PET

For each contestant, calculate the number of points and keep track of which contestant had the most points. This is possible with or without arrays.

KEMIJA

Each time a vowel occurs in the original sentence, the encoded sentence will have the letter p and the same vowel again. We need to find all vowels in order and, for each, delete the following two letters.

CROSS

The entire task is to implement the procedure from the problem statement. One way is to isolate the cross-hatching method for a single number in a separate function and keep calling it with different arguments until there is no more work for any number. See included source code.

MATRICA

In many problems where the lexicographically smallest solution is sought, an algorithm can be formed if the following question can be answered: "Suppose I have this partial solution. Is it possible to extend to make a complete solution?" In this problem, it would mean that a certain number of cells have been filled (and part of the letters used) and we want to know if it is possible to fill the rest of the matrix (so that it is symmetric).

If we can answer this question, then we can fill the cells in order and, for each cell, consider putting letters in it in increasing order: "If I put the letter A, will I be able to complete the matrix? No? What if I put the letter B?" etc. This is an example of a greedy algorithm, taking advantage of the fact that for lexicographical order it is always better to put smaller letters as early as possible.

How to answer the question? For the matrix to be symmetric, cells other than ones on the diagonal must be paired. In other words, for every cell above the diagonal, the corresponding cell below the diagonal must contain the same letter. In even more words, letters we can't pair up must go on the main diagonal. Of the 26 letters possibly available to us, the number of those available in odd quantities must not be greater than the number of cells on the diagonal, because we can't pair up those odd letters.

Any implementation of this idea and complexity $O(N^2)$ should have scored 60%.

For 80% an efficient implementation was needed, with lower constant factors. It is interesting that a solution that uses less memory (i.e. doesn't build the entire matrix then outputs part of the columns, but keeps only needed columns) is noticeably faster in practice.

For full points, by analyzing the first algorithm deeper, we can observe that, within a single row, we often place the same letter in large blocks. Two types of events can interrupt this block-filling:

1. We run out of the letter we are filling with.
2. We run into a column that needs to be output.

Using this approach, the number of steps in our algorithm is proportional to $N \cdot (P+26)$.

BST

The problem asks us to calculate the depth of each new node and keep track of the sum of depths.

To start with, note that we cannot just directly simulate the algorithm in the problem statement. If the tree is not balanced (and the authors of the test data will ensure that it is not), its depth will grow to O(N) and the overall complexity will be $O(N^2)$, way too much with N up to 300000.

Because the tree takes O(1) memory per node, we can afford to construct it explicitly (so that we know for each node its left child, right child, parent and depth), as long as the construction is efficient – for example, O(log N) per number or faster. As it turns out, in this problem it suffices to keep just the depths.

Let X be the number we are inserting, L be the largest already inserted number smaller than X, and R be the smallest already inserted number larger than X.

We can show by contradiction that L will necessarily be an ancestor of R or vice versa. Also, because of the binary search tree property, after insertion X will become the left child of R or the right child of L, depending on which of the two is deeper in the tree.

So if we could efficiently determine the number L and R for every X, it would be easy to maintain the entire tree and calculate the numbers sought in the problem.

Assuming they made these observations, contestants using C++ had it easy because the built-in data structures set and map (which are themselves implemented as binary search trees behind the scenes, but balanced) support these operations – efficiently finding the element just above and just below a given element. The overall complexity of such a solution is O(N log N).

In Pascal or C, such a data structure is not available so we need to keep thinking.

Suppose we build a doubly linked list and initialize it to contain the numbers 1 through N in sorted order. If we remove the numbers in the input in reverse order, the list will be able to give us exactly what we want. At every point, the list will contain exactly the numbers that would have been inserted up until then, and it's easy to retrieve L and R, given X; these are the predecessor and successor of X in the list. The overall complexity of this solution is O(N).

Another intriguing approach is to keep in an array of length N, for each number that has been inserted, its predecessor and successor. When inserting a new number X, we do a bidirectional (breadth-first) search around X to find a number that has already been inserted. If we first find L, its successor is the number R. If we first find R, its predecessor is the number L. It is not obvious, but because this array gets dense fast, the overall complexity of inserting all numbers is O(N log N), even though individual insertions can take a number of steps linear in N.

The included source code demonstrates all three approaches; the first written in C++, the other two in Pascal.

NAJKRACI

A small modification to Dijkstra's shortest path algorithm allows us to find roads that are part of shortest paths from a source city.

For a fixed city C, the roads form a directed acyclic graph (DAG) and we can use dynamic programming to calculate:

- to(v), the number of paths from city C to city v;
- from(v), the number of paths from city v to any other city.

To calculate the values of to(v) we use the formulas:

$to(C) = 1$

$to(v) = \sum to(u)$, for each city u such that edge (u, v) is part of the DAG.

To calculate the values of from(v) we use the formula:

$from(v) = 1 + \sum from(w)$, for each city w such that edge (v, w) is part of the DAG.

The values of to(v) are calculated in order in which Dijkstra's algorithm processes the vertices, while from(v) is calculated in reverse order.

Knowing these values we can calculate how many shortest paths from C to some other city contains a road R = (A, B).

If road R is not part of the DAG, then this amount is zero. Otherwise it is to(A) · from(B). The total number of shortest paths containing road R is calculated as the sum of these products for every starting city C.