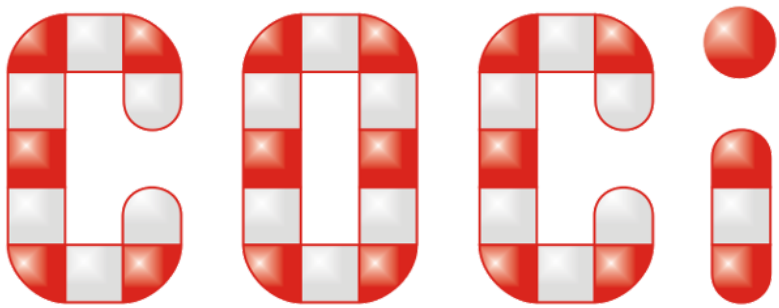

COCI 2016/2017

Round #3, November 26th, 2016

Solutions

Task Imena	**Author: Luka Barišić**

In order to solve this task, we read the book word by word. Let *cnt* be a counter initially set to zero. While iterating over the words of the book, we increment the counter by 1 each time we come across a word that is also a name. Additionally, if we come across a word that has a punctuation mark as its last character, it means that the current sentence is ending, so we need to output the counter and reset it to zero.

Pseudocode (written in Python 3.x):

```
def isName( word ):
    # cut the last letter if it's a punctuation mark
    if( word.endswith( ('.', '?', '!') ) ):
        word = word[:-1]
    if (not word[0].isupper()):
        return False
    for c in word[1:]:
        if( not c.islower() ):
           return False
    return True

n = int( input() )
words = input().split()
cnt = 0
for word in words:
    if ( isName( word ) ):
     cnt += 1
    if( word.endswith( ('.', '?', '!') ) ):
        print( cnt )
        cnt = 0
```

Necessary skills: for loop, word by word input, checking the last character in a string
Category: ad-hoc

Task Pohlepko	**Author: Tonko Sabolčec**

The solution that was good enough for 50% of points works under the assumption that a field's right and down neighbours are different. This way, we can start from the initial field and in each step compare the neighbours - we will always move the token to a field that is alphabetically smaller.

The problem arises when both neighbours are different. This problem can be solved by keeping track of, in each step, a list of all optimal positions in which we could have ended up

after a certain amount of steps. We start with the list containing the initial field (0, 0) and in each step update the list so that we first check the minimal value of the neighbours of all positions in the current list, and then create a new list that will contain all neighbouring positions with that value. Since we can reach a field in 2 ways, we must be careful not to add the same position twice to the list, because this way we would copy the number of appearances of the same position in each iteration.

Necessary skills: matrices, strings
Category: ad-hoc

Task Kroničan	**Author: Dominik Gleich**

It was possible to get partial points worth 40% of total points by trying out each permutation of glasses, where the label of the glass in the permutation denotes spilling the content of the glass with that label to one of the glasses to the right of that label in the permutation. We will always choose spilling over such that the effort is minimal, since we don't care which of the remaining glasses we spill into.

For 100% of total points, we use dynamic programming. Let the state be a bitmask of the glasses we haven't yet spilled into another glass, and the content of the rest of the glasses is contained within these glasses. The transition we can make is picking one of the glasses from the set and spilling the content into any other glass from the set. The total complexity of the transitions is $O(N^2)$, and can be potentially accelerated, but there is no need. The total complexity is $O(2^N * N^2)$.

For implementation details, consult the official solution.

Necessary skills: dynamic programming, bitmasks
Category: dynamic programming

Task Kvalitetni	**Author: Mislav Balunović**

Let's assume that we've processed the expression in the standard way, using the stack data structure and therefore obtained a tree of expressions.

First, let's notice that, if an expression can achieve value X, then that expression can achieve any value in the interval [0, X]. Because of this, it is sufficient to calculate the maximum for each subexpression.

Let a quality expression A consist of expressions A_1, A_2, …, A_k.

We distinguish between two cases:

- Smaller expressions are combined with an addition operation:
  Then the maximum of expression A is equal to the sum of the maximums of expressions A_1, …, A_k or L_k if the sum is too high.
- Smaller expressions are combined with a multiplication operation:
  If we assume that we don't have the constraint on the maximum of subexpressions, we could assume, for each subexpression, that it's value is exactly L_k / k, and then it is easily shown (for example, using the inequality of arithmetic and geometric means) that it is the maximum

  Let's denote the maximums of subexpressions with V_1, …, V_k and assume, without loss of generality, that it holds $V_1 \square V_2 \square \ldots \square V_k$. Let's denote with $X_1 \square \ldots \square X_k$ the values that these subexpressions will hold in our solution.
  We distinguish between two cases:
  - ***L_k / k V_1***
    In this case, the best solution is obtained by setting all expressions to L_k / k.
  - ***L_k / k > V_1***
    In this case, we set $X_1 = V_1$, reduce L_k by V_1 and repeat the procedure for X_2, …, X_k.
    It is easily shown that, this way, we get the optimal solution, and for the formal proof it is crucial to take advantage of the following lemma. We leave the proof as an exercise to the reader.

    Let there exist i, j between X_1, …, X_k such that $X_i < V_i$, $X_i < X_j$.
    Then a positive real number d exists such that, if X_i is increased by d, and X_j reduced by d, the product of numbers X_1, …, X_k is increased, and all other conditions remain satisfied.

Necessary skills: arithmetics
Category: mathematics, greedy

Task Zoltan	**Author: Stjepan Požgaj**

In order to determine the length of such longest strictly increasing subsequence, we must, for each position X in the initial sequence, determine the length of the longest strictly increasing subsequence starting at a position to the right of X and ending at position X (the sequence is read from right to left), and the number of ways in which we can achieve that maximum. The same idea applies for the longest strinctly decreasing subsequence. We can do this in a relatively simple fashion, using the Fenwick tree data structure in the time complexity of O(N * log N).

We can notice that the solution is a union of a strictly increasing and a strictly decreasing subsequence such that the largest element of the strictly increasing subsequence is smaller than the smallest element of the strictly decreasing subsequence. If A is the length of the

lnogest strinctly increasing subsequence ending at position X (including position X), and B the same for the strictly decreasing subsequence, and if num_A, num_B are, respectively, the number of ways to obtain them, then the maximum length of the numbers to the right of X (including position X) is A + B - 1, and the number of ways for getting this solution is num_A * num_B.

The required maximum length is the maximum of the described maximum lengths for each position. We denote this number with R. Then the number of ways for which we can achieve this length is the product of the number of ways for all positions where the maximum length is equal to R multiplied with 2^{N-R}.

The factor 2^{N-R} is necessary because, if a solution consists of R numbers, then each of the remaining N-R numbers could be placed independently before or after all numbers.

For additional details, consult the official solution.

The time complexity of the solution is O(n * log n).

Necessary skills: longest increasing subsequence, Fenwick tree
Category: dynamic programming

Task Meksikanac	**Authors: Ivan Paljak, Domagoj Bradač**

The problem can be broken down into two simpler problems:

1. For a given polygon, find all integer points contained in its interior or on the edge.
2. For each possible position of the polygon, determine whether a fly exists with its position being equal to an integer point in the polygon, after its translation.

The first part of the algorithm will be solved using *ray casting*. A point is located within a polygon if and only if an arbitrary ray from that point intersects with the polygon in an odd number of points. However, there are special cases, because the ray can touch a polygon's vertex, and can contain a polygon's side. For details, consult the official solution or the following post.

At the same time, we will check all integer points having the same x coordinate. We can find the intersections of each polygon's side with a vertical line through that x coordinate, and use the *sweep line* algorithm to check, for each point on the line, if they are inside of the polygon. This way, the complexity of the first part of the algorithm will be $O(Xp * (Yp + N \log N))$. But, we can perform the entire procedure in only one *sweep*, by maintaining a *set* of lengths, so we don't have to sort the intersections for each x coordinate. The complexity of this procedure is $O(Xp * (Yp + N) + N \log N)$.

We will reduce the second part of the algorithm to a problem in one dimension. Let (X_1, Y_1), (X_2, Y_2)... (X_k, Y_k) denote all integer points within a given polygon translated such that its

smallest X and Y coordinates are 0. We can represent these points with a binary string S_1, where we write 1 at position $2(Yp + 2) * Xi + Yi$ for each of the *k* points inside of the polygon, and write 0 for the remaining positions in the string. Now, translating the polygon for (X, Y) corresponds to translating the string for $2(Yp + 2) * X + Y$. We define the string S_2 that represents the flies in the same way.

We are left with determining, for each possible polygon translation, whether the bitwise *and* of the translated string S_1 and string S_2 is equal to 0. We can do this by using a *bitset* in the complexity of $O((Xp * Yp)^2)$, but with a very small constant. But, a faster solution exists: we will reverse the string S_2, and observe strings S_1 and S_2 as polynomials in the standard notation. From their product, we can make out the required information. The details of this approach is left as an exercise to the reader. If we implement the multiplication of polynomials using *FFT*, this part of the algorithm is of the complexity $O((Xp * Yp) * log(Xp * Yp))$.

The total complexity of the algorithm is $O(Xp * N + N \log N + (Xp * Yp) * log(Xp * Yp))$.

Necessary skills: geometry, multiplication of polynomials
Category: geometry, ad-hoc