CROATIAN OPEN COMPETITION IN INFORMATICS

Round 5

SOLUTIONS

COCI 2011/2012	**Task KRIŽALJKA**
Round 5, March 17th, 2012	**Author:** Bruno Rahle

The first part of the solution is detecting the first letter of word **A** that also appears in word **B**. It can be solved simply using two nested for-loops (the outer one to iterate over letters of **A**, the inner one over letters of **B**). A useful side-effect of such a solution is that the first found letter pair will be the two first (leftmost) such letters in their respective words, i.e. we will have automatically obtained the indices of the word intersection.

After finding the word intersection, we need to output the crossed words. The simplest method is, again, using two nested for-loops, the outer one iterating over indices of **B**, and the inner one over indices of **A**. On every iteration of the inner loop, we simply check whether we need to output a letter of **B** or a letter of **A**, or none of them (in which case we output a dot).

Required skills: for-loop, strings

Category: ad-hoc, strings

COCI 2011/2012	**Task EKO**
Round 5, March 17th, 2012	**Author:** Adrian Satja Kurdija

A naive solution is to try all possible sawblade heights, from lowest to highest or vice versa, until the correct height is found. We can simply try a height value by iterating over all trees and summing the cut-off parts. This algorithm has complexity O(**N** * max_height), which is too slow.

A faster solution can be obtained by reducing the number of heights that we need to try. Let us assume that we have tried out a height of **H'**, cutting off less than **M** metres of wood. This result implies that the needed height **H** is less than **H'**. Conversely, if by cutting at a height of **H''** we obtained at least **M** metres of wood, the needed height **H** must be greater than or equal **H''**.

The natural algorithm to apply here is binary search: we keep the upper and lower bound of the possible height interval, and in each step we try a height **H'** in the middle of that interval and, depending on the result, reduce the search interval to the upper or lower half.

The problem can also be solved by sorting the tree heights from highest to lowest, setting the cutoff height to the highest tree height and gradually lowering it to the height of each lower tree until cutting off at least **M** metres of wood. Detailed analysis of this approach is left as an exercise to the reader.

Required skills: arrays, binary search

Category: binary search

COCI 2011/2012	**Task DNA**
Round 5, March 17th, 2012	**Author:** Adrian Satja Kurdija

This task can be solved using dynamic programming. Let *f(***K***)* be the smallest number of mutations needed to convert the first **K** characters to A. Conversely, let *g(***K***)* be the smallest number of mutations needed to convert the first **K** characters to B.

If the **K**th character is already equal to A, then obviously
*f(***K***)* = *f(***K**-1*)*.

On the other hand, if the **K**th character is equal to B, we have two options:

1. Change the **K**th characters using a first-type mutation, giving
*f(***K**-1*)* + 1 mutations;

2. Change the prefix of length **K** (second-type mutation); in this case, we first need to convert the first **K**-1 characters to B (in order to convert them all to A using the prefix mutation), which requires *g(***K**-1*)* mutations, resulting in a total of
*g(***K**-1*)* + 1 mutations.

Therefore, if the K-th character is equal to B, then
*f(***K***)* = min{ *f(***K**-1*)* + 1, *g(***K**-1*)* + 1 }.

We also need to derive analogous relations for *g(***K***)*.

Now the problem can be easily solved by calculating *f(*1*)*, *g(*1*)*, *f(*2*)*, *g(*2*)*, *f(*3*)*, *g(*3*)* and so on until obtaining the result *f*(**N**).

The problem has another, simpler solution. We can iterate over the string **backwards**, converting each encountered character to A. If we find an A, we can simply continue. However, if we find a B, we have to decide which mutation to use. It turns out that a correct strategy is to look at the character in front of B. If A precedes the current B, then we convert only the B, and if another B precedes the current B, we convert the whole prefix. The proof of correctness of this algorithm, as well as a trick to quickly flip the whole prefix, is left as an exercise to the reader.

Required skills: mathematical analysis

Category: dynamic programming

COCI 2011/2012	**Task RAZBIBRIGRA**
Round 5, March 17th, 2012	**Author:** Frane Kurtović

Notice that we don't need the whole words to solve the problem, only the first and last letter are important. Therefore, it is sufficient to memorize, for each pair of letters, the number of words that begin and end with the respective letters.

Now we can select all possible combinations of letters in the corners of the square (26^4 possibilities). For each combination, we simply add the number of possible squares with the respective corners.

On each edge, we can place any word starting and ending with the appropriate letters. This word selection would always be independent if we could use the same word multiple times in the same square. If the number of possible words on the appropriate edges is denoted by *A*, *B*, *C*, and *D*, the solution would be *A*B*C*D* (ignoring the fact that we cannot select the same word multiple times).

However, if there are two edges requiring the same pair of beginning and ending letters, for example if the first two edges satisfy that constraint, the solution is *A*(A-1)*C*D*. It follows from the fact that we cannot choose from all *A* words for the second edge since we have already used one word up for the first edge.

By generalizing this result, we find that if *K* edges require the same pair of beginning and ending letters, and we have *A* such words, we can select the *K* edges in *A*(A-1)*....*(A-K+1)* ways.

The complexity of the solution, not counting input, is $O(26^4)$.

Required skills: mathematical analysis

Category: combinatorics

COCI 2011/2012	**Task BLOKOVI**
Round 5, March 17th, 2012	**Author:** Anton Grbin

Let us observe a stable arrangement of rectangles. The stability condition requires that the X-barycentre of all rectangles other than the lowest one has distance of at most 1 unit from the X-centre of the lowest rectangle. If we also factor in the mass of the lowest rectangle, we obtain the X-barycentre of the whole arrangement which also has distance of at most 1 unit from the X-centre of the lowest rectangle.

Let **dx(S)** be the difference of the x-coordinate of the rightmost point of some rectangle in the arrangement **S** and the X-barycentre of the arrangement. This value stays constant if the arrangement is translated along the X-axis.

Let us denote a stable arrangement of the top **k** rectangles with $\mathbf{S_k}$. We will deem $\mathbf{S_k}$ as optimal if there exists no arrangement of the top **k** rectangles with a larger **dx** value than $\mathbf{S_k}$.

The required solution of the problem is $\mathbf{dx(S_{n-1}) + 1}$, for some optimal $\mathbf{S_{n-1}}$.

The optimal arrangements can be determined using dynamic programming. For each state we memorize **dx** and the total mass **M** of the current optimal arrangement.
The trivial case is $\mathbf{S_1}$, with values **dx** = 1, **M** = mass of the **last** rectangle in input.

Determining $\mathbf{S_{j+1}}$ from $\mathbf{S_j}$:
The rectangle **j+1** (counting from the top) is placed under the current arrangement, with two possible cases:

- the X-barycentre of $\mathbf{S_j}$ coincides with the upper right corner of the new rectangle
- the X-barycentre of $\mathbf{S_j}$ coincides with the upper left corner of the new rectangle

We simply select the new arrangement with the larger **dx** value among the two possibilities.

Required skills: understanding weighted averages, mathematical analysis

Category: dynamic programming

COCI 2011/2012	Task POPLOČAVANJE
Round 5, March 17th, 2012	**Author:** Filip Pavetić

Notice the following: if, for a given position *i* in the large word (the one describing the street), we can find the longest short word (tile) that can be placed starting at position *i*, the problem is reduced to finding the union of intervals, which is solvable using a simple sweep.

The remaining problem is efficiently finding the longest tile for each position, which can be done in one of the following ways:

a) suffix array

A large number of problems with strings, including this one, can be solved using a suffix array. It is a sorted array of all suffixes of a string and can be computed using multiple methods. The simplest one, with complexity $O(\mathbf{N} \log^2 \mathbf{N})$, using hashing is sufficient for this problem.

It is important to notice that the indices of suffixes, whose prefix is one of the given **M** words, are grouped together in a suffix array, forming a suffix interval. This interval can be found using binary search for each word. After finding all the intervals, we need to find, for each suffix, the longest of **M** words whose interval covers that suffix, which can be found using sweep. After that, another sweep can easily determine the tileable positions, using the longest word that can be placed at each position.

There are other efficient solutions using hashing.

b) Aho-Corasick tree

The Aho-Corasick tree can be used to find many short words in a long one using a single pass over the long word. The tree can be constructed with complexity O(sum_of_short_word_lengths), while finding matches depends on their number.

The basic idea behind the tree is well described in the following materials:
http://www.cbcb.umd.edu/confcour/CMSC858W-materials/Lecture4.pdf
http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides04.pdf

For the purposes of this problem, while building the tree, for each node we can keep the data about the longest of the given short words that is a suffix of the prefix represented by a particular node. Let $M(x)$ denote that value for node x. $M(x)$ is then the maximum of:

- depth(x), if a short word ends in x
- M(failureLink(x)) *

*failureLink is described in the given materials

Building the tree still has complexity O(sum_of_short_word_lengths), while the (since we are not interested in all match positions, but only the maximum length for each position in the long word) complexity of searching through the long word is linear in its length. When encountering a letter, we try to descend down the tree. If it is not possible, we follow the failureLinks until finding either a node which we can descend from, or the root of the tree. In the node where we have ended up, we take the previous computed maximum and set the interval bounds for the future sweep.

Required skills: suffix array, Aho-Corasick tree, sweepline

Category: strings, sweepline