COCI 2016/2017

Round #4, December 17th, 2016

Solutions

Task Bridž	**Author: Branimir Filipović**

In order to solve this task, we needed to count the number of appearances of characters ‘A’, ‘K’, ‘Q’, and ‘J’ in all cards K_i together.

The solution of the task R is given in the following formula:

$$R = 4 * A + 3 * K + 2 * Q + 1 * J$$

Pseudocode (written in Python 3.x):

```
n = int(input())
r = 0

for i in range(n):
  s = input()

  for j in s:
    if j == 'A':
      r += 4
    elif j == 'K':
      r += 3
    elif j == 'Q':
      r += 2
    elif j == 'J':
      r += 1

print (r)
```

Necessary skills: for loop, strings
Category: ad-hoc

Task Kartomat	**Author: Nikola Dmitrović**

First, we will input the given destinations in an array, and load the first couple of characters of the chosen destination into a special variable. For each destination that is already given and for each destination that we could have chosen, we check if it starts with this series of characters. If it does, we determine the letter than can be chosen in the next step, or, more precisely, the letter that follows after the given series of characters. Such obtained letter is appended to the previously chosen letters.

In the end, we replace each letter from the string “***ABCDEFGHIJKLMNOPQRSTUVWXYZ***” for which we haven’t determined that it can be chosen in the next step with the character ‘*’. We are left with carefully printing the letters of the alphabet in blocks of 8 letters.

Source code (written in Python 3.x):

```
N = int(input())

L = []
for i in range(N):
      L += [input()]

destination = input()

first_next_letters = ''
d = len(destination)

for i in range(N):
      if L[i][:d] == destination:
      first_next_letters += L[i][d]

keyboard = '***ABCDEFGHIJKLMNOPQRSTUVWXYZ***'

for i in keyboard:
      if i not in first_next_letters:
      keyboard = keyboard.replace(i,'*')

print(keyboard[0:8],keyboard[8:16],keyboard[16:24],keyboard[24:32]
, sep = '\n')
```

Necessary skills: strings
Category: ad-hoc

Task Kas	**Author: Ivan Paljak**

The entire task comes down to correctly distributing the banknotes as described in the first paragraph of the task. So, we need to distribute **some** of *N* banknotes into two parts so that the sum of the money in each of the two parts is equal, and the total sum of the money of the unused banknotes is the smallest possible.

Using a naive algorithm, we could have tried out every possible combination and get 50% of total points. Since each banknote can end up with either Kile or Pogi or nobody, we can conclude that the time complexity of such algorithm is O(3^n).

Similar to the previous thought process, let’s imagine that we iterate respectively over the banknotes and try assigning them to Kile, Pogi, or nobody. In any step of such algorithm we

should know which banknote we are currently processing and how much money have Kile and Pogi collected so far. Let $f(k,a,b)$ be a function that returns the largest possible sum of money that Kile and Pogi can collect if after $(k-1)$ distributed banknotes Kile has a kn, and Pogi has b kn.
Evidently, $f(k,\ a,\ b)\ =\ max\{f(k+1,\ a,\ b), f(k+1,\ a+c[k],\ b), f(k+1,\ a,\ b+c[k])\}$ where $c[k]$ denotes the value of the k^th banknote. Of course, $f(n+1,\ a,\ b)$ is 0 if a is different than b, or a in the contrary. If we implement such a solution using the technique of dynamic programming, we have constructed an algorithm of the complexity $O(ns^2)$, where s represents the sum of all banknotes.

To obtain all points, it was necessary to notice that it is sufficient to keep track in the state the absolute value of the difference of the amount of money Kile and Pogi have collected so far. We transition from state $f(k,\ diff)$ to states $f(k+1,\ diff)$, $f(k+1,\ diff+c[k])$ and $f(k,\ |diff-\ c[k]|)$ that represent, respectively, skipping of a banknote, assigning a banknote to the person currently having more money, and assigning a banknote to the person currently having less money. Since we have $O(ns)$ states, and the transition if constant, we can conclude that the algorithm is of the complexity $O(ns)$, which is sufficient to obtain all points.

Even though this wasn't a requirement in the task, we advise you to think of an algorithm to solve this task with a memory limitation of 32 MB.

Necessary skills: dynamic programming, state optimization
Category: dynamic programming

Task Rekonstruiraj	**Author: Adrian Satja Kurdija**

Knowing that the numbers in the input have at most five decimal places, we transform them to positive integers by multiplying them with 100 000. Mirko's unknown numbers (also multiplied with 100 000) are obviously some of the divisors of the obtained numbers.

How to efficiently find divisors of X? We can do this by iterating over all of its potential divisors *d* from 2 to sqrt(X) and check whether X and X/*d* are divisors of X. For each number, we must find the divisors that are "good", in other words, if they belong to the given set of Mirko's numbers. A divisor is "good" if all of its multiples from a given interval are located in the required set, which is possible to check efficiently.

After finding the set of all the "good" divisors, we must choose the minimal subset where the multiples "cover" all given numbers. This is a variant of the problem known as *set cover* where you need to choose the minimal number of given subsets that cover the entire given set. *Set cover* is an NP-complete problem, which means that an efficient solution does not exist. Luckily, solutions exist that are correct in "most cases". The task author excuses himself for thinking that a greedy algorithm, one that always chooses the smallest good divisor that covers a yet uncovered number, necessarily produces the minimal final set, but there is an example for which this doesn't hold (as an exercise, find the example). A greedy algorithm is one of the possible *heuristics* - algorithms that are very successful, but sometimes produce a suboptimal solution. The test data was such that a greedy algorithm or any other reasonable heuristic performs correctly, scoring all of the points.

Necessary skills: decimal numbers, divisor lookup
Category: number theory, heuristics

Task Rima	**Author: Domagoj Bradač**

We will reverse the given words and add them to a prefix tree. Two words rhyme if and only if the node that represents one of them is a parent of the node that represents the other word or if the two nodes have a common parent.

Let's observe a sequence where each two adjacent words rhyme. The sequence begins at a node in the tree and makes a couple of steps upwards or to the side. In other words, towards the parent or between the nodes with a common parent. Once it makes a step downwards, towards the child, it can only make steps to the side or downwards.

Now we can solve the task using dynamic programming. For each node, we calculate the largest number of nodes in its subtree that we can traverse moving only upwards or downwards (it doesn't make a difference), and to the side. For each node, we can assume that it is the highest node in the sequence, and find the two longest paths between its children. We must be careful to count all nodes with a common parent. For details, consult the official solution.

The total complexity is O(S), where S is the sum of lenghts of all strings.

Necessary skills: trie
Category: strings, dynamic programming

Task Osmosmjerka	**Author: Adrian Satja Kurdija**

Let's fix a block of the crossword. We can assume that we always choose the initial letter from that block. Probability is defined as the number of favorable selections divided by the number of possible selections. The number of possible selections of two words is equal to the square of the number of possible selections of one word, and it is equal to the number of possibilities for the initial field (M times N) multiplied with the number of possible directions (8).

A bigger issue is calculating the number of favorable selections, the ones that provide two equal words. If a word R appears in X possible places, then we can read it once in X ways, and twice in X*X ways. Therefore, it is necessary to add up the squares of these numbers X, or, more precisely, to identify different possible words and know in how many ways each of them can be read.

We do this by using a *hash* function that converts a string to a number for easier comparison. Hashing is problematic here because of the size of K (the length of the word). In the case where M = N, we can notice that all words are periodic with regards to period N, so it is sufficient to look at length K % N, and all such hashes can be calculated fast enough using the *rolling hash* approach. This solution was worth 100 points.

In the general case, we can, as an auxiliary step, calculate all hashes which length is a power of 2, so that the hash of length 2^(i+1) is obtained by combining two hashes of length 2^i. Then we write K in binary as the sum of powers of 2. This way, we calculate the hash of each possible word as a combination of a couple of already calculated hashes. We can determine the number of times each of them appears by sorting the set of hashes of possible words and therefore obtain the required probability.

Necessary skills: hash, probability definition
Category: strings