P_b$  if  $k = 1$ . But, then we singled out edges  $(a, c_1), (c_1, c_2), (c_2, c_3), \dots, (c_{n-1}, c_n), (c_n, b)$ . Their total cost is  $P_b - k * P_a = P_b \% P_a$ . However, now we can remove edge  $(a, b)$  and instead of it place a path from  $a$  to  $b$  or some part of it and therefore have a connected graph with smaller or equal weight.

**Necessary skills:** Kruskal's algorithm, the sieve of Eratosthenes

**Category:** graphs, mathematics

<b>Task Gauss</b>	<b>Author: Mislav Balunović</b>
-------------------	---------------------------------

First, let's notice that, in the optimal solution, Gauss will perform the operation of the second type (leaving the current number on the board) only for one number. Let's denote with  $A$  the initial number, with  $B$  the final number, and with  $C$  the number we left on the board.

Secondly, let's notice that the number of operations between numbers  $A$  and  $C$  and between  $C$  and  $B$  is at most 20, because in each moment the new number is smaller than or equal to the half of the old number. It is not difficult to notice that the cost of the shortest way from  $A$  to  $C$  is equal to the cost of the shortest way from  $A/C$  to 1 (analogously for  $C$  and  $C/B$ ).

Given the specificity of the formula to calculate the costs, we see that the cost of the shortest path to 1 is equal for, for example, numbers 12 and 18. More precisely, let  $e_1, e_2, \dots, e_k$  be exponents of the prime numbers in the factorization of a number into prime numbers. Then the cost of the shortest path from that number to 1 is equal to all such numbers that have the same multiset of exponents  $\{e_1, e_2, \dots, e_k\}$ .

We can generate all such numbers and see that there is less than 250 of them.

We assign to each number  $n$  the smallest number with which it shares a multiset and denote it with  $k(n)$ .

Let's summarize the results so far:

The shortest path from  $A$  to  $B$  is over a number  $C$ . The cost from  $A$  to  $C$  is equal to the cost from  $A/C$  to 1, and that is equal to the price of  $k(A/C)$  to 1. Analogously for  $C$  to  $B$  and  $k(B/C)$ .

Let's denote with  $b[x][y][C][L]$  the shortest path from A to B such that it holds  $k(A/C) = x$  and  $k(C/B) = y$  and the total length of paths from A to C and C to B is equal to L (not counting the moves from C to itself).

This array can be calculated quick enough in the beginning of the programme with one optimization - we will calculate it only for pairs x and y such that  $x * y$  is at most 1 000 000, because otherwise the pairs are not useful to us.

How do we reply to queries when we have this array?

Let's go over all possible C and look at the formula for the total number of moves L, and denote with  $L'$  the number of moves when we leave C on the board:

The total cost is  $b[x][y][C][L - L'] + L' * \text{cost}[C]$ .

Now we see that, for a fixed L, we need to find a pair (C,  $L'$ ) that minimizes the value of the upper expression, which is actually a standard problem where we need to calculate the lower hull of the lines, and for each L find the intersection of the vertical line  $x = L$  with the hull.

We must also notice that all lines for a fixed C have the same slope, so it is sufficient to calculate the larger intersection of its lines with the y-axis and only add that one to the hull. The queries where L is smaller than 20 needs to be processed separately using dynamic programming.

**Necessary skills:** number theory, convex hull of lines

**Category:** number theory