Editorial

Tasks, test data and solutions were prepared by: Nikola Dmitrović, Dominik Fistrić, Bartol Markovinović, Bojan Štetić, Pavel Kliska, and Krešimir Nežmah. Implementation examples are given in the attached source code files.

Task Lampice

Prepared by: Bartol Markovinović
Necessary skills: for-loop, array/list

The problem asks to find a continuous subarray (that is pattern) which repeats k times in a row. The simplest way to do this is to iterate over all continuous subarrays and check if it repeats k times in a row. We can iterate over these subarrays using two nested for loops, one going over all possible left ends of the subarray, and one going over all right ends. Denote by l and r the left and right end of the subarray, respectively. The length of the current subarray (denote it by d) is then equal to $r-l+1$. In order to check whether it repeats k times, we must check whether the following $k-1$ subarrays of length d are the same as the initial subarray (from l to r), that is whether $[l+d, r+d], [l+2d, r+2d], \ldots, [l+(k-1)d, r+(k-1)d]$ is the same as $[l, r]$. This is done using two nested for loops, one going over those subarrays and the other checking if they are the same with the initial one (from l to r).

Task Cijanobakterije

Prepared by: Pavel Kliska
Necessary skills: tree traversal, diameter of tree

One possible characterisation of a tree is that it is an acyclic graph with n nodes and $n-1$ edges. This means that in the first subtask, the problem is actually to determine the length of the longest chain in a tree. This is a standard problem of finding the diameter of a tree. It can be solved by noticing that all nodes which are furthest away from an arbitrary node are the ends of some longest chain. Therefore, it is sufficient to take an arbitrary node of the tree, find a node which is furthest away from it, and then find the distance from this node to the node furthest away from it.

For the whole solution, one should notice that when connecting two trees, the diameter of the new tree can't be larger than the sum of their diameters (otherwise, we would have a chain in the first or the second tree which is larger than it's diameter). On the other hand, a diameter which has the size of the sum of the two diameters is achievable because we can connect the ends of the two diameters. Therefore, by repeatedly connecting the trees, the largest diameter we can obtain is equal to the sum of the diameters of the individual trees, which is also the answer to the problem.

Task Akcija

Prepared by: Krešimir Nežmah
Necessary skills: dynamic programming and binary search or greedy and fracturing search

The subsets from the problem have a very rich structure and it is possible to arrive at the solution from different angles. The scoring was generous, but the last subtask should be challenging. For most contestants, the most natural idea was some sort of dynamic programming, but we will present a solution which uses a different (perhaps more complex) idea, which can be generalized to other problems that ask for the k-th best answer.

The presented solution consists of several parts:

1. How to check if a subset is obtainable?
2. How to find the best obtainable subset?

3. How to find candidates for the next best obtainable subset?
4. How to choose between the candidates to arrive at the k best ones?

The proofs are left as an exercise for the reader.

How to check if a subset is obtainable?

Claim 1: If a subset is obtainable, a valid sequence of calls can be made by ordering the products by d_i.

Claim 2: In an empty array put $+1$ on each position i ($i = 1, ..., n$), and -1 on each position d_i and make prefix sums over it. A set of products is obtainable iff there are no negative numbers in such an array.

We can build a min. segment tree over the array from claim 2, which allows us to insert/erase products while maintaining the information whether the current set is obtainable. An update is simply ± 1 for some suffix of the array.

How to find the best obtainable subset?

The following greedy algorithm works:
Sort the set in question so that $w_1 < ... < w_n$.
We iterate over the products in order and try to add the current product if it maintains the attainability property (which we can check with the segment tree).

Claim 3: The greedy algorithm produces a set which is maximum in size, and out of all such sets it has the minimum cost, i.e. it is the best choice.

Proof outline: The greedy algorithm produces an obtainable subset which can't be enlarged (i.e. is maximal). Every two maximal subsets are actually of the same size - the maximum one. The order $w_1 < ... < w_n$ gives the minimum cost.

How to find candidates for the next best obtainable subset?

Let S be the best obtainable subset of the index set $\{1, ..., n\}$. For each $i \in S$ we want to know what is the best obtainable subset of $\{1, ..., n\} \setminus \{i\}$. One of them will clearly be the second best, but as we'll see later, it's useful to know the cost for all of them.

Claim 4: The best obtainable subset of $\{1, ..., n\} \setminus \{i\}$ is made by removing i from S and adding some $j \notin S$.

We can find the desired j for some i in the following way:
In the segment tree we store the array from claim 2 for set S, and we make an update of $+1$ for the suffix starting at d_i, which corresponds to removing i from S. A valid choice for j is now any index such that there are no zeroes in the suffix of the array starting from d_j. Therefore, d_j has to be to the right of the rightmost zero in the current array, which is actually the rightmost zero to the left of d_i (now we see that the update was unnecessary, and we could have made a segment tree query to find the rightmost zero left of d_i). In any case, the only condition for chosing j is that $d_j \geq c$, for some c that we know how to calculate. Out of all such j, we should choose the one with minimum w_j. The answer can be precomputed for all c by doing a sweep over the indices outside of S, decreasing by d_j.

How to choose between the candidates to arrive at the k best ones?

In a given moment, the current search spaces will be described by describing the status of each index:

- This index must be in the set.
- This index can't be in the set.
- For this index we have a choice whether it is in the set or not.

We'll have a priority queue to store all the different search spaces not yet explored. The search spaces will be disjoint and their union will cover all possibilities not yet visited. In the priority queue, they will be sorted by the cost of the best obtainable subset within that search space.

It was already mentioned how to find the best obtainable subset within a search space. The situation is a bit different since this time we have indices which must/can't be taken. But the idea can easily be modified just by not even taking into consideration the indices which we can't take, and the ones we must take we use for updating the segment tree before executing the greedy algorithm. It's easy to see that the mentioned claims are still true in this modified case.

We pop the minimum element from the priority queue to determine the next best obtainable subset. The poped search space then needs to be partitioned into smaller pieces which don't include the best obtainable subset. Let S be this best subset. The smaller subsets have the following form:

- The first element from S can't be taken, for the rest we can choose.
- The first element from S must be taken, the second must not, for the rest we can choose.
- The first two elements from S must be taken, the third must not, for the rest we can choose.
- ...

Of course, the indices outside of S are left to be of the same status as they were before.
According to what was mentioned so far, when deleting an index from S, it is enough to add a single new index. Therefore, when adding new search spaces to the priority queue, we won't keep track of the status of all indices, rather we'll keep track of just the indices which change their status.

We'll have a total of k popmin operations. For each of them, we'll have to find the best obtainable subset for some search space, and then partition it into smaller pieces (and for each of them determine it's cost, to be able to add it to the priority queue). Using the segment tree, this can all be done in $O(n \log n)$ for each popmin operation. The priority queue will have at most nk elements. The total complexity is then $O(nk \log(nk))$.

The presented idea is called fracturing search. The reason a lot of things are true for these obtainable subsets is because they have a matroid structure.

Task Ekoeko

Prepared by: Pavel Kliska and Krešimir Nežmah
Necessary skills: ad-hoc, greedy, Fenwick tree, counting inversions

Let's first solve the subtask where the first and the second half of the string are anagrams. It's never useful to swap two adjacent characters if they are the same, which is why the relative order of equal characters never changes. That's why we can pair up the i-th occurrence of some letter in the first half with the i-th occurrence of that letter in the second half, and it the end these characters have to be in the same positions in their corresponding halves. In this way we obtain n pairs and the problem can be formulated as follows: given a sequence of $2n$ numbers, where the first and the second half are both permutations of $1, 2, \ldots, n$, equate the halves using the minimum number of adjacent swaps.

We claim that it's optimal to make swaps only in the second half until we make it equal to the first half. To show this, denote by $d(p, q)$ the minimum number of swaps to turn a permutation p into another permutation q. Notice that for every permutation r, we can get from p to q by first going from p to r and then from r to q, so we conclude that $d(p, r) + d(r, q) \geq d(p, q)$. In our case, r represents the final permutation which will be repeated in each half. We see that equality occurs for example for $r = p$, which is what we wanted to show.

Without loss of generality, we can change the labels of the pairs so that the first half consists precisely of the numbers $1, 2, \ldots, n$, and the second half is some permutation. This is now a classic problem of

sorting a permutation with the minimum number of swaps, which turns out to be the same as counting the number of inversion, i.e. number of pairs of indices $i < j$ where $q_i > q_j$. We can count the number of inversions in $O(n \log n)$ using a Fenwick tree. Alternatively, you can think of it as doing a greedy algorithm by first moving the number 1 in permutation q to the first position, then number 2 to the second position and so on. For more details and similar ideas see here.

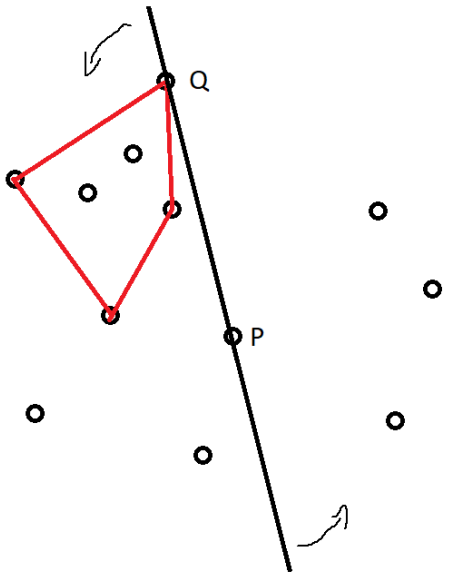
For the whole solution, we must also at some point get the initial string to a state in which the first and the second half are anagrams. Having in mind the partition into n pairs, we know for each letter whether it should end up in the first or in the second half. In other words, we have n left characters and n right characters which have to end up in their respective halves. We can make all swaps between left and right characters before making any swaps between two left or two right characters, so it's optimal to first get each letter to the desired half, and then solve the problem as described above. It's easy to show that we should first move the first left character to the first position, then the second left character to the second position, and so on. The number of swaps needed to get a left character to its final position is equal to the number of right characters which are to the left of it. This can be calculated in $O(n)$, making the final complexity $O(n \log n)$.

Task Kućice

Prepared by: Krešimir Nežmah
Necessary skills: geometry, circular sweep line, two pointers

For the first subtask, the number of points within the convex hull of any subset of points is equal to the size of that subset, so the expected value is $\frac{n}{2}$ and $m = 2^{n-1}n$.

We can calculate m by going over each of the 2^n subsets of points, finding the convex hull and checking to see how many points lie in the hull. A naive implementation of this was sufficient to solve the subtask in which $n \leq 15$.



We can also calculate m in a different way - by fixing a point and counting how many subsets there are where the point lies in the hull. It turns out that it's easier to count the complement, i.e. how many subsets there are whose hull doesn't contain the fixed point. Notice that after fixing a point and a subset whose hull doesn't contain it, there is always a line that passes through the point, but which doesn't intersect the hull. If we call the fixed point P, we can think of this line as rotating around P counterclockwise until it hits a point Q on the hull (see image above).

For a fixed choice of P and Q it is sufficient to find the number of points which lie on one side of the line PQ. If we call this number k (not counting Q), then the number of subsets whose hull doesn't contain P and for which Q is the point that is 'first in the counterclockwise direction' is 2^k.

Therefore, the solution can be determined in the following way. Fix a point P and sort the remaining points circularly around P. Iterate over Q and maintain the number of points on each side of the line PQ. The points in a halfplane form a circular segment, so it is enough to maintain the last point (with respect to the circular order) that is still in the current halfplane, using the method of two pointers. For each point P we do one sort in $O(n \log n)$ and then calculate the number of subsets not containing P in $O(n)$ with two pointers. The total complexity is then $O(n^2 \log n)$.

For the circular sort and for maintaining the current halfplane it is helpful to use the function $\texttt{CCW}(A, B, C)$ which determines for three given points A, B and C whether they are listed in counterclockwise order:

$$\texttt{CCW}(A, B, C) = x_A(y_B - y_C) + x_B(y_C - y_A) + x_C(y_A - y_B),$$

where $\texttt{CCW}(A, B, C) > 0$ if and only if the points A, B and C are listed in counterclockwise order. To sort the points circularly, we can divide them into the ones above and the ones bellow point P, sort these two groups separately, and then merge them. Within a group we use the `CCW` function as a comparator with arguments being the two points in question and the fixed point P.