COCI 2016/2017

Round #7, March 4th, 2017

Solutions

Task Baza	**Author: Branimir Filipović**

In order to successfully solve this task, we first need to input and store the entire database, the matrix of dimensions NxM.

Then, for each query, we need to iterate over each row of the matrix and count how many rows of the database correspond to the query in the way described in the task.

Pseudocode (written in Python 3.x):

```
def checker(row, query):
  n = len(row)

  for i in range(n):
    if ((query[i] != -1) and (row[i] != query[i])):
      return False

  return True

n, m = map(int, input().split())
base = []

for i in range(n):
  row = list(map(int, input().split()))
  base.append(row)

q = int(input())

for i in range(q):
  query = list(map(int, input().split()))

  z = 0
  for j in range(n):
    if (checker(base[j], query)):
      z += 1

  print (z)
```

Necessary skills: *for* loop, matrices
Category: ad-hoc

Task Uzastopni	**Author: Adrian Satja Kurdija**

Trying out every possibility is too slow because of the size of the number N. It is important to notice that the number of possible summands cannot be very large. More precisely, the number of summands will not be greater than the square root of 2N, because otherwise their sum would exceed N (see this for yourself). Therefore, we have enough time to iterate over all possible K (where K is the number of consecutive summands), and for each of them search for the corresponding summands, i.e., the first and the last number -- we can do this either mathematically (the formula is easily derived), or by binary search.

Necessary skills: mathematical problem analysis
Category: ad hoc

Task: Igra	**Author: Nikola Herceg**

We will construct a word letter by letter, starting from the first one, in a way that we try to place the lexicographically smallest letter to the current position. After we choose a letter, we need to check whether the remaining letters can be placed so that none of them match the same letter in Mirko's word (MR). Therefore, we are not interested in the order of these letters, but only if they can be arranged somehow.

Let a, b and c denote the number of remaining letters 'a', 'b' and 'c', respectively, and A, B and C the number of remaining letters 'a', 'b' and 'c' in MR. Let's try to place k letters 'a' to the positions of letters 'b' in MR, i.e., reduce a and B by k. The remaining letters 'a' we place to the positions of letters 'c' in MR, i.e., reduce C by a and set a to 0. We set the remaining letters 'b' in MR to letters 'c', i.e., reduce c by B and set B to 0. Now, we need to place the remaining letters 'c' to the positions of letters 'a' in MR, i.e., reduce A by c and set c to 0. If we couldn't have made any of the previous moves, that means no solution exists for the chosen k. If we could have, we are left with checking whether letters 'b' can be placed to the positions of letters 'a' and 'c' in MR. This will hold if A + C = b. In that case, the letters can be placed for this k and the check is done.

We will test this out for each k between 0 and B. If the check didn't succeed for any k, then it is impossible to place the remaining letters so that none of them match the same letter in Mirko's word, otherwise we can. This can needs to be done for each of n letters, so the total complexity is O(n^2). We leave the linear solution as an exercise for the reader.

Necessary skills: strings
Category: greedy

Task Poklon	**Author: Ivan Paljak**

Let S be the total mass of the weights after balancing the scale. Then the total mass of the weights on the beams of the large scale is equal to S/2, and the sum of the weights on the beams of these scales is equal to S/4, and so on. Generally, a beam of a scale at depth *r* holds a load of mass S/2^r.

Since some beams (at depth r) already have a weight of mass m, additionally the following inequality must hold: S/2^r >= m. Finally, we have S = max {m * 2^r} for each weight of mass m, that is at depth r.

We are left with efficiently comparing the numbers of the form a * 2^b, where a and b are small enough to fit in a 32-bit data type. Since these numbers in binary notation have at most 32 ones (multiplication with 2^b corresponds to left shift b times), it is sufficient to just linearly iterate over the ones in the binary notation of these numbers.

For implementation details, consult the official solution.

Necessary skills: dfs, trees, mathematical problem analysis
Category: ad-hoc, graphs

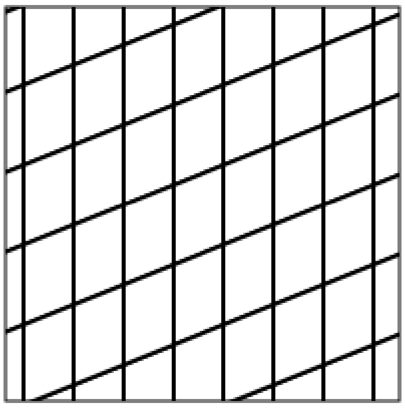
Task Paralelogrami	**Author: Mislav Balunović**

If all points are colinear, then we obviously can't make a single move, so the solution does not exist.

First, notice that the operation of mapping the point C over points A and B can be written in a simple algebraic form, instead of a geometric one:

(C_x, C_y) is mapped over (A_x, A_y) and (B_x, B_y) into point $(A_x + B_x - C_x, A_y + B_y - C_y)$

Let's assume that there are 3 non-colinear points. We will prove that we can bring them into a part of the plane that holds points with x and y coordinates being at least 5. Then, each of the remaining N - 3 points can be mapped into the first quadrant using exactly one operation. From the upper equation, it is clear why this newly created point is located in the first quadrant.



If there are 3 non-colinear points, let's prove that we can bring them in the aforementioned part of the plane. Let the 3 points be A, B, C and let D be a point obtained by a single mapping, for example of point C over AB.

Now, we have a parallelogram ACBD. Notice that, by using this parallelogram, we can pave the plane in the way shown in the image (by translating that parallelogram).

Also, notice that, for each of the parallelograms used for paving the plane, there is a series of operations after which our 3 points become 3 vertices of that parallelogram.
Since the parallelogram paves the entire plane, it also paves the quadrant that holds the points with x and y coordinates being at least 5. Therefore, there must exist a parallelogram that is entirely located within that quadrant and a series of operations after which our points become 3 vertices of that quadrant.

One remaining question is how many moves this algorithm takes. There are at least 2 algorithms that find a sufficiently small number of moves:

- We run a BFS algorithm where the state is the 3 current points, and the transition is the application of one of the 3 possible operations in each state. We limit the BFS not to spread into triangles that do not intersect with the part of the plane $[-10, \infty] \times [-10, \infty]$, since we know that the solution exists even without spreading into such triangles. The coordinates are small, so we use a brute force approach to see that, worst case scenario, we need less than 1200 operations to get to the required part of the plane, which is sufficient.
- We denote with A, B, C the vertices of our triangle. Notice that there exists a vertex of the triangle, without loss of generality, let's say that it is vertex C, such that the part of the plane bounded by rays CA and CB intersects with the part of the plane where we want to bring the 3 points. We map point C over line AB. We leave the proof of correctness of this algorithm as an exercise for the reader.

Necessary skills: bfs
Category: geometry, ad-hoc

Task Klavir	**Author: Dominik Gleich**

Let's first try to solve the task for one word, meaning not for all of its prefixes.
Let's denote with S the sequence of tones for which we calculate the expected number of key presses in order to hear it. Let A be the sequence of randomly pressed keys. We denote three types of positions in sequence A, p_i^1, p_i^2, p_i^3.
Each position represents the start of tone sequence S in sequence A.

1) The general occurrence position of tone sequence S.
2) Each occurrence position of tone sequence S that is not a part of the suffix of another occurrence of tone sequence S.
3) Each occurrence position of tone sequence S that starts as suffix of another type of occurrence.

There are infinitely many of each of these positions in an infinite random sequence A.

Now, let's observe the positions of type 2 and the consecutive position differences. It is not difficult to notice that the average difference between consecutive positions is actually the solution of the problem. More precisely,

$$\lim_{n\to\infty} \frac{(p_{i+1}^2 + l - 1) - (p_i^2 + l - 1)}{n}$$

$$\lim_{n\to\infty} \frac{p_{i+1}^2 - p_i^2}{n}$$

is the solution of the problem. Let’s denote with F(1), F(2), F(3) the frequencies of appearances of tones of the types 1, 2 and 3. Given that the frequency of type 1 is any occurrence, and our generated tone sequence is completely random; the occurrence frequency is $1/N^l$, where N is the number of different piano tones. The occurrence frequency of number 2 is equal to:

$$\frac{1}{F(2)} = \lim_{n\to\infty} \frac{p_{i+1}^2 - p_i^2}{n}$$

Since the occurrence of type 1 is actually a union of occurrences of type 2 and 3, we conclude that F(1) = F(2) + F(3).

Now that we have connected these three frequencies, let’s try to connect the occurrence frequencies of the second and third type.

Since each occurrence of the third type is created from the occurrence of the second type, and by adding a sequence of letters to the end of the occurrence of the second type: more precisely, if the suffix is at the end of the occurrence of the second type, which is at the same time a prefix of the occurrence of the tone sequence of length *i*, then we need to add *l* - *i* characters in order to get one occurrence of the third type. Since all letter additions are independent and since we add the letters completely randomly, it holds

$$F(3) = F(2) * \sum \frac{1}{N^{l-i}}, i \boldsymbol{\epsilon} \left[kmp(l), kmp\big(kmp(l)\big), kmp\Big(kmp\big(kmp(l)\big)\Big), \ldots\right]$$

Here, we used the kmp function, as in the fail function in the KMP algorithm. For more details on KMP, consult additional resources.
Further manipulations gets us:

$$F(3) = F(2) * \sum \frac{1}{N^{l-i}}, i \boldsymbol{\epsilon} \left[kmp(l), kmp\big(kmp(l)\big), kmp\left(kmp\big(kmp(l)\big)\right), \ldots\right]$$
$$F(1) - F(2) = F(3)$$
$$F(1) - F(2) = F(2) * \sum \frac{1}{N^{l-i}}, i \boldsymbol{\epsilon} \left[kmp(l), kmp\big(kmp(l)\big), kmp\left(kmp\big(kmp(l)\big)\right), \ldots\right]$$
$$F(2) = \frac{F(1)}{1 + \sum \frac{1}{N^{l-i}}, i \boldsymbol{\epsilon} \left[kmp(l), kmp\big(kmp(l)\big), kmp\left(kmp\big(kmp(l)\big)\right), \ldots\right]}$$

Since the solution we want is 1/F(2), we then get:

$$\frac{1}{F(2)} = \frac{1 + \sum \frac{1}{N^{l-i}}, i \boldsymbol{\epsilon} \left[kmp(l), kmp\big(kmp(l)\big), kmp\left(kmp\big(kmp(l)\big)\right), \ldots\right]}{F(1)}$$

$$\frac{1}{F(2)} = \frac{1 + \sum \frac{1}{N^{l-i}}, i \boldsymbol{\epsilon} \left[kmp(l), kmp\big(kmp(l)\big), kmp\left(kmp\big(kmp(l)\big)\right), \ldots\right]}{\frac{1}{N^{l}}}$$
$$\frac{1}{F(2)} = N^{l} + \sum N^{i}, i \boldsymbol{\epsilon} \left[kmp(l), kmp\big(kmp(l)\big), kmp\left(kmp\big(kmp(l)\big)\right), \ldots\right]$$

The last expression is simply efficiently calculated modulo m. The solution for all prefixes of the tone sequence, which was the original task, is left as an exercise for the reader.

Necessary skills: Knuth-Morris-Pratt, mathematical problem analysis
Category: ad-hoc, probabilities