CROATIAN OPEN COMPETITION IN INFORMATICS

2013/2014

ROUND 4

SOLUTIONS

COCI 2013/2014	**Task NASLJEDSTVO**
4th round, January 18th, 2014	**Author:** Adrian Satja Kurdija

It is possible to solve this task “mathematically”, by testing out certain cases. We leave the proof of this procedure as a practice for the reader and continue to explain a simpler solution.

We use a for loop to go through all possible values of the initial number of medallions **M**; for example, from 1 to 1000. For each such potential **M** we check whether it is a possible solution. How? The youngest daughter would take **M** div **N** medallions, which means there are **M** - **M** *div* **N** medallions remaining. If that number equals the actual remainder **O**, we have found ourselves a solution. Additionally, we have to keep track of the smallest and biggest solution in two variables; we are going to output them in the end.

Please note: *div* stands for integer division.

Necessary skills: for loop

Category: ad hoc

COCI 2013/2014	**Task GMO**
4th round, January 18th, 2014	**Author:** Adrian Satja Kurdija

It is possible to solve this task with dynamic programming, which we lack memory space for at first glance. We leave this option for the reader to implement for practice and continue to explain a simpler solution.

We use a for loop to choose the position where the swine gene would begin in the apple's DNA. Next, using an inner loop, we build this gene from the chosen beginning in a natural way: if the next character of the apple equals the next character of the swine gene we want to achieve, we move to the next character; else we have to insert the character we want to achieve. It never pays off to insert a character if we don't need to insert it. This is why this (greedy) algorithm is going to provide the best total cost for a chosen beginning. Finally, the solution is the minimum of costs for all possible beginnings.

Necessary skills: greedy algorithms or dynamic programming

Category: ad hoc

COCI 2013/2014	Task SUMO
4th round, January 18th, 2014	**Author:** Adrian Satja Kurdija

Firstly, we need to know the answer to the following question. For a given **K**, is it possible to divide the fighters into two teams so that in the first **K** fights there are no confrontations of fighters from the same team?

Actually we need to check whether a graph with **N** nodes (which represent fighters) and **K** edges (which represent fights) is bipartite. In other words, is it possible to paint its nodes with two colors so that the adjacent nodes are of different colors. We check this by actually painting the nodes: starting from node 1, we paint it white, then paint its neighbors black (because we have to), then paint its neighbors white (because we have to) and so on. This painting is done with BFS or DFS algorithm. The algorithm ends when we have either succesfully painted all the nodes, which means the painting can be done, or when we've come across a contradiction: an adjacent node of the node **X** from which we're currently expanding is already painted the same color as **X**, which means the painting is not possible.

Now is it clear that we are looking for the smallest **K** for which this procedure is going to fail. Testing out all possible **K** numbers is too slow. That is why we use binary search to find the value of the number **K**.

Necessary skills: binary search, graph search

Category: graphs

COCI 2013/2014	Task GUMA
4th round, January 18th, 2014	**Author:** Matija Bucić

To begin with, we will explain the solution of an easier version of this task, which appeared in the original Croatian edition of COCI, called HONI. The difference between this version and COCI's version is that in the easier version, the cuts had to be continuous (not interrupted).

First, we have to notice that, if the vertical part should be split into **K** parts, there must be **K** - 1 cuts on it, some which were prolonged from previous vertical parts and some which have just started. The required solution is the sum of cuts which have just started for all vertical parts: this way, we count each cut exactly once – when we start it.

For each vertical part, we want to know the number of cuts prolonged from previous parts. Let us assume that the previous vertical part was cut into 30 parts and that the current vertical part should be cut into 70 parts. Both numbers are divisible by 10. This means that the first 30 parts can be seen as 10 large parts by 3, and the new 70 parts as 10 large parts by 7. The big parts match and the cuts between them can be prolonged. The number of those cuts is 10 - 1 = 9. The remaining 69 - 9 cuts should have to be started and added to the final solution.

We can deduce from this example that, in general, the number of cuts that can be prolonged from the part **i** to the part **i** + 1 is equal to the greatest common divisor of numbers $\mathbf{a_i}$ and $\mathbf{a_{i+1}}$ minus one. (Let us notice that in the case when the numbers are relatively prime, this is also true because we can prolong 1 - 1 = 0 cuts.) We leave the proof of this procedure for the reader to implement for practice. The greatest common divisor can be calculated using, for example, Euclid's algorithm because the naive method is too slow for large test data.

The COCI version of this task was significantly more difficult: the cuts could "jump over" vertical parts. In other words, the cuts didn't have to be continuous. What would the minimal possible number of cuts be in this situation?

We are looking for the number of different heights which we can make a cut on. Again, given the current vertical part which should be cut into **T** pieces, we are looking for the number of cuts which can be prolonged from one of the previous vertical parts. The heights we're interested in are 1/**T**, 2/**T**, ..., (**T** – 1)/**T**.

When these fractions are irreducible, the denominators are divisors of **T**. For each divisor **d**, we come up with fractions **x**/**d**, where (**x**, **d**) are relatively prime because the fraction is irreducible. Has the cut **x**/**d** appeared before? Yes, if it's equal to **y**/**T'** for a previous part numbered **T'**, which means **T'** is a multiple of **d**. If we've marked the appearance of all previous divisors **d'** of **T'** (using, for example, an array of 0 and 1), it is simple to check if **d** has appeared before.

The complete procedure goes like this: for each divisor **d** of the number **T** (we find the divisors with complexity O(sqrt(**T**)) we check whether we've used it before. In other words, if there are cuts **x**/**d**. If not, we mark it as used and add all cuts **x**/**d** to the final solution, meaning that we are looking for number of **x**s smaller than **d** and relatively prime with **d**, which is the Euler function of number **d**. Its values can be calculated in advance using formulas for Euler's totient function.

Necessary skills: mathematical problem analysis, Euclid's algorithm

Category: mathematics

COCI 2013/2014	**Task ČOKOLADE**
4th round, January 18th, 2014	**Author:** Marin Tomić

To begin with, we will explain the solution of an easier version of this task, which appeared in the original Croatian edition of COCI, called HONI. In the HONI version, the candy is from the interval [1, 10^6], whereas in the COCI version it is from the interval [1, 10^8]. Also, **N** in HONI would not exceed 1000, whereas in COCI it would not exceed 100.

We will interpret the task in the following way: for an array of numbers of length **N**, for each x from 1 to **N**, we need to determine the minimal number with which we need to divide every array item so that the we are left with a group of exactly x equal numbers in the array or we need to determine that such number doesn't exist. If we mark the greatest number in the array with **V**, it is clear that all the solutions belong in the interval [1, **V**+1].

Our first solution is now formed, its complexity being O(**N*****V**). For each x from the interval [1, **V**+1] we can determine how many numbers give a certain quotient when performing division with x with the help of a counter in an array, which we update accordingly.

For a solution which performs for 100% of points, we need to come up with a faster way to calculate how many numbers give a certain quotient when performing division with the number x.

The following can be noticed:

- the numbers which give the quotient 0 when performing (integer) division with x are from the interval [0, x-1]
- the numbers which give the quotient 1 when performing (integer) division with x are from the interval [x, 2x-1]
- in general, the number which give the quotient k when performing (integer) division with x are from the interval [kx, (k+1)x-1]

If we can build a structure which can quickly give an answer to the query *How many numbers in the array are from the interval [a, b]* then we can determine how many numbers give the quotient k when performing division with x. Given the fact that the array numbers are not greater than 10^6, the structure can be a simple array A[] where A[t] stands for the number of elements in the original array smaller than t. Now we can calculate the number of numbers in the original array from the interval [a, b] with the formula A[b] - A[a - 1].

The greatest quotient k which can happen for a certain x is **V**/x. This is why we need to make **V**/x + 1 queries to determine how much numbers there are which give the quotient 0, 1, 2, ... **V**/x when performing division with x. Because we have to test all numbers from 1 to **V**+1 for a certain x, we have **V** + **V** * (1/1 + ½ + ☐ + … + 1/(**V**+1)) queries in total. Therefore, the complexity of this algorithm is O(**V** $\log_2$ **V**).

The COCI version of this task had different limitations: **N** ≤ 100, **V** ≤ 10^8. This version required a different approach.

For each number K in the array, we can observe the following array: [K/1], [K/2], [K/3], …, [K/K], [K/(K+1)], where [x] marks the floor of value x. We can notice that this array is going to consist of blocks of consecutive equal values. We will call number *i* **important** if [K/i] is different from [K/(i-1)].

Now we can use an algorithm similar to the initial solution of complexity O(**N***V**), except now we don't need to test every x from 1 to **V** + 1. Instead, it is sufficient to check only the important numbers for each element we have in the array because the not important number x will have all quotients equal to those of some important number.

Now the complexity is O(**N * the total number of important numbers * lg N**). It can be shown that for a number K, there exist sqrt(K) different important numbers at most, so the total complexity is O(**N * N * sqrt(V) * lg N**).

Consult the source code for further implementation details.

Necessary skills: mathematical problem analysis, preprocessing

Category: mathematics

COCI 2013/2014	**Task UTRKA**
4th round, January 18th, 2014	**Author:** Goran Žužić

Let us label $M_{ab}^{(k)}$ as the biggest advantage Mirko can achieve on a route with *k* steps **at most** which begins in village *a* and ends in village *b*. The task is to find the minimal k_{min} such that one diagonal element in the matrix $M^{(kmin)}$ is strictly positive. We also want to know the maximal such diagonal element in the matrix $M^{(kmin)}$, as it is the second required output number.

We can notice that if we know $M^{(p)}$ and $M^{(q)}$, we can calculate $M^{(p+q)}$ with a procedure similar to matrix multiplication. The complexity of this procedure is O($\mathbf{N}^3$).

These observations let us use binary search for *k* and then using fast (matrix) multiplication we calculate $M^{(k)}$. Unfortunately, the complexity of this procedure is O($\mathbf{N}^3 \log^2 \mathbf{N}$), which wasn't enough for 100% of points.

In order to achieve maximum points, it was necessary to calculate matrices $M^{(k)}$ for $k = 2^0, 2^1, 2^2$, … (complexity O($\mathbf{N}^3 \log \mathbf{N}$)) and then find the minimal *k* in the procedure which determines its digit by digit in binary notation (from the strongest to the weakest position). The total complexity of this procedure is still O($\mathbf{N}^3 \log \mathbf{N}$) which was sufficient for 100% of points.

Necessary skills: fast matrix multiplication, binary search

Category: graphs