COCI 2017/2018

Round #2, November 4th, 2017

Solutions

Task Košnja	**Author: Stjepan Požgaj**

Let N and M denote the number of rows and columns of the matrix that represent the lawn, respectively.

Let’s assume that N ≤ M. One of the optimal ways for Mirko to visit each field while making the minimal number of turns is that he starts from the upper left field facing right, visits the entire first row, goes to the second row, turns right, visits the entire second row, goes to the third row, and so on.
The described way takes exactly 2 · (N - 1) turns.

In the case when M ≤ N, he needs to visit the fields in the same way, but now he needs to visit the columns in order. This takes exactly 2 · (M - 1) turns.
Therefore, the solution to the problem is 2 · min(N - 1, M - 1).

The proof that the described way is indeed the optimal way is left as an exercise to the reader.

Solution (written in Python 3.x):

```
k = int(input())
for i in range(k):
    n, m = map(int, input().split())
    print(2 * min(n - 1, m - 1))
```

Necessary skills: variable assignment

Category: ad-hoc

Task Zigzag	**Author: Nikola Dmitrović**

For each given letter, we must find a word from the list that starts with that letter. That word must be used the least amount of times. This algorithm is clear from the task. Given the constraints for the input data, it is necessary to be careful with the solution implementation, since the naive solution has a problem with the time limit.

A helpful trick is to, for each letter, sort the words starting with that letter, and iterate over them in a circular manner using a pointer. It is obvious that the first chosen word will be the least used one, next time we will choose the second word, and when we reach the final words, we will again return to the first word.

You can find two proposed implementations (the real one and the naive one for 60% of total points) in the official solutions.

Necessary skills: list (dictionary), sorting algorithm

Category: ad hoc

Task Doktor	**Author: Daniel Paleka**

Let the *value* of the contiguous subsequence (from now on, just subsequence) be equal to the difference between the number of fixed points created by rotating that subsequence and the number of fixed points that disappear by rotating that subsequence. It is clear that we want to choose the subsequence of the maximal possible value.

We can see that a card with the number x that is not a fixed point becomes a fixed point by rotating a subsequence only if the *center* of the subsequence (the middle member if the subsequence is of odd length, and the point between two middle members if the subsequence is of even length) is equidistant to that card and the card in the x^{th} index or, in other words, the position where the card with the number x should be.
If the previous condition is met, a card will become a fixed point if and only if it is contained in the subsequence.

Using the observations above, we can try to design a solution: for each possible center of the subsequence, iterate over all subsequences with that center from the smallest to the largest member. We will always spread by exactly 2 cards, so in each step, we can check in constant time if the value of the subsequence has changed, and for how much. This solution calculates the values of all subsequences in the asymptotic complexity of O(N^2).

In order to improve the previous solution, we will take advantage of the fact that we are searching for a subsequence with the maximal value, and not the values of all subsequences. In fact, the value of the subsequence when we iterate over all subsequences with the same center in the previous solution only increases at positions where a new fixed point is created. Therefore, it is sufficient to observe, for each center, only the subsequences with card x at the first card, and end at position x, or vice versa.

The previous passage motivates the following solution: for each card, determine the center that can make it a fixed point, and store it in a list of cards for that center. Then, for each center, we sort the associated list by the distance between the card and the center, and iterate over the list and increase the number of newly created fixed points by 1 for each new element of the list.

In order to calculate the value of each of the given subsequences, we also need to find out the number of fixed points that disappear by reversing a subsequence. We can do this in constant complexity if we previously calculate the number of fixed points in all prefixes of the sequence of cards. The asymptotic complexity of the solution by parts is O(N) to calculate the fixed points in all prefixes, O(N) for associating the cards to the centers, O(N log N) in the worst case for sorting the lists of all the centers, and amortized O(N) for iterating over all considered subsequences, because the total number of elements in all the lists is equal to the number of cards, which is N. The total asymptotic complexity is therefore O(N log N), which is sufficient to get all points.

Additionally, it is not too difficult to solve the task in O(N), but this is left as an exercise to the reader.
For any clarifications, consult the official solution.

Necessary skills: prefix sums

Category: ad-hoc

Task San	**Author: Tonko Sabolčec**

To begin with, let's observe a simple version of the task where the skyscraper heights are sorted ascendingly. Notice that then the heights don't have a role, it is sufficient to find the number of different subsets of skyscrapers such that the sum of the coins on them is larger than or equal to K. A naive solution that tests every possibility is of complexity $O(2^N)$. We can halve the array of skyscrapers (let the first half contain $N/2$ skyscrapers, and the other the rest). For the left half, we create a list L with all the sums of coins that can be obtained by testing all subsets of skyscrapers in the left half. For example, if the left half has 3 skyscrapers that contain on their roof, respectively, 1, 2 and 3 coins, when we will add the following values to list L: (0, 1, 2, 3, 3, 4, 5, 6) - notice that the number 3 appears 2 times because there are 2 different ways which we can obtain that number of coins. Analogously, we will generate a list R for the right half. Now the task is broken down to the following: for each number in list L, find how many numbers there are in list R such that their sum is less than or equal to K. If we sort the numbers in the list R, then the answer to that query can be found using binary search. Lists L and R consist of $O(2^{N/2})$ elements, so the total complexity of this algorithm is $O(N \cdot 2^{N/2})$.

The idea for solving the original task is similar. The only problem are the heights of the visited skyscrapers that must be sorted ascendingly. For each valid subset of skyscrapers in the left half, we will add to list L the pair (s_L, h_L) where s_L denotes the sum of coins in the subset of skyscrapers, and h_L the height of the **last** skyscraper in the subset. For the right half, we will generate a similar list, the only difference being that we will add to list R the pair (s_R, h_R) where h_R represents the height of the **first** skyscraper in the subset. Now the task can be solved by counting, for each pair (s_L, h_L) from list L, the number of pairs (s_R, h_R) in list R such that it holds $s_L + s_R \geq K$ i $h_R \geq h_L$. We can solve this by grouping the sums in list R by values h_R, and for each pair (s_L, h_R) from the left half, we iterate over all groups for which $h_R \geq h_L$, and determine the number of valid subsets by using binary search in the same way as it was explained for the simpler version.

This approach, where we divide the array into two parts and test every combination for each part, is known as *meet-in-the-middle*.

Necessary skills: testing every combination, binary search

Category: *meet-in-the-middle*

Task Usmjeri	**Author: Adrian Beker**

To begin with, we'll make node 1 the root of the tree. For node u that is not a root, let $p(u)$ denote its parent. Instead of directing the edges, let's imagine we're colouring them in two colours so that one colour denotes that the edge is directed from the child to the parent, and the other from the parent to the child. Let's observe two nodes a and b. Let node c be their lowest common ancestor (LCA). Notice that there is a path from a to b or from b to a if and only if the following three conditions are met:

- All edges on the path from a to c are of the same colour
- All edges on the path from b to c are of the same colour
- If c is different from a and b, then edges $(a, p(a))$ i $(b, p(b))$ are of different colours

Let's now construct a graph where the nodes denote the edges of the given tree in the following way. For each given pair (a_i, b_i) with LCA-om c_i, we will add the following edges to the graph:

- The nodes that represent the adjacent edges on the path from a_i to c_i will be connected with a blue edge
- The nodes that represent the adjacent edges on the path from b_i to c_i will be connected with a blue edge
- If c_i is different from a_i and b_i, the nodes that represent edges $(a_i, p(a_i))$, $(b_i, p(b_i))$ will be connected with a red edge

Now we want to know the number of possible ways to colour the nodes of this graph in two colours so that the nodes connected with a blue edge are of the same colour, and the ones connected with a red edge are of different colours. We can see that the connected components of the graph are mutually independent, so the solution is equal to the product of solutions by individual components. Furthermore, we can see that the colour of a node uniquely identifies the colours of all the other nodes in its component. Additionally, if we have a valid colouring scheme, it stays valid if we change the colour of all the nodes. This means that each component has 0 or 2 valid colouring schemes, i.e. we only need to determine whether such a colouring scheme exists. We can do this with a DFS algorithm, starting from an arbitrary node and spread recursively, changing the colour when we reach a red edge and taking care of possible colouring contradictions. If there are no contradictions in any of the components, the final solution will be 2^k, where k is the number of components, otherwise it is 0.

Now we are only left with constructing the aforementioned graph. A naive construction is of the complexity O(M · N) and wasn't fast enough for all the points. Notice that for each node *x* of the tree, except the root and its children, we need to determine whether the nodes that represent edges *(x, p(x))* i *(p(x),p(p(x)))* are connected with a blue edge. We can do this by using a recursive function *connect(x)* that returns the minimal depth of a node such that we have added blue edges on the path from that node to a node in the subtree of *x*. If the value of *connect(x)* is smaller than the depth of *p(x)*, then we add the blue edge, otherwise we don't. The value of *connect(x)* is calculated by taking into account its values for the children of *x* and also *high[x]* - the minimal depth of a node such that we have added blue edges on the path from that node to *x*. We can get these values by, for each given pair (a_i, b_i) with LCA c_i, we update the values $high[a_i]$ and $high[b_i]$ with the depth of node c_i if it is smaller than the values $high[a_i]$, or $high[b_i]$, so far.

The total time complexity of this solution is O((M + N) log N), and memory O(N log N + M).
For implementation details, consult the official solution.

Necessary skills: DFS algorithm, lowest common ancestor (LCA)

Category: ad-hoc, graphs

Task Garaža	**Author: Tonko Sabolčec**

Let's observe any array of natural numbers A and the GCD (greatest common divisor) of each prefix in the array, P. For example:

i	1	2	3	4	5	6	7	8
$\boldsymbol{A_i}$	216	144	96	96	120	560	9	8
$\boldsymbol{P_i}$	216	72	24	24	24	8	1	1

Let's observe the two adjacent values P_i and P_{i+1}. The following properties hold:

- P_i is divisible by P_{i+1}.
- $P_i \geq P_{i+1}$. In the case that P_i != P_{i+1}, then it holds $P_i \geq 2 \cdot P_{i+1}$, which is why the value P_i will change $\log_2 10^9$ times at most.

We conclude that array *P* can be written in a different way, so that we group all the prefixes with the same GCD, as an array of at most O(log 10^9) pairs of numbers (*g*, *d*) where *g* represents one of the values P_i, and *d* represent the number of appearances of that value in array *P*. For example, the array *P* from the above example can be written as: {(216, 1), (72, 1), (24, 3), (8, 1), (1, 2)}. Analogously, we can write the GCD of all suffixes of array *A*.

We will solve this task using a segment tree where we will store the following data for each interval:

- GCD of all prefixes *P* (in the above mentioned form, so as an array of pairs (*g*, *d*)),
- GCD of all suffixes *S* (also in the above mentioned form), and
- The number of interesting contiguous subsequences in that interval, *count*.

In order to apply the *tournament* tree data structure on this task, we need to define how we can determine new data for the union of intervals, based on the aforementioned data for two adjacent intervals. Let there be two **adjacent** intervals *L* and *R*. Based on the GCD of all prefixes *L.P* and *R.P*, let's try to determine the GCD of all prefixes *C.P* (where *C* represents the union of intervals *L* and *R*). Array *C.P* is calculated in the following way:

$g(C.P_i) = g(L.P_i)$ for $1 <= i <= len(L.P)$
$g(C.P_i) = \text{NZD}(g(L.P_{len(L.P)}), g(R.P_{i - len(L.P)}))$, for $len(L.P) + 1 <= i <= \text{len}(L.P) + \text{len}(R.P)$

We can notice that in some cases it will be possible to additionally group some elements in the array *C.P* by values g(*C.P*), and that the length of the total array is never going to be larger than $\log_2 10^9$. The complexity of determining this array is $O(\log 10^9)$, and analogously, we can determine the array C.*S* that represents the GCD of all suffixes of interval *C*.

All that is left to determine is how we can calculate the number of interesting subsequences in the interval *C* (*C.count*). These values will be equal to the sum of values *L.count, R.count* and the number of interesting subsequences that are in both intervals *L* and *R*. Let *L* correspond to the interval [*l*, *m*], and *R* to the interval [*m*+1, *r*]. Let's observe the smallest *p* from the interval *L*, for which a *q* exists from the interval *R* such that it holds that the subsequence [*p*, *q*] is interesting, and the subsequence [*p*, *q*+1] is not. The GCD of the subsequence [*p*, *q*] can be calculated as the largest common divisor of the GCD-suffix until position *p* in array *L.S* and the GCD-prefix until position *q* in array *R.P*. However, we should notice one more thing: as we increase *p*, so will the value *q* either remain the same or increase by a positive number. Therefore, the number of interesting subsequences contained in both intervals *L* and *R* can be calculated using the two-pointer technique, *p* that will iterate over array *L.S* and *q* that will iterate over array *R.P*. The complexity of this approach is O(len(*L.S*) + len(*R.P*)), i.e. $O(\log 10^9)$. To consult the details of joining two adjacent intervals in the tournament tree, consult the official solution.

The total complexity of the joining is $O(\log 10^9)$, so the total complexity of the algorithm is $O((N + Q) \log N \log 10^9)$.

Necessary skills: segment tree, calculating the greatest common divisor

Category: data structures, mathematics