Task BELA	**Author: Branimir Filipović**

In order to calculate the total number of points in one game of Bela, we need to sum the points of each card in the game. To determine the number of points each card is worth, we need two pieces of information: whether the card is in the dominant suit (its suit is equal to **B**), and the value of the card's symbol. Now all we need to do is look up the card's value from the given table. This can be done simply using conditional statements.

Necessary skills: conditional statements, loop statements, string processing
Category: ad-hoc

Task PUTOVANJE	**Author: Dominik Gleich**

The only important thing to do is to simulate Mislav's movement and food consumption if he starts his journey from every possible position and decides to start eating from that point. When he starts eating, he will eat a fruit if it can fit inside his stomach, otherwise he will not eat it and continue trying to eat other fruits. Therefore, we need to calculate how much fruit he will eat if he starts eating from the fruit on that position, from every position. The maximal obtained number is the solution. The total complexity of this solution is O(n^2).

Necessary skills: arrays
Category: ad-hoc

Task PIANINO	**Author: Marin Tomić**

Let's denote the keys already pressed as a_1, …, a_N. We define an array of numbers b_1, …, b_N in the following way:

$$b_1 = 0$$
$$b_{i+1} = b_i + 1 \text{ if } a_i > a_{i-1},\ i > 1$$
$$b_{i+1} = b_i - 1 \text{ if } a_i < a_{i-1},\ i > 1$$
$$b_{i+1} = b_i \text{ if } a_i = a_{i-1},\ i > 1$$

Let's denote the partial sum of array b as p_i. It is easy to see that the i^{th} note that Mirka will press is exactly $p_i * K + a_1$.

If Mirka plays the i^{th} note of the song, it holds $p_i * K + a_1 = a_i$. If $p_i = 0$, then the accuracy of the i^{th} note does not depend on K at all, otherwise it will be played correctly if K is equal to $(a_i-a_1) / p_i$ (notice that the quotient must be a non-negative integer).

Therefore, for each note for which p_i is not 0 there exists at most one good candidate for K. All the candidates can be put in an array and then we can sort them, find the one with the most appearances and choose that one as the optimum.

The complexity of the solution is O(N lg N) because of the sorting.

Necessary skills: combinatorics
Category: ad-hoc, mathematics

Task PAROVI	**Author: Mislav Balunović**

Let's denote the family of sets that have the partition located at index k as A_k, and the family of all
possible initial sets as X.

Then the solution is all sets from $X \setminus \bigcup_{i=1}^{n} A_i$

The inclusion-exclusion principle provides us with an efficient way of calculating the size of the solution set. It holds:

$$|X \setminus \bigcup_{i=1}^{n} A_i| = \sum(-1)^k \cdot |\bigcap_{j=1}^{k} A_{i[j]}|$$

In order to calculate the cardinality of the required intersection, we need to count how many pairings such that the partitions are located at certain k indices there are (we are not interested about partitions at other locations).

That number is equal to 2^t, where t is the total number of pairs that don't "cross over" any partition location. The time complexity of this solution is $O(2^N * N)$. For implementation details, consult the official solution).

Please note: A solution using dynamic programming also exists and is left as an exercise to the reader.

Necessary skills: inclusion-exclusion principle, mathematics
Category: mathematics

Task KRUMPIRKO	**Author: Dominik Gleich**

The solution worth 30% of points is trivial and implements precisely what is required in the task: from all possible choices of divisions of potato bags in the stores, which is O(2^n), choose the one with the minimal product of average prices where at least one half is of size **L**. The complexity of this solution is O(2^n * n). Let's now try to solve the task for all points.

Let's denote the total price of the potatoes in the first store as $\mathbf{c_1}$ and the total number of potatoes as $\mathbf{w_1}$, and analogously in the second store as $\mathbf{c_2}$ and $\mathbf{w_2}$, the product of the average prices is equal to $\mathbf{c_1}*\mathbf{c_2}/\mathbf{w_1}/\mathbf{w_2}$. Given the fact that the sum of prices and the sum of the number of potatoes is constant, we can modify this expression as $\mathbf{c_1}*(\mathbf{c} - \mathbf{c_1})/(\mathbf{w_1}*(\mathbf{w}-\mathbf{w_1}))$. If we fix the parameter $\mathbf{w_1}$, we can notice that this expression is minimal when $\mathbf{c_1}$ is minimal as well. Now our task is to find the minimal $\mathbf{c_1}$ for each $\mathbf{w_1}$ such that the first store contains exactly **L** or **N - L** bags of potatoes. The minimal such $\mathbf{c_1}$ is found using dynamic programming. Let **f(n, w, l)** be the smallest price when choosing exactly **l** bags of potatoes out of the first **n** bags of potatoes so that the chosen bags contain exactly **w** potatoes. The relation in this dynamic is left as an exercise to the reader. For implementation details, consult the official solution. The total complexity of the solution is O(**wn^2**).

Necessary skills: dynamic programming
Category: mathematics

Task SAN	**Author: Marin Tomić**

Let tab[y] denote the number of occurrences of the number y in the table. We can (not in an efficient way) calculate it using the following algorithm for each y from 1 to N:

```
for x from 1 to N:
      tab[x] = tab[x] + 1
      tab[x + rev(x)] = tab[x + rev(x)] + tab[x]
```

This algorithm is linear, but N can be up to 10^{10} so it is not efficient enough. It is crucial to notice that the amount of numbers for which tab[y] > 1 is very small. Let's denote the number of different numbers x such that x + rev(x) = y as cnt[y].

Notice that cnt[y] > 0 if and only if tab[y] > 1. Let's denote the array of numbers consisting of all the numbers for which cnt[y] > 0 as S and sort it ascendingly. The values of tab[y] for these numbers can be calculated using the following algorithm:

```
for each x in S:
      tab[x] = tab[x] + cnt[x]
      tab[x + rev(x)] = tab[x + rev(x)] + tab[x]
```

This algorithm consists of |S| loop iterations. It is crucial to notice that the amount of numbers in set S is relatively small (only a couple of millions) so the upper algorithm is efficient enough in order to calculate the values tab[y] for each y for which the value is larger than 1. For each other y it holds tab[y] = 1, so now we actually have the value tab[y] calculated for each y, which enables us to efficiently answer queries (using partial sums of the array tab). Notice that the indices in the array tab are very big, but we don't need the whole array so we will actually store it in a map.

We are left to calculate the values of cnt[y] for all numbers. We will recursively find all numbers y and the values cnt[y] for each y for which the value is larger than 0. Let's see what happens when we add a 6-digit number with itself, only in reverse:

```
     a1        a2        a3        a4        a5        a6
   +  a6        a5        a4        a3        a2        a1
------------------------------------------------------------

  c0  b1+c1     b2+c2     b3+c3     b3+c4     b2+c5     b1
```

Here c0, …, c5 are 0 or 1 and denote the carrying digits, b1 is (a1+a6) modulo 10, b2 is (a2+a5) modulo 10 and b3 is (a3+a4) modulo 10.

The values cnt[y] could be determined so that we fix the digits a1, …, a6 in all possible ways, but there are a lot of combinations for that. Another approach would be to determine b1, b2 i b3, and c0, …, c5 in all possible ways (therefore the sum is uniquely determined) and calculate how many different selections of a1, …, a6 results in exactly b1, b2 and b3 and c0, …, c5. How can we do this? We will use a recursive function gen which takes 4 parameters, their meanings described in the following table:

pos	means we are currently determining b_{pos}
c1	means that c_{pos-1} is equal to c_1

c2	means that $c_{6-pos+1}$ is equal to c2
num	the value of the current part of the sum a1...a6 + a6...a1

Here is the pseudocode for function gen:

```
gen(pos, c1, c2, num):

    if pos = 4:
      (we have determined the whole sum and the number of ways to
obtain it)
      if c1 != c2 return
      (c1 and c2 must be equal because they both represent c3)
      increase cnt[num] by 1 and return

    for pairs of digits (x1, x2):
      (x1 and x2 determine b_pos, we are determining digits pos and
6-pos+1 of num)
      if c1 = 1 and x1+x2 < 10 continue
      if c1 = 0 and x1+x2 >= 10 continue
      bb = (x1 + x2) % 10


      nnum = num
      set digit 6-pos+1 of nnum to bb + c2
      set digit pos of nnum to bb + 1 and call
        gen(pos+1, 1, x1+x2 >= 10, nnum)
        (c1 = 1 denotes that we want the carry digit)

      set digit pos of nnum to bb and call
        gen(pos+1, 0, x1+x2 >= 10, nnum)
        (c1 = 0 denotes that there is no carry)
```

The upper approach can be generalized for an arbitrary number of digits. The upper recursion for a number of length L performs approximately $81^{L/2}$ operations, which is too slow for all points. It is crucial to note that we don't need to iterate over all pairs of digits, because it is sufficient to iterate over all possible sums and for each sum know how many ways there is to obtain it. Then we perform approximately $19^{L/2}$ operations, which is fast enough for all points. We leave the details as an exercise to the reader, but they can also be found in the official solution.

Necessary skills: recursion, map
Category: dynamic programming, mathematics