INTRODUCTION
Enclosed are three programming problems. We ask that you read all three descriptions thoroughly then create a program to solve one of the problems. If you choose to do more than one problem, we will choose and evaluate only one of your solutions. We are assessing a number of things including the design aspect of your solution and your object oriented programming skills. For the solution, we request that you use C# and provide suitable unit tests for your solution. You may not use any external libraries to solve this problem, but you may use external libraries for testing purposes.

Please also include a brief explanation of your design and assumptions along with your code.

TO SUBMIT YOUR CODE:
1. Create an email to ISTechRecruitment@ishltd.co.za.
2. Please title your submission with the name of the PROBLEM SELECTED.
3. Attach your solution to the email. PLEASE NOTE: Certain file extensions will be blocked for security purposes. You should NOT include any executable attachments, including those with .exe, .dll or .lib extensions. For security reasons, please do not submit your solution as an .msi file.

As a general rule, we allow three days from the date that you receive this email to submit your code, but you may request more time if needed.

INTRODUCTION TO THE PROBLEMS:
There must be a way to supply the application with the input data via a text file. The application must run. You should provide sufficient evidence that your solution is complete by, as a minimum, indicating that it works correctly against the supplied test data. Please note that you will be assessed on your judgement as well as your execution.

PROBLEM ONE: KRUNCH

A krunched word has no vowels ("A", "E", "I", "O", and "U") and no repeated letters. Removing vowels and letters that appear twice or more from MISSISSIPPI yields MSP. In a krunched word, a letter appears only once, the first time it would appear in the unkrunched word. Vowels never appear.

Krunched phrases similarly have no vowels and no repeated letters. Consider this phrase:

```
RAILROAD CROSSING
```

and its krunched version:

```
RLD CSNG
```

Blanks are krunched differently. Blanks are removed so that a krunched phrase has no blanks on its beginning or end, never has two blanks in a row, and has no blanks before punctuation. Otherwise, blanks are not removed. If we represent blanks by "_",

```
MADAM_I_SAY_I_AM_ADAM__
```

krunches to:

```
MD_SY
```

where the single remaining blank is shown by "_".

Write a program that reads a line of input (whose length ranges from 2 to 70 characters), and krunches it. Put the krunched word or phrase in the output file. The input line has only capital letters, blanks, and the standard punctuation marks: period, comma, and question mark.

SAMPLE INPUT:

```
NOW IS THE
TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR
COUNTRY.
```

SAMPLE OUTPUT:

```
NW S TH M FR L GD C Y.
```

PROBLEM TWO: BUY LOW, BUY LOWER

The advice to "buy low" is half the formula to success in the bovine stock market. To be considered a great investor you must also follow this problems' advice:

"Buy low; buy lower"

Each time you buy a stock, you must purchase it at a lower price than the previous time you bought it. The more times you buy at a lower price than before, the better! Your goal is to see how many times you can continue purchasing at ever lower prices.

You will be given the daily selling prices of a stock (positive 16-bit integers) over a period of time. You can choose to buy stock on any of the days. Each time you choose to buy, the price must be strictly lower than the previous time you bought stock. Write a program which identifies which days you should buy stock in order to maximize the number of times you buy.

Here is a list of stock prices:

```
Day   1  2  3  4  5  6  7  8  9  10 11 12
Price 68 69 54 64 68 64 70 67 78 62 98 87
```

The best investor (by this problem, anyway) can buy at most four times if each purchase is lower than the previous purchase. One four day sequence (there might be others) of acceptable buys is:

```
Day   2  5  6  10
Price 69 68 64 62
```

INPUT FORMAT:

- Line 1: N (1 <= N <= 5000), the number of days for which stock prices are given;
- Lines 2...n: A series of N space-separated integers, ten per line except the final line which might have fewer integers.

SAMPLE INPUT:

```
12
68 69 54 64 68 64 70 67 78 62
98 87
```

OUTPUT FORMAT:

Two integers on a single line: The length of the longest sequence of decreasing prices and the number of sequences that have this length (guaranteed to fit in 31 bits). In counting the number of solutions, two potential solutions are considered the same (and would only count as one solution) if they repeat the same string of decreasing prices, that is, if they "look the same" when the successive prices are compared. Thus, two different sequences of "buy" days could produce the same string of decreasing prices and be counted as only a single solution.

SAMPLE OUTPUT:

```
4 2
```

PROBLEM THREE: TASK DURATION

You are trying to solve a very complex problem. In order to simplify it, you have divided it into many sub tasks. Most of these sub-tasks can be run in parallel, but some are dependent on the previous resolution of other tasks. There is no limit on the number of tasks that can be run in parallel. Each task has an associated computation time. You are given a subset of these tasks. For each of them you need to specify the minimal computation time for resolving the task (you must consider task dependencies).

INPUT FORMAT:

#Number of tasks
n

#Task duration <- task x has an associated computation time of tx
x,tx

#Task dependencies <- the resolution of task x depends on previously solving tasks y,z,w
x,y,z,w

It is impossible for two different tasks to be dependent on the resolution of one common task:

#Task dependencies <- this is not valid
x,y
z,y

The expected output is the following format: taskId taskComputationTime
x tx
y ty
z tz

From the standard input you will receive a set of tasks for which to compute the total time in the following format:

x,y,z

SAMPLE INPUT:

```
#Number of tasks
6

#Task duration
0,2
1,3
2,4
3,9
4,7
5,9

#Task dependencies
0,4
3,0,1,2
4,5
```

SAMPLE INPUT FOR TASKS TO COMPUTE TIME:

```
3,1,4
```

SAMPLE OUTPUT:

```
3 27
1 3
4 16
```