

**CROATIAN OPEN COMPETITION IN  
INFORMATICS 2013/2014**

**ROUND 1**

**SOLUTIONS**

<b>COCI 2013/2014</b>	<b>Task TRENER</b>
<b>1<sup>st</sup> round, September 28<sup>th</sup>, 2013</b>	<b>Author:</b> Marin Tomić

We need to count which letters of the alphabet appear in input at least 5 times as the first letter of a surname. This can be accomplished using an array of size 26 and counting the number of occurrences of each first letter while reading the input. After that, it is sufficient to iterate over that array once and output the letters with count at least 5. It is also necessary to keep track whether any letter satisfied the condition and output "PREDAJA" if this was not the case.

Refer to the solution source code for implementation details.

**Necessary skills:** arrays, for loop, strings

**Category:** ad-hoc

<b>COCI 2013/2014</b>	<b>Task KUŠAČ</b>
<b>1<sup>st</sup> round, September 28<sup>th</sup>, 2013</b>	<b>Author:</b> Adrian Satja Kurdija

It can be shown that the following cutting strategy is optimal. We arrange the sausages in a single line, one after the other (thus obtaining a line segment consisting of **N** shorter line segments). Cutting this line into **M** equal segments yields the required solution.

Although we are conceptually making **M** - 1 cuts, some of them are not actual cuts, but fall between sausages (shorter line segments) instead. For example, with two sausages and four tasters, the first cut is real, dividing the first sausage in half, the second cut is not real because it is actually between the two sausages, and the third cut is real, dividing the second sausage in half.

We therefore need to count the "between" cuts. For the **K<sup>th</sup>** cut to be a between-cut, the first **K** out of the **M** portions must consist of the first **X** sausages, where **X** is a integer. In other words,  $(K / M)$  out of **N** sausages equals **X** sausages. **X** can then be obtained:  $X = (K * N) / M$ . It is now clear that **X** will be an integer (and the cut a between-cut) if **M** divides **K \* N**. We can simply use a for loop to check, for each possible **K** from 1 to **M** - 1, whether it is a real cut or a between-cut.

Alternatively, there is an explicit formula:  $solution = M - gcd(N, M)$ . Proof is left as an exercise for the reader.

**Necessary skills:** for loop, fraction and integer multiplication

**Category:** ad-hoc

<b>COCI 2013/2014</b>	<b>Task RATAR</b>
<b>1<sup>st</sup> round, September 28<sup>th</sup>, 2013</b>	<b>Author:</b> Matija Bucić

The first step towards solving this problem is finding a way to quickly compute the total income of a rectangular field. We will first describe a good solution of that subproblem.

It is possible to precompute in advance a **N**-by-**N** matrix **P**, where **P**[**x**][**y**] contains the sum of values in the rectangle with opposite corners in (0, 0) and (**x**, **y**). For the purposes of this problem, it is sufficient to calculate the matrix values with  $O(N^4)$  time complexity, iterating over the whole rectangle for each matrix cell. Of course, it is possible to carry out the computation in  $O(N^2)$ , which is left as an exercise for the reader.

This precomputation is useful since we can then determine the total income of any rectangle in time  $O(1)$  instead of  $O(N^2)$  using the inclusion – exclusion formula:

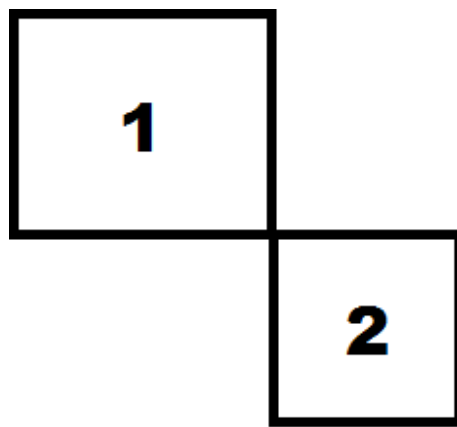
$$\text{sum}(x_1, y_1, x_2, y_2) = P[x_2][y_2] - P[x_1 - 1][y_2] - P[x_2][y_1 - 1] + P[x_1 - 1][y_1 - 1]$$

A simple way of finding all valid rectangle pairs is trying out all possible quadruples of points as opposite corners of the two rectangles and checking whether they have the same sum using the formula above. This solution has a complexity of  $O(N^8)$  and yields 20% of points.

The solution above can be easily optimized by iterating over triples instead of quadruples, immediately fixing the common vertex of the two rectangles. This reduces the complexity to  $O(N^6)$  and yields 40% of points.

For the final optimization, we can exploit the fact that individual unit square incomes are restricted to the interval between -1000 and 1000, which means that the total income of any rectangle falls between  $-1000 * N^2$  and  $1000 * N^2$ . Since **N** is at most 50, we can use an array of size  $2000 * N^2$  to count the rectangles with each possible total income.

We will demonstrate the idea on the case of rectangles sharing the upper left and lower right corners, as in the picture. We first iterate over all possible common rectangle vertices. Then, for a fixed common vertex, we iterate over all possible upper left corners of rectangle 1 and use the array to track the number of rectangles 1 for each possible sum. After that, we iterate over all possible lower right corners of rectangle 2 and, for each rectangle 2 with a sum  $S$ , look up the number of rectangles 1 with the same sum  $S$  and add that number to the solution. We repeat an equivalent procedure for rectangle pairs sharing the lower left and upper right corners.



The complexity of the algorithm described above is  $O(N^4)$  and it is sufficient to score 100% of points. Refer to the solution source code for implementation details.

**Necessary skills:** matrices, auxiliary arrays, preprocessing

**Category:** ad-hoc

<b>COCI 2013/2014</b>	<b>Task LOPOV</b>
<b>1<sup>st</sup> round, September 28<sup>th</sup>, 2013</b>	<b>Author:</b> Domagoj Čevič

The problem can be solved using a simple greedy algorithm. We first sort the jewellery pieces by value. Then, for each piece of jewellery starting with the most valuable, we do the following:

- we choose the bag with the smallest maximum mass out of bags that are able to store the current piece of jewellery, if there is such a bag available; we remove that bag from the set of available bags and add the value of the current jewellery piece to the solution
- if there is no available bag that can store the current piece, we skip the piece and continue

In order to implement the above algorithm, we need a data structure efficiently supporting the following three operations: insert a number, find the first number larger than some number  $x$  (or report there is no such number), remove a number. If using C++, a standard STL structure – *multiset* – can be used for this purpose.

Pascal programmers will need some more effort to solve this problem. For them we recommend the following algorithm, with a very similar basic idea as the algorithm above.

While the above algorithm requires coding a balanced binary tree or a Fenwick tree, the following algorithm requires only sorting and the binary heap, which are somewhat simpler to implement.

The binary heap is a data structure supporting three operations: insert a number, find the maximum number, remove the maximum number. It is one possible implementation of a priority queue (for example, in C++ STL).

The algorithm is as follows:

- sort the jewellery and bags together in a single array by mass/capacity
- iterate over the sorted array starting with the minimum mass/capacity
- if the current item is a jewellery piece, insert its value into the heap
- if the current item is a bag and if the heap is not empty, take the most valuable piece of jewellery so far (which is guaranteed to fit into the

current bag because of the sorting), remove it from the heap and add its value to the solution

The proof that the above two algorithms are correct is left as an exercise for the reader.

**Necessary skills:** heap or multiset, greedy algorithms

**Category:** greedy algorithms

<b>COCI 2013/2014</b>	<b>Task ORGANIZATOR</b>
<b>1<sup>st</sup> round, September 28<sup>th</sup>, 2013</b>	<b>Author:</b> Domagoj Čevič

Notice that the solution equals the size of a team multiplied by the number of participating clubs, while a club participates if it has a member count divisible by the team size.

The easiest method to count the participating clubs for some team size  $S$  is iterating over all club sizes and counting clubs that have a size divisible by  $S$ . The complexity of this algorithm is  $O(N * \text{maxS})$ , where  $\text{maxS}$  is the maximum club size, because the team size can be any value between 1 and  $\text{maxS}$ , and counting for each team size takes  $O(N)$ .

We need a faster method of counting the competing clubs. Since club sizes are limited to 2 million, we can use an array **a** of size  $\text{maxS}$  to store, for each possible size, the number of clubs with that size.

In order to count the participating clubs for some team size  $d$ , we need to compute the sum:  $\mathbf{a}[d] + \mathbf{a}[2 * d] + \mathbf{a}[3 * d] + \dots$ , which requires  $\text{maxS} / d$  steps.

Trying out all possibilities for  $d$  from 1 to  $\text{maxS}$  requires  $\text{maxS} / 1 + \text{maxS} / 2 + \text{maxS} / 3 + \dots + \text{maxS} / (\text{maxS} - 1) + 1$  steps, which approximately equals  $\text{maxS} * \lg \text{maxS}$ . Thus, the complexity of this algorithm is  $O(\text{maxS} * \lg \text{maxS})$ , which is sufficient to score all points.

**Necessary skills:** array manipulation, complexity calculation

**Category:** ad-hoc



<b>COCI 2013/2014</b>	<b>Task SLASTIČAR</b>
<b>1<sup>st</sup> round, September 28<sup>th</sup>, 2013</b>	<b>Author:</b> Marin Tomić

Let us first solve a simpler version of the problem, where the robot doesn't stop upon finding a matching serial number, but tries out all **N** segments instead.

Let  $f(W, S)$  be the number of suffixes of string  $S$  that have  $W$  as a prefix. Let  $S[x..y]$  denote a substring of  $S$  from position  $x$  to position  $y$ .

The number of comparisons that the robot will make for a word  $W$  with length  $L$  is then:

$BS + f(W[1..1], S) + f(W[1..2], S) + \dots + f(W[1..i], S) + \dots + f(W[1..L], S)$ ,  
where  $BS$  is the total number of segments that the robot has started comparison with. With our simplification assumption,  $BS$  always equals **N**, beaus the robot doesn't stop comparison upon finding a matching word.

A more efficient way to compute  $f(W, S)$  is needed. One possible method is building a suffix array of  $S$ . A suffix array is a set of all suffixes of a string  $S$  sorted lexicographically. Binary search can be applied to the suffix array in order to find the interval of suffixes beginning with the character  $W[1]$ . Let us denote this interval by  $[l_1, r_1]$ . The value of  $f(W[1..1], S)$  then equals  $r_1 - l_1 + 1$ . Next, within this interval, we need to find the interval of suffixes with  $W[2]$  as the second letter,  $[l_2, r_2]$ .  $f(W[1..2], S)$  is then  $r_2 - l_2 + 1$ . We repeat the procedure for all remaining characters in  $W$ .

The complexity of this search is  $O(L * \lg N)$ , which is fast enough since the sum of all query lengths will not exceed 3 000 000.

What about the full problem, where the robot stops upon finding a match?

For each word  $i$ , let us denote the first position where it appears with  $p_i$ . Let  $g(W, S, p)$  be the number of suffixes of string  $S$  with the prefix  $W$  starting at a position less than or equal to  $p$ . We can now derive a new formula for the total number of comparisons:

$BS + g(W[1..1], S, p_i) + \dots + g(W[1..L], S, p_i)$ ,  
where  $BS$  now equals  $p_i$  since we stop comparing after that position.

In order to make computing the function  $g$  easier, we will not respond to queries in the order that they appear in input, but sort them in a suitable order instead and store the solutions in an array. After computing all the solutions, we can output them in the original order.

We will sort the queries by  $p_i$  in ascending order. We will also utilize a structure supporting two operations: add 1 to some position, find the sum of an interval. This structure will enable us to find the number of important prefixes in an interval. It can be implemented using a Fenwick tree or an interval (tournament) tree.

We will process queries as follows:

- suppose we are currently responding to a query first matching at  $p_i$
- in the structure, we set 1 to corresponding positions for all suffixes up to  $p_i$ ; notice that it is not necessary to do so for all suffixes for each query, but only for suffixes after the one we stopped at in the previous query, since the queries are sorted by  $p_i$
- as in the simpler version of the problem, we find the intervals  $[l_1, r_1]$ ,  $[l_2, r_2]$ , ...,  $[l_L, r_L]$ ; however, we do not add  $r_i - l_i + 1$  to the solution, but instead the sum of the interval  $[l_i, r_i]$  in the structure, which corresponds to the number of suffixes starting at position  $p_i$  or less in the interval  $[l_i, r_i]$  in the suffix array
- we store the result into the solution array at the appropriate position

Finally, we just need to output the solution array.

There is one more detail left unresolved: how do we compute the numbers  $p_i$ ? Upon finding the last interval  $[l_L, r_L]$  for some word  $W$ , we know that all suffixes from that interval have  $W$  as a prefix. The first position where  $W$  appears is thus the starting position of the leftmost of those suffixes, that is, the suffix with the smallest index. We therefore need a structure for finding the minimum number in an interval. This can be done using an interval tree or a similar structure.

Refer to the solution source code for implementation details.

There are multiple algorithms for building a suffix array with varying complexities that can be found on the Internet. The official solution uses an algorithm with complexity  $O(N * \lg^2 N)$ . Also, both an interval tree and a Fenwick tree are implemented in the official solution. The complexity of a query in both structures is  $O(\lg N)$ .

The total complexity of the algorithm is  $O(N * \lg^2 N + query\_lengths\_sum * \lg N)$ .

**Necessary skills:** advanced data structures (suffix array, interval tree, Fenwick tree), binary search

**Category:** strings, data structures