CROATIAN OPEN COMPETITION IN INFORMATICS 2014/2015

1st round, October 18th, 2014

solutions

Task PROSJEK	**Author: Goran Gašić**

Let S_k be the sum of the first k numbers in sequence A. It holds

$$B_k = \frac{S_k}{k}$$

It follows

$$B_{k+1} = \frac{S_k + A_{k+1}}{k+1}$$

From here we get the expression for A_{k+1}

$$A_{k+1} = (k+1) \cdot B_{k+1} - S_k$$

We calculate elements of the sequence A_{k+1} and their sum S_k iteratively by using one loop over sequence B.

Necessary skills: operators, sequences
Category: ad-hoc

Task KLOPKA	**Author: Adrian Satja Kurdija**

Iterating once over given points (or during the input) we find the leftmost point (the one with the minimal x-coordinate), the rightmost point (the one with the maximal x-coordinate), the lowermost point (the one with the minimal y-coordinate) and the topmost point (the one with the maximal y-coordinate). We do this using the standard algorithm of finding minimum/maximum: remember the minimum/maximum so far, compare it to the new value and change it if needed.

The four vertices found make a rectangle, but we need a square. In order for the square to cover all points, its side has to be equal to the longer side of the mentioned rectangle. When we've found the side length of the square, we need to square the value (multiply it by itself) to get the required area.

Necessary skills: for-loop, finding the minimum and maximum value
Category: ad-hoc

Task PIRAMIDA	**Author: Marin Tomić**

Firstly, let us notice that changing the word direction from row to row doesn't have any impact on the number of individual letters in a certain row.

A naive solution would be to simulate writing the letters into each row and count the number of appearances of individual letters in the matrix `num[26][N]` and then output the value from the matrix for each query. The memory complexity of this solution is O(**N**) and time complexity O($\mathbf{N}^2$+**K**), which is enough for 50% of total points. You can find an implementation of this solution in the file `piramida_n2.cpp`.

For a more effective solution, we need to examine how string `s` looks written down in each row. There are two options:

1) `s = w[i..j]` for some `i` and `j`. In other words, string `s` is a substring of string `w`.
2) `s = w[i..L] + x * w + w[0..j]` for some `i` and `j`. In other words, string `s` consists of a suffix of string `s` (possibly an empty one), then the whole word `w` repeated `x` times and, finally, a suffix of string `w` (possibly an empty one)

For example, if the word `w` would be "`ABCD`", the 12th row of the pyramid would be:

```
CDABCDABCDAB = CDA + 2 * ABCD + AB = w[1..4] + 2 * w + w[0..2]
```

In order to determine `x`, `i` and `j` for a certain row, we need to be able to determine what letter the word in that row begins with. It is easily shown that for row `r` that position is equal to the remainder of dividing number `r * (r-1) / 2` with `m`. It is necessary to carefully implement this formula because `r` can be very big. When we have calculated the position, it is easy to determine parameters `x`, `i` and `j`.

Now we can come up with the formulas for certain cases. Let `f(c, i, j)` be the number of appearances of `c`th letter of the alphabet in the substring `w[i..j]`. The formulas are the following:

1) `number_of_appearances = f(c, i, j)`
2) `number_of_appearances = f(c, i, L) + x * f(c, 0, L) + f(c, 0, j)`

To implement this solution effectively, we need to be able to quickly calculate the value of function `f`. We can do this by constructing a matrix `p[26][L]` where the element at index `[i][j]` tells us how many times the `i`th letter of the alphabet appears in the substring `w[0..j]`. Then we calculate `f(c, i, j)` by the formula `f(c, i, j) = p[c][j] - p[c][i-1]`.

The memory complexity of this solution is O(**M**), and time complexity O(**M**+**K**), which was enough for all the points in HONI. This solution is implemented in the file `piramida_honi.cpp`.

In COCI, this solution was enough for 70% of total points because the maximal string length was bigger, so a matrix of the dimension `26 * L` couldn't fit in 32 MB. It was necessary to solve queries separately for each letter. In this case, instead of a matrix of the dimension `26 * L`, a matrix of the dimension `L` was enough. This solution is implemented in the file `piramida_coci.cpp`.

Necessary skills: preprocessing, mathematics
Category: ad-hoc, preprocessing

Task MAFIJA	**Author: Adrian Satja Kurdija**

A greedy algorithm works in this task: if there is a person X whom nobody has accused, we can declare that person as a mobster. If that person accused person Y, then the person Y cannot be declared as a mobster and that person can be removed completely too. When removing person Y, we need to decrease the count of times the person Z has been accused, so that the person Z can eventually be declared as a mobster.

After repeating this procedure as much as we can, we will end up with cycles. In other words, there won't be a person that hasn't been accused. Then any person can be declared as a civilian and removed, so we can apply the aforementioned procedure again and so on, until there are any unmarked persons left.

Readers that are well informed about graphs will notice that the task is basically finding a maximum independent set on a pseudoforest. It pays off to choose **leaves**, delete their neighbours and repeating the procedure until there are cycles left (although solvable). This solution is completely equivalent to the previous one, but is easier to visualize.

Both cases need to be implemented carefully so the complexity of the algorithm is O(**N**).

Necessary skills: graphs or greedy algorithms
Category: ad-hoc

Task ZABAVA	**Author: Adrian Satja Kurdija**

Firstly, we need to notice that the sequence of students moving into the buildings is irrelevant: all that matters is how many students move into an individual building, because the parties in a certain building happen completely independently of other buildings. Therefore, we can assume that we are given M numbers: the number of students moving per building.

Let us observe a building and assume that it will be emptied exactly P times and see how it should be emptied optimally. Let us assume that the first time we emptied it was after x(1) students moved in it, the second time after a new x(2) students moved in it, and so on until the final time we emptied it after x(P) students, after which an additional x(P + 1) students moved into it until the end.

The noise level before the first time it was emptied is $1 + 2 + \ldots + x1 = x1(x1 + 1)/2$, and the analogous formula holds for the noises after the first time it was emptied, the second time and all the rest until the end.
The total noise level in a building is therefore:

$$\tfrac{1}{2} * \mathrm{sum}(\ x_i(x_i + 1)\ ), \text{ for } i = 1 \ldots P + 1.$$

If we discard the division by two, this noise level is simply equal to the sum of squares x_i^2 plus the regular sum x_i, but that sum is constant and equal to the total number of students who moved into that building. So, we need to minimize the sum of squares of P + 1 numbers, and the sum of these numbers is given.

From the arithmetic and quadratic mean inequality, it holds that the sum of squares is minimal when the numbers are equal. Here, it is occasionally impossible to achieve because of (in)divisibility, but the numbers will be "almost equal", in other words, they will differ from one another only by 0 or 1. This division and the required sum can be easily calculated in constant time by using simple formulas. This is an effective way of solving the problem for one building and a fixed number of times that building is emptied.

Now the task can be solved by dynamic programming. If dp(m, k) marks the least possible noise level we can achieve in the first m buildings with k times of emptying it, that value is calculated so that we try every possible way of emptying the mth building (let's call it P) and check whether the given noise level, which is:

dp(m - 1, k - P) + (solution for the mth building with P times of being emptied, as described above), is the minimal noise level so far. The final solution is, of course, dp(M, K). The complexity of this algorithm is O(**MK**^{2}).

Necessary skills: mathematical problem analysis, dynamic programming
Category: dynamic programming

Task KAMP	**Author: Antonio Jurić**

Let us observe the case when the camp is placed at house number 1. Because the village is represented as a **tree** graph, the path from the camp to any individual house to which the volunteers must go is **unique**. Let us assume that Mirko must **return back to camp** after he drives all the teams.

When he drives one team to their house and it turns out that another team has to go to a house that is on the way to the first house, Mirko will of course drive both teams in order to save time. Because of the assumption that Mirko has to return back to camp, the solution would then be **twice the sum of all edges** that connect the camp and all the houses the volunteers go to.

Since the task condition is that Mirko has to stay and help the last team (so, not going back to camp), the best solution is to always drive the farthest team last because that way we drive only once through the tree edges that connect the farthest house. Now the final solution is **twice the sum of all edges** that connect the camp and all the houses the volunteers go to **minus the length of edges in the longest branch**.

Finding the edges that connect the camp and houses the volunteers go to can be done in complexity O(**N**), and when we apply this algorithm in a way that we try to place the camp in every house, the final complexity would be O(**N**^{2}) and that solution is good enough for half of total points.

Let K be the minimal subtree of the given tree that contains all houses the volunteers have to go to and possibly some other houses that are in the way between the volunteers' houses. Let S be the sum of all edges of subtree K. Then the solution for a house X is:

solution[X] = distance[X] + 2**S – farthest[Y],*

where *distance[X]* is the minimal distance between house X and subtree K or, more specifically, some house Y from subtree K. Additionally, *farthest[Y]* is the length of the longest chain from house Y in subtree K. This formula assumes that we know the maximal length of the longest chain for each house in K, which is possible to calculate by a careful DFS traversal of the tree. The complexity of this solution is O(**N**) and gives total points. For details about calculating the longest chain from every house in K, see the source codes.

Necessary skills: tree, DFS
Category: graph theory