CROATIAN OPEN COMPETITION IN INFORMATICS

7th round, March 7th, 2015

solutions

Task TETA	**Author:** Marin Tomić

The prices of the meals that are not a part of a menu can be just added to the final solution and removed from the tray. After that, we take one piece of each meal that are part of a menu from the tray. If the sum of their prices is larger than the menu price, we will pay the menu price and remove the meals from the tray. Otherwise, we will pay for each meal separately and remove them from the tray.

Consult the source code for details.

Necessary skills: while-loop
Category: greedy algorithms

Task KRIZA	**Author:** Ivan Paljak

A naive approach comes down to simulating each attempt at unlocking of the door. Given the fact that Sisyphus has to move at most **N** keys to the other side when facing each door, having visited **K** doors in total, the complexity of such algorithm is **O(N*K)** which is enough for 40% of points on this task.

We need to notice that, when facing the **i**th door, the rightmost key on the pendant unlocks door **i-1** and that there is a constant number of keys on the pendant between keys marked with **i** and **j**. Then it's **O(1)** to find out how many misplaced keys Sisyphus has placed in the door with a fixed label **i**. This leads us to a solution of the complexity **O(K)** which is enough for 60% of points on this task.

Finally, we should notice that different states of keys on the pendant are necessarily a part of a cycle of length **N**. If we fill out an array **A[]**, where **A[i]** tells us how many misplaced keys Sisyphus has placed starting from the first to the **i**th door (for **i** <= **N**), we can answer the given question in **O(1)**. Filling out array **A[]** is done in the complexity **O(N)**, which is enough for 100% of points on this task.

Additionally, we need to be careful about the first unlocking. In other words, the transition from the initial state to the state after the first door that could, but doesn't have to, be a part of the cycle described in the previous paragraph. Also, we need to notice that the solution can overflow the 32-bit integer.

Necessary skills: cycles
Category: ad-hoc

Task ACM	**Author:** Adrian Satja Kurdija

For each of the six possible permutations of the team members (first, second, third) the problem is solved using dynamic programming. The state consists of the task we are currently on and the team member (first, second or third) whom the previous task has been assigned to. When making a transition, we choose if we want to assign the current task to the same or the next team member. In the attached source code, dynamic programming is implemented recursively, using memoization. Of course, an iterative implementation is also possible.

Necessary skills: dynamic programming
Category: dynamics

Task JANJE	**Author:** Ivan Paljak

Firstly, let us notice that each picture can be represented as a graph where the number of nodes corresponds to the number of areas that need to be filled out, and two nodes are connected if the corresponding areas are adjacent. Each node has to be paired with one of the **K** colors in a way that two adjacent nodes are colored differently and three different colors are used at most.

A significant part of the official test cases contained a small enough **K** and an image with small enough number of nodes so that naive implementations of the aforementioned procedure were enough for 50% of points on this task.

Let **j(n), d(n)** and **t(n)** denote the number of ways the **n**$^{\mathbf{th}}$ image can be colored **exactly** with one, **exactly** with two and **exactly** with three different colors, respectively. Then the final solution is equal to **j(n)*K + d(n)*K*(K - 1)/2 + t(n)*K*(K-1)*(K-2)/6.** In all pictures, it holds **j(n)** = 0 so we can ignore coloring using only one color. If the picture can be colored using two colors, then **d(n)** = 2. We are still left with finding **t(n)**.

All pictures besides the caterpillar, pyramid and trampoline are represented by a graph with a small enough number of nodes **v** so determining the number **t(n)** in the complexity of **O(**3^**v**) is fast enough. This approach was enough for 75 points on this task.

It needs to be pointed out that it was possible to calculate the numbers **j(n)**, **d(n)** and **t(n)** for each picture by hand. A lot of competitors approached the task in this way, but often made a mistake during calculation.

The mentioned three images (caterpillar, pyramid and trampoline) should be analyzed into more detail. For example, we can notice that, by filling out the second row of the pyramid, we will unambiguously determine the rest of the pyramid. We leave similar observations as a practise to the reader.

Necessary skills: graphs, combinatorics
Category: ad-hoc

Task PROSJEK	**Author: Ivan Katanić**

It is a difficult task to directly answer what the maximal possible average is, so let's instead try to answer the question of whether the maximal possible average is larger than or equal to some value P. If we succeed in this, then we can solve the task using binary search over value P.

This task also seems difficult, but let's notice this: if we subtract X from every number in the array, then the average of all subsequences is reduced by exactly X. Now we can subtract P from all the numbers in the array and see if a subsequence exists with an average larger than or equal to 0. This task is simpler because it is equivalent to the question whether a subsequence exists with the sum larger than or equal to zero.

If we calculate the prefix sum S of the input array, then the sum of a subsequence [a, b] is equal to S[b] - S[a-1]. For every possible b, we can ask ourselves whether such a exists that S[b] - S[a-1] >= 0, or S[b] => S[a-1]. Also, we mustn't forget the condition for the minimal subsequence length, b-a+1 >= K or a <= b+1-K.

Iterating over boundary value b from left to right, we can store the minimal value S[a-1] seen so far in a special variable and check if it meets the requirement for current b. If it does, we have found a subsequence with an average larger than or equal to P.

Each time we increment variable b by 1, exactly one new value comes into consideration for a and it is a = b-K+1. By doing so, we check if S[b-K] is smaller than the current minimal value and update it if it is. For details, consult the source code.

Necessary skills: binary search, mathematics
Category: miscellaneous

Task POLICE	**Author:** Mislav Bradač

The first thing we need to check is whether there is an empty place on any shelf. In case there isn't, we check whether all the books are in the right place. If they're not, no solution exists and we output -1, otherwise the solution is 0.

Let's see how to solve the test cases worth 50% of points in which for each book it holds that it is on the same shelf both in the beginning and in the end. We divide the shelves into three categories:

1. the shelf with an empty place in the beginning
    a. **k** - the number of books on the shelf
    b. **lis** - the longest subsequence of books that are in a good relative order in the beginning
    c. each book that isn't in **lis**, we need to lift in order to arrange the books on that shelf, in other words, we need **k - lis** lifts
    d. let's notice that by pushing books we can free a position on the shelf where we need to place a book before we lift it
    e. **lis** is easily found in **O(k log k)** using the well known algorithm for finding the longest increasing subsequence
2. the shelf is full, but all the books are in their place
    a. we don't need a single lift for shelves of this type
3. the shelf is full, but there is a book that is not in its place
    a. we move one book that isn't in **lis** over to an empty space on another shelf
    b. we arrange the other books on the shelf in the right order, not lifting the books contained in **lis**
    c. we put the book we moved to another shelf to its right place
    d. for this type of shelf, we need **k - lis + 1** lifts

As for test cases that have books that aren't on the same shelf in the beginning and in the end, we will split those shelves into two categories:

1. the shelves with books that are on that shelf both in the beginning and in the end and every book that is there in the end was also there in the beginning
    a. we solve these shelves in the same way we solved the shelves in test cases worth 50% of total points
2. other shelves

Now we are left with solving the second category of shelves. Let's build an undirected graph where the nodes are the shelves. Two nodes are connected if there is a book that is located on the shelf represented by the first node in the beginning, and in the end it's located on the shelf represented by the second node.

Let's solve each component of this graph separately and introduce two new cases:

1. all shelves in the component are full in the beginning
    a. let's now build a directed graph of this component so that we direct the edge from the node where the book is coming to (the node that the book is

located in the end) to the node from where the book came from (the node that the book is located in the beginning)

b. notice that each node will have the same indegree and outdegree and that we can build a Euler cycle
c. we place a book from node **A** that is not on the same shelf in the beginning and in the end to an empty place on a shelf that has an available place in that moment
d. we start from node **A** and move in a Euler cycle, when we go from node **A** to node **B**, we move the book from node **B** to node **A**
e. during the Euler cycle traversal, in the moment when a shelf has an empty place, we place the books on that shelf that are there in the beginning and in the end in a good relative order - we do this in the way described in the first category of the solution for 50% of total points
f. **sum_k** - the number of books on shelves in that component
g. **sum_lis** - the sum of lis of individual shelves in that component
h. we need **sum_k - sum_lis + 1** lifts to solve this component
i. notice that we don't need to build a Euler cycle because a reconstruction of the solution is not required

2. there is a shelf in the component that is not completely full in the beginning
   a. pick a node with a larger indegree than outdegree and a node that has a larger outdegree than indegree and add a dummy edge between these two nodes
   b. repeat this procedure until these type of nodes exist
   c. build a Euler cycle and start with the node with an empty place in the beginning, if we pass through a right edge, we move the book, if we pass through a dummy edge, nothing is moved
   d. it is important to notice that there will always be an empty place in the node where we are located in, even if we entered that node using a dummy edge; this holds because of the fact that if a shelf has a larger outdegree than indegree, then the number of available places in the beginning on that shelf is larger than or equal to the difference of outdegree and indegree
   e. we solve the books on the same shelf in the beginning and in the end in the same way as in the previous case
   f. we need **sum_k - sum_lis** lifts to solve this component

The total complexity of this algorithm is O(N * M * log M).

Necessary skills: dfs, longest increasing subsequence
Category: combinatorics, graphs