CROATIAN OPEN COMPETITION IN INFORMATICS 2011/2012

Round 6

SOLUTIONS

COCI 2011/2012	**Task JACK**
Round 6	**Author:** Adrian Satja Kurdija

N is small enough to try out all the possible combinations of three cards. There are less than 100^3 combinations. We can use three nested *for*-loops: first one to choose the first card we will take, second one to choose the second card, and third one to choose the final card.

In the inner loop, we check if the sum of the chosen cards is greater than **M**. If it is, we continue to the next combination. If it's not, we check if this sum is greater than the best sum found so far and update that sum if it is. At the end we simply output the largest sum found.

Necessary skills: array manipulation, nested *for*-loops

Category: ad hoc

COCI 2011/2012	Task PROZORI
Round 6	**Author:** Adrian Satja Kurdija

After reading in 5**M** + 1 rows and 5**N** + 1 columns of the input, we can notice that only some fields are of importance to us. More specifically:

- for the 1st column of windows, it’s enough to analyze the 2nd column of the input matrix,
- for the 2nd windows column, it’s enough to analyze the 7th column of the input matrix,
- for the 3rd windows column, we need only the 12th column of the input matrix...

In general:
- for the **K**th column of windows, it’s enough to analyze the (5**K**-3)-rd column of the input matrix.

Inside each of the selected columns, the next four fields after each boundary ‘#’ are of interest to us: the number of asterisks in those four fields will determine the type for that window.

While traversing columns, we must keep track of the number of windows found so far for each type. Best way to do this is to have a small array with five integers, and increase the number at the corresponding index as we come across different types of windows. At the end, we just output the numbers from that array.

Necessary skills: character-matrix manipulation

Category: strings

COCI 2011/2012	**Task ZAGRADE**
Round 6	**Author:** Bruno Rahle

The first step is to find matching pairs of brackets. This is easily done by traversing the expression from left to right and using stack to keep track of currently unmatched open brackets. More precisely, when we encounter some open bracket, we push it's location to the stack. When we encounter a closed bracket, this bracket and the bracket at the top of the stack will form a matching pair. There are some wrong approaches for matching the brackets, and they were worth 40% of the total number of points.

The next step is to generate all the possible expressions. There are two ways to do this, using bitmasks or using recursion. Recursion can decide which pairs of brackets to use, and some other function can remove those pairs and store the obtained expression. Here is the pseudocode:

```
choose(pair_index, total_pairs, removed_pairs):
      if pair_index = total_pairs:
            call gen_expression(removed_pairs)
            return
      add pair_index to removed_pairs
      choose(pair_index+1, total_pairs, removed_pairs)
      remove pair_index from removed_pairs
      choose(pair_index+1, total_pairs, removed_pairs)
```

Here we can substitute recursion with bitmasks. Idea is that any integer can tell us which brackets to remove if we see how that integer is written in binary form. If there is a digit 1 at some index in binary representation, than we will remove the corresponding pair of brackets, and otherwise we won't. Now we can count from 0 to $2^{\mathbf{N}}$ - 1, and generate all the possible expressions without the use of recursion. Here is a good tutorial on this approach:
http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=bitManipulation

Finally, we must sort the obtained expressions and remove duplicates.

Solution that doesn't take care of duplicates gets 70% of the total number of points.

Necessary skills: string manipulation, bitmasks, recursion, stack

Category: strings, sorting

COCI 2011/2012	**Task REZ**
Round 6	**Author:** Anton Grbin

First thing to notice is that, in order to obtain as many parts as possible, no three cuts will intersect in a common point. This is easy to see: if there is an intersection of three cuts, then we will get an extra part by moving slightly one of those cuts.

Now we can find a maximum number of parts as a function of number of used cuts:

$$number\ of\ parts(r) = 1 + \sum_{i=1}^{r} i$$

So we must find the **minimum r** for which $number\ of\ parts(r) \geq K$. Since our function is obviously monotonic, we can use binary search to find the minimum **r**.

If we output only the expected **r**, we get 50% of the total number of points.

There are many ways to find the actual cuts, and here is the simplest one. Make the $\mathbf{i}_{th}$ cut by connecting these two points:

(5000 - **i**, -5000) and (-5000, -5000 + **i**)

Necessary skills: binary search, plane geometry

Category: ad-hoc

COCI 2011/2012	**Task PASTELE**
Round 6	**Author:** Goran Gašić

Let’s formulate the expression for colorfulness of some set:

$$max_{i,j}(max(|R_i - R_j|, |G_i - G_j|, |B_i - B_j|)).$$

This can be rewritten as:

$$max(max_{i,j}(R_i - R_j), max_{i,j}(G_i - G_j), max_{i,j}(B_i - B_j)).$$

If we look at R, G, and B values of colors as points in space, colorfulness is equal to the largest side of the smallest bounding rectangular cuboid. Since values of the input coordinates are rather small, this leads us to a solution that traverses each of the possible rectangular cuboids and checks if there are at least **K** points inside it. The number of points in any rectangular cuboid can be found in constant complexity using the inclusion-exclusion principle. This solution earns 50% of the points.

Since colorfulness is equal to the largest side, we won’t lose anything if we check only cubes. This optimization will earn additional 30% of the points.

Finally, we can use binary search to find the smallest cube with one corner fixed that contains at least **K** points.

Necessary skills: binary search, inclusion-exclusion principle

Category: computational geometry, searching

COCI 2011/2012	Task KOŠARE
Round 6	**Author:** Gustav Matula

Each box can be represented using a bitmask: **i**-th bit will tell us whether **i**-th toy is present in this box. Our problem can now be reformulated: find the number of subsets of boxes such that total *bitwise-or* has no zeroes in it's binary representation.

We could use dynamic programming and find solutions for each *f(masks_left, current_bitwise_or).* This solution has complexity of O(**N** * $2^{\mathbf{M}}$) and earns 50% of the points. We won't go into details of this solution, but various implementations can be found among competitors solutions.

Let's see how to come up with faster solution. Let **b[mask]** be the number of masks from the input that are submasks of **mask**. Then the number of subsets whose *bitwise-or* is a submask of **mask** is equal to **c[mask]** = $2^{\mathbf{b[mask]}}$. Once we have **c** calculated, we get the solution in complexity O($2^{\mathbf{M}}$) by using the inclusion-exclusion principle.

But how to get sequence **b**? Let's take some mask **1mask** that begins with 1. Here **mask** denotes rest of the bitmask. Let's say we already know the number of it's submasks that begin with 1. Number of submasks that begin with 0 is equal to number of submasks of **0mask**. This suggests recursive approach: we solve our problem separately for masks beginning with 0 and 1, and then combine those solutions into the total solution.

Complexity of finding **b** is T($2^{\mathbf{M}}$) = O($2^{\mathbf{M}}$) + 2T($2^{\mathbf{M-1}}$) = O(**M** · $2^{\mathbf{M}}$).

Total complexity becomes O(**NM** + **M** * $2^{\mathbf{M}}$).

It is also possible to avoid using inclusion-exclusion principle, since we can calculate **c** within at the bottom of the same recursion we use for finding **b**. Examine the official solutions for details.

b could also be found by brute force in complexity O($3^{\mathbf{M}}$), and this approach is worth 70% of the points.

Necessary skills: bit masks, divide & conquer approach, inclusion-exclusion principle
Category: combinatorics