Assignment

The solution must be contained by the Program.cs file, which should be renamed before submission. The name of the file to be submitted has to be the unique identifier code of the task, and your Neptune code, separated by an underscore, using capital letters: **TASKCODE_NEPTUN.cs**

Create a program similar to a word search game that finds the secret message.

The goal of the word search game is to find the given words (W) in a text (T), based on which we can determine the secret message (S). The words to be searched can appear in the text either reading from left to right or in reverse order, but always continuously. The given words may have multiple occurrences in the text, and individual characters may appear in more than one word (i.e., the words may overlap).

After finding the words, there may be letters that were not used in any of the words. The task is to collect these characters from left to right, which will give us the secret message that we need to print on the screen.

Input (Console)

- *1st line* - the number of words to be found (N)
- *the next N lines* - the W words to be found
- *(N + 1)th line* - the T text in which the W words need to be found

Output (Console)

- the secret message S formed from the remaining characters in the T text

Constraints

- $1 \leq N \leq 100$
- $1 \leq |W| \leq 50$
- $|W| \leq |S| \leq 1\ 000$
- $T, W \in \{0 - 9a - zA - Z\}$
- $1 \leq |T| \leq 999$

Example

Input

```
2
BAL
BOB
LABOBOR
```

Output

```
OR
```

Interpretation

Based on the first input value, two words need to be found in the text, which will be the second and third input values: BAL and BOB. The fourth input value is the text where the previous words need to be found: LABOBOR.

The first word (BAL) is found by reading it from right to left, with one occurrence. The second word (BOB) is also found with one occurrence, overlapping with the first word: LABOBOR. Ignoring the characters that belong to the found words, we get the word OR, which is the secret message that needs to be displayed in the output.

Test cases

Test the correct operation of the solution with at least the following inputs.

1.

Console input
```
2
BAL
BOB
LABOBOR
```

Console output
```
OR
```

2.

Console input
```
2
ALMA
PAPA
ALMAPPAPA
```

Console output
```
P
```

3.

Console input
```
4
AZ
AZTA
ALAKZAT
ALKAT
AlAZTAKLAtaALAKZATt
```

Console output
```
Altat
```

4.

Console input
```
2
3MB3R
M3R
3M3R3MB3R3
```

Console output
```
33
```

5.

Console input
```
7
SR5
6
O6CSUS
C5R
2JY
4IHG
J
tSR5Cn2JYgGHI4wbO6CSUSjJ6
```

Console output
```
tngwbj
```

The above test cases do not necessarily include all possible states of the input and output, so you may also test the correct operation of the application with your own test cases.

Supplements

General rules related to the task:

- The solution has to be made as a `Console Application`, ignoring "top-level statements" but keeping any imports.
- The source file `Program.cs` has to be submitted. The name of this file should be replaced by the identifier of the task and the author's Neptune code, separated by an underscore, using capital letters: **TASKCODE_NEPTUN.cs**.
- During the implementation, if possible, use concepts, approaches, or algorithms explained in the lecture and the lab, taking into account those defined in *Constraints*, but apart from these, feel free to be creative in solving the task.
- When creating the application, always prefer to use the appropriate types and to create professional-looking (*formatted, free of unnecessary variables, instructions*) code. We suggest following the standard naming conventions detailed here.
- **Do not copy or submit another person's solution!** In any such case, all identical solutions (including structural similarity) will be marked as duplicates and the solutions will be rejected.
- **Solutions submitted after the deadline or with an incorrect name, or solutions that do not correspond to the description, or have compilation issues, or runtime exceptions (assuming a correct input) won't be evaluated.**
- The task description is structured as follows (* - optional):
  - *Task description* - the problem you need to solve
  - *Input* - information about the input
  - *Output* - information about the expected output
  - *Constraints* - restrictions or limitations related to the input, output and algorithm, which must be taken into account, and the input limits defined here are met by the tests in all cases, so there is no need to prepare for cases that are not defined here
  - **Remarks* - additional comments related to the task or implementation
  - *Example* - an example to help understanding the problem
  - *Teszt cases* - additional test cases to test the correct operation of the algorithm, which does not necessarily include all possible states of the input and output(s)
- **In all cases, the application must print or input exactly what the task requires, as the solution is evaluated automatically.** So, for example, if the application starts by writing the message `"Give me a number:"` to the console, then the evaluation will fail, the solution will be marked as incorrect because only a number should have been read instead of reading a number along with any message.
- During the evaluation, the application will only be tested with the correct input according to *Constraints*, so the application does not have to take care of values outside the given ranges.
- A single `Console.ReadLine()` method call can be included as the last statement of the submitted solution.
- The automatic evaluation consists of four parts:
  - Unit Tests - to test the operation of the application at runtime
  - Syntax check - to check the structure of the application
  - Search for duplicates - to reject identical solutions
  - Additional metrics - only informative
- An HTML report is generated from the results of the evaluations, which will be published through the e-learning platform.
- The minimum requirement for the submitted solution:
  - It should not contain any compile warnings (`Solution contains 0 compile time warning(s)`).
  - It should not contain compile errors (`Solution contains 0 compile time error(s)`).
  - It should pass all syntax tests (`0 test warning, 0 test failed`).
  - It should not fail any unit tests (`0 test failed, 0 test warning, 0 test was not run`).
- The solution must compile and execute with ***.NET 6*** framework using ***C# 10***. Regardless, other versions of the .NET framework and C# language can be used during creation, but before submitting,

make sure that your solution is compatible with *.NET 6* and *C# 10* versions.
- In addition to the restrictions above, there are other general language constructs and functionalities that you are not allowed to use to solve the task (*this list may change from task to task*):
  - Methods: `Array.Sort`, `Array.Reverse`, `Console.ReadKey`, `Environment.Exit`
  - LINQ: `System.Linq`
  - Attributes
  - Collections: `ArrayList`, `BitArray`, `DictionaryEntry`, `Hashtable`, `Queue`, `SortedList`, `Stack`
  - Generic collections: `Dictionary<K,V>`, `HashSet<T>`, `List<T>`, `SortedList<T>`, `Stack<T>`, `Queue<T>`
  - Keywords:
    - Modifiers: `protected`, `internal`, `abstract`, `async`, `event`, `external`, `in`, `out`, `sealed`, `unsafe`, `virtual`, `volatile`
    - Method parameters: `params`, `in`, `out`
    - Generic type constraint: `where`
    - Access: `base`
    - Contextual: `partial`, `when`, `add`, `remove`, `init`
    - Statement: `checked`, `unchecked`, `try-catch-finally`, `throw`, `fixed`, `foreach`, `continue`, `goto`, `yield`, `lock`, `break` - *in loop*
    - Operator and Expression:
      - Member access: `^` - *index from end*, `..` - *range*
      - Type-testing: `is`, `as`, `typeof`
      - Conversion: `implicit`, `explicit`
      - Pointer: `*` - *pointer*, `&` - *address-of*, `*` - *pointer indirection*, `->` - *member access*
      - Lambda: `=>` - *expression, statement*
      - Others: `?:` - *ternary*, `!` - *null forgiving*, `?.` - *null conditional member access*, `?[]` - *null conditional element access*, `??` - *null coalescing*, `??=` - *null coalescing assignment*, `::` - *namespace alias qualifier*, `await`, `default` - *operator, literal*, `delegate`, `is` - *pattern matching*, `nameof`, `sizeof`, `stackalloc`, `switch`, `with` - *expression*, `operator`
    - Types: `dynamic`, `interface`, `object`, `Object`, `var`, `struct`, `nullable`, `pointer`, `record`, `Tuple`, `Func<T>`, `Action<T>`,