

CROATIAN OPEN COMPETITION IN INFORMATICS

2013/2014

ROUND 6

SOLUTIONS

Special thanks to Bruce Merry for sharing his writeup.

COCI 2013/2014	Task VJEKO
6th round, March 8th, 2014	Author: Marin Tomić

We will split the sample into two parts; the first part being before the asterisk (let us call it S) and the second part after the asterisk (let us call it T). The file name matches the pattern if it is in the form of S + R + T, where R is some string (possibly an empty one). Therefore, the file name must begin with S and end with T. However, this is not a sufficient condition.

If we take the sample "ab*bc", we have S = ab and T = bc. The file name "abc" begins with S and ends with T, but it still doesn't match the sample "ab*bc". It is necessary to check whether S and T overlap in the file name. In other words, is $\text{length}(S) + \text{length}(T) > \text{length}(\text{file name})$.

Hence, if a word doesn't begin with S or doesn't end with T or the aforementioned condition with lengths is met, we output "NE" (Croatian for no), else we output "DA" (Croatian for yes).

A solution which does not check whether S and T overlap is sufficient for 50% of points.

Necessary skills: string comparison

Category: strings

COCI 2013/2014	Task FONT
6th round, March 8th, 2014	Author: Marin Tomić

We need to find the number of different subsets of the set of given words such that the combined words in the subsets contain all lowercase letters of the English alphabet. In a set of **N** words, we can choose 2^N different subsets. Because **N** is an integer less than 25, we have a maximum of $2^{25} = 33554432$ subsets. This number is small enough for us to be able to pass through all possible subsets and check whether they contain the whole alphabet.

A regular way of passing through all subsets of a set is using a recursive

function.

```
f( i ):
    if i = N + 1: check whether the inserted sets contain the whole
alphabet
    (we've passed through all words)
    else:         add the ith word to the set
                  call f( i + 1 )
                  remove the ith word from the set
                  call f( i + 1 )
```

Additionally, we need to decide how to represent these sets in the computer's memory so we could implement the addition and removal of a word in the set.

One possible way is using an array **appeared**[26] which keeps track of how many times each letter of the alphabet appears in the current subset and using a variable **total** which keeps track of how many different letters there are in the current subset.

When we add a word to the subset, we increment the counter in the array **appeared**[letter] for each letter of the word, and increment the counter **total** only if 0 was written so far in **appeared**[letter] (if we've added a letter which hasn't appeared in the set so far). A similar procedure is done for removing a word from the set.

The complexity of this algorithm is $O(2^M * L)$, where **L** is the maximal length of the word. This solution is sufficient for 50% of points.

Another way of representing sets of letters in the computer's memory is using a bitmask. A bitmask is a series of ones and zeroes where the ith place is zero only if the ith element is present in the set and zero if it isn't. For example, the set {a, b, d, z, y} is represented with a bitmask 110...01011 (the numbers are enumerated from right to left). The union operator of the sets represented with bitmasks corresponds to the OR operator of their bitmasks.

The advantage of this implementation of sets in the memory is because the processor deals with bitwise operations really quickly, the complexity being $O(1)$.

The complexity of the algorithm when using this kind of implementation of set operations is $O(2^M)$.

For implementation details, consult the source code.

Necessary skills: finding all subsets, bitwise operations

Category: bit manipulation

COCI 2013/2014	Task KOCKICE
6th round, March 8th, 2014	Author: Antonio Jurić

Let's observe the function $f(x)$ where x represents the height of the middle column (the one with the minimal height) which tells us the number of minimal moves necessary to rearrange the piles in the described way. We can calculate this because then the minimal number of moves is **uniquely** determined: if a column has more bricks than necessary, we need to remove them and if it is missing bricks, we need to add new bricks.

We have two piles, but it is stated that in the end the corresponding columns have to be of equal heights. Then Mirko's pile is $f1(x) = |c_{m1} - x| + |c_{m2} - x| + \dots + |c_{mn} - x|$ and Slavko's pile is $f2(x) = |c_{s1} - x| + |c_{s2} - x| + \dots + |c_{sn} - x|$ where $c_{m1}, c_{m2}, \dots, c_{mn}$ i $c_{s1}, c_{s2}, \dots, c_{sn}$ are constants which depend on the height and position of the column.

If we observe the graph of the function $(f1 + f2)(x)$ we will notice that the function is decreasing at first, then increasing and it has only one global minimum.

It is easily noticed that this minimum is exactly the first point m for which the following holds: $(f1 + f2)(m) < (f1 + f2)(x + 1)$. In other words, it is the point where the function starts to increase. Also, we can notice that for every point to the left of m the following holds: $(f1 + f2)(x) > (f1 + f2)(x + 1)$ and for every point to the right of m : $(f1 + f2)(x) < (f1 + f2)(x + 1)$. This is why we can locate point m using binary search, by comparing the relation of function values of two adjacent points.

For implementation details, consult the source code.

Alternative approach (*Bruce Merry*). The first trick is to notice that the slightly odd target shape can be removed from the problem: simply

subtract $|i - N / 2|$ from element i of the inputs (the shortest target shape) and now solve for a flat target. You have $2N$ numbers and need to find the m such that $\sum(|a_i - m|)$ is minimised. This is just the median of the numbers (exercise: prove this, if you don't already know why). The median can be found in $O(N)$ time using `std::nth_element`; a binary search is also fast enough.

Necessary skills: mathematical analysis of the problem

Category: binary search

COCI 2013/2014	Task KRUŽNICE
6th round, March 8th, 2014	Author: Luka Kalinovčić, Anton Grbin

Since the circles do not intersect, we can look at them as intervals. More specifically, we observe their intersection with the x-axis.

Let us construct a relation **contains** such that A **contains** B if the interval B is inside of interval A, with allowed touching on the edges.

Let us construct a relation **directly_contains** such that A **directly_contains** B if A **contains** B and there is no other circle C for which holds that A **contains** C and C **contains** B.

Intuitively, A **directly_contains** B if B is one of the first smaller circles in A.

Because there is at most one circle that **directly_contains** an arbitrary circle, this relation is a tree.

Every circle will increase the number of regions by 1 or 2. In the case when a circle directly contains more other circles which touch along all its length, that circle is going to increase the solution by 2. In the contrary, by 1.

The tree of circles can be built using the sweep algorithm where an event is the beginning or the end of a circle. The events are processed by their x coordinates. As the structure of sweep algorithm we will use a stack which keeps track of the current parent circle and whether we have lined up every

directly contained circles next to each other. In the moment of popping a circle from the stack, the solution is increased by 2 if all the directly contained circles are lined up next to each other. In the contrary, the solution is increased by 1. The final solution needs to be incremented by 1 because it represents the whole region above all the circles.

The complexity of the algorithm is $O(N \log N)$ where N is the number of circles.

Necessary skills: stack data structure, sweep line traversal, tree

Category: sweep line

COCI 2013/2014	Task HASH
6th round, March 8th, 2014	Authors: Anton Grbin

This requires a "meet-in-the-middle" attack: it's a common idea in crypto, but is also useful in a number of exponential-time competition problems. For $N = 10$, we can't consider all 26^{10} words separately. However, we can consider 26^5 five-letter prefixes and 26^5 five-letter suffixes, and then figure out how to match them up. For each prefix, we can compute its hash, and store a table for how frequently each such hash occurs. For each suffix, we can compute what the hash of the prefix would need to be for the final hash to be K , and then use the table to find the number of prefixes that match this suffix. To compute what the hash would need to be, we work backwards: assume the final hash is K , and then remove one letter at a time from the end.

Let us observe an iteration of the stated hash function with the assumption that the current value of hash is S and the ordinal number of the next letter is x .

$$S' = ((S * 33) \text{ xor } x) \% \text{MOD}$$

We will try to get the value of the previous state S when we know the current state S' and the ordinal number of the letter which got us to the state x .

Given the fact that we can look at the operation remainder when dividing with a power of two (2^M) as discarding any bits after the M^{th} , it is easily noticed that the following holds:

$$(A \text{ xor } B) \% \text{MOD} = (A \% \text{MOD}) \text{ xor } (B \% \text{MOD})$$

Therefore,

$$S' = ((S * 33) \% \text{MOD}) \text{ xor } (x \% \text{MOD})$$

because the bitwise XOR is inverse to itself and x will always be smaller than MOD, this is also true:

$$S' \text{ xor } x = (S * 33) \% \text{MOD}$$

The modular multiplicative inverse of 33 in field of size MOD will exist if 33 and MOD are relatively prime. Since MOD is a power of two, this condition is completed. The modular inverse can be obtained using extended Euclid's algorithm, fast exponentiation or with brute force because M is small enough.

Let us display the complete inverse relation of the hash function:

$$(S' \text{ xor } x) * \text{inv}(33, \text{MOD}) = S$$

Having inverse relation in place, meet in the middle attack is done which gives us time complexity of $O(26^{(N/2)})$.

Necessary skills: extended Euclid's algorithm or Euler's theorem and fast exponentiation, bitwise operations, advanced recursion, breaking the problem into smaller parts

Category: number theory, meet-in-the-middle

COCI 2013/2014	Task GRAŠKRIŽJA
6th round, March 8th, 2014	Authors: Luka Kalinovčić, Adrian Satja Kurdija

Let us sort the given traffic lights by their x-coordinate. Let \mathbf{x}' be the middle (median) x-coordinate in that array. Let \mathbf{A} be a set of given traffic lights to the left of \mathbf{x}' and \mathbf{B} to the right.

We will construct a harmless path between each pair of traffic lights \mathbf{a} , \mathbf{b} such that \mathbf{a} is from the set \mathbf{A} and \mathbf{b} is from the set \mathbf{B} . How? By adding new traffic lights to the locations $(\mathbf{x}', \mathbf{y})$ for each y-coordinate \mathbf{y} from the sets \mathbf{A} and \mathbf{B} . Now for traffic lights \mathbf{a} and \mathbf{b} we have a harmless way $(\mathbf{x}_a, \mathbf{y}_a) \rightarrow (\mathbf{x}', \mathbf{y}_a) \rightarrow (\mathbf{x}', \mathbf{y}_b) \rightarrow (\mathbf{x}_b, \mathbf{y}_b)$.

Now all we need to do is connect the traffic lights within the set \mathbf{A} with one another, as well as those within the set \mathbf{B} . We do this by recursively repeating the described procedure, specifically for set \mathbf{A} and specifically for set \mathbf{B} . This way of thinking is called *divide and conquer*.

A little bit of thinking is needed to be sure that the new traffic lights don't generate any new dangerous paths, but the implementation is reasonably simple.

What is the number of additional traffic lights? Given the fact that we divide the set into two parts, the maximal depth of recursion is $O(\lg \mathbf{N})$. Let us observe an initial traffic light at the location (\mathbf{x}, \mathbf{y}) . Worst case scenario, at every depth of the recursion, it will be included in a set and there it will generate a new traffic light $(\mathbf{x}', \mathbf{y})$. Therefore, one initial traffic light generates $O(\lg \mathbf{N})$ new traffic lights, which gives us a total of $O(\mathbf{N} \lg \mathbf{N})$ new traffic lights. With careful implementation, the exact number turns out to be less than 700 00.

Necessary skills: sorting, recursion, *divide and conquer* principle

Category: ad hoc