Task ZAMKA	**Author: Nikola Dmitrović**

For each integer larger than or equal to **L**, we need to determine the sum of its digits and check whether the sum is equal to given **X**. The search halts when we first find such a number (**N**). For each integer less than or equal to **D**, we need to determine the sum of its digits and check whether the sum is equal to given **X**. The search halts when we first find such a number (**M**).

The solution is written in Python 3.x.

```
left = int(input())
right = int(input())
X = int(input())

for i in range(left, right + 1):
      sum = 0
      number = i
      while i > 0:
          sum += i % 10
          i //= 10
      if sum == X:
          print(number)
          break

for i in range(right, left - 1, -1):
      sum = 0
      number = i
      while i > 0:
          sum += i % 10
          i //= 10
      if sum == X:
          print(number)
          break
```

Necessary skills: for and while loops, algorithm for summing the digits of a number
Category: ad-hoc

Task MULTIGRAM	**Author: Mislav Balunović**

Let there be given a string of length **N**.
We can iterate over all divisors **K** of number **N** and check whether it's possible for a string to consist of anagrams of length **K**.
In order to check this, we need to determine whether the substrings [1, K], [K+1, 2K], …, [N-K+1, N] are mutually anagrams.
It is sufficient to sort all these strings and check if they're equal then.

Necessary skills: strings
Category: ad-hoc

Task PERICA	**Author: Dominik Gleich**

We need to calculate how many times each number from the input appears as the largest of **K** numbers in every combination. If we sort the numbers in ascending order, we can see that this number is going to be maximal in an array of **K** numbers if all other numbers are located to its left. Therefore, the $\mathbf{i^{th}}$ number is maximal the exact number of times as the number of ways to choose **K-1** numbers out of the first **i** numbers. If we denote the number of ways to choose **K** numbers out of **N** numbers as f(n, k), then it is easy to see that f(n, k) = f(n-1, k) + f(n-1, k-1).
Using this relation, we can calculate each f(n, k).
The solution is therefore the sum of the product of the corresponding f(i, k-1) i v[i], where v[i] is the value on that location.

Necessary skills: combinatorics
Category: ad-hoc, mathematics

Task POPLAVA	**Author: Mislav Balunović**

Let's first notice that the maximal amount of water contained in a histogram with **N** columns is exactly (**N** - 1) * (**N** - 2) / 2.
The configuration to achieve this is, for example, [N, 1, 2, …, N - 2, N - 1].

The main idea of the solution is to remove several columns from the middle and put them in descending order on the side. Then the total amount of water would depend only on the columns left in the middle. A column of height h in the middle would contribute with N - 1 - h water.

Let's assume that we left columns of height h_1, h_2, …, h_k in the middle.
If we use the notation $v_i = N - 1 - h_i$, then we have reduced the problem to finding a subset of numbers {1, 2, …, N - 2} which sum is precisely X.
Such numbers can be found using a greedy algorithm. The proof is left as an exercise to the reader.

Necessary skills: greedy, mathematics
Category: mathematics

Task OOP	**Author: Dominik Gleich**

In order to solve the task, let's first try to see how we can compare a pattern and a word. Given the fact that '*' is replaced by any series of lettters, and the letters to the left and to the right of '*' remain unchanged, everything to the left of '*' must be a prefix of the word which we're comparing to, and everything to the right must be the suffix of the word we're comparing to. Therefore, if we denote everything to the left of '*' as L and everything to the right as R, it is sufficient to check whether L is a prefix of the word, R is the suffix of the word, and that the sum of lengths of L and R is less than or equal to the length of the word. In the case where this does not hold, an overlap between the prefix and suffix could occur, i.e. ("ab*ba" s "aba", "ab" i "ba" share letters in "aba"). Now we can solve the task for 40% of total points, if we use a quick comparison between strings using hashing.
The complexity of this solution is O(NQ).

Let's now solve the task for all points. First we construct a prefix tree out of all words from the input. A node in the tree that corresponds to the prefix of length p of the word X of length L will be used to store the hash value of the suffix of length L - p of the word X. After we've constructed this structure, let's try to see what query we need to answer in order to find out the number of words that correspond to the pattern. Let's denote the part of the pattern to the left of '*' as L. The answer to the query is actually the number of hashes located in the subtree of prefix L that are equal to the hash of the right part of the pattern (the part after '*'). How can we answer this question quickly?

If we use DFS to traverse the prefix tree and denote the time we first visited the node with $\mathbf{D_t}$ and the time we returned from node **t** to its parent with $\mathbf{F_t}$, it is easy to see that all nodes **x** that are located in the subtree of **t** have $\mathbf{D_x}$ in the interval $\mathbf{D_t}$, $\mathbf{F_t}$.

If we group the same hashes from the prefix tree in a structured sorted by $\mathbf{D_x}$ of the corresponding nodes, we can, for those hashes, answer how many hashes **H** there are in the subtree of node **t**. We do this by querying the structure for hash **H** how many $\mathbf{D_x}$ from the interval $\mathbf{D_t}$, $\mathbf{F_t}$ there are. If we use a map <int, vector<int>> as the structure, we can easily support the aforementioned query with a simple binary search over the corresponding vector for a given hash. If S is the number of characters in the input, the total complexity of the solution is O(S lg S).

Necessary skills: dfs, prefix tree, rolling hash, binary search
Category: data structures

Task PODNIZOVI	**Author: Mislav Balunović**

Let's first observe the following algorithm:

```
S = {} // current set of subarrays

for i in {1..K}
    x = lexicographically smallest subarray from S
    s.remove(x)
    print hash(x)
     for y in {subarrays created by adding one element to the end of
x}
        s.add(x)
```

Therefore, we iterate over subarrays ascending lexicographically and after we process a subarray we add all subarrays created from it into the processing set.

We will make a series of optimizations of the given algorithm.
Let's define the children of the subarray as all the subarrays created by adding one element to the end of the current subarray.

1) Notice that there is no need to add more than one child of a subarray into the set. Only when we process a child, do we add the next lexicographical child of a subarray into the set.
2) When we are searching for the next smallest lexicographical child of a subarray, we can do this quickly by using a Fenwick tree or a tournament tree.
3) The comparison of subarrays from the set can be done quickly if we compare prefixes of the size 2^k.

The time complexity of the algorithm is $O(N \lg^2 N)$.
For implementation details, consult the official solution that implements the described algorithm. An alternative solution uses a recursive processing of subarrays, which results in somewhat simpler code.

Necessary skills: tournament tree
Category: ad-hoc