Practice Problem 1: The Anagram Alchemy of the Mushroom Kingdom

In the adventurous land of Super Mario Brothers, Mario and Luigi have stumbled upon a peculiar puzzle left behind by the ancient Koopa sages. The brothers discovered that certain magical words hold the key to unlocking hidden power-ups, and these words are anagrams of each other, woven with the same letters but in a different order.

To harness these mystical power-ups and gain an edge in their quest against Bowser, the brothers need to quickly determine if two given words are magical anagrams. They have enlisted your help to create a program that can solve this riddle.

Specifically, you need to implement the function:

```
   static boolean areAnagrams(String word1, String word2)
```

which returns a

```
   boolean
```

Indicating whether `word1` and `word2` are anagrams or not.

<u>Input/Output Format</u>

Input:

The first line in the test data file contains the number of test cases (*N*). After that, each line contains one test case, i.e., two words consisting of letters a-z (no capital letters or other symbols). You can assume that *1 <= N <= 10000.*

Output:

For each test case, the program will output a single string “Yes” or “No”, indicating whether the two words are anagrams or not. Note that the words may contain repeated letters. For two words to be anagrams, they must contain the same number of repetitions if any.

Note:

We have provided a skeleton program that reads the number of test cases and the input strings. It also prints the output based on the `areAnagrams()` method that you need to implement.

Examples:

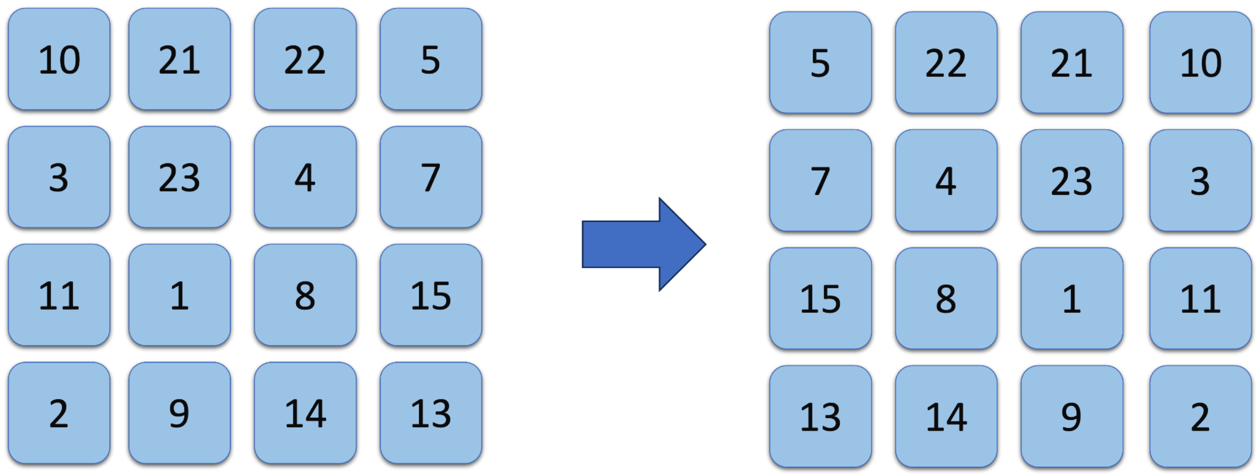
The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**
`3`	
`listen silent`	`Yes`
`binary brainy`	`Yes`
`admirer maried`	`No`

Practice Problem 2: Bowser's Block Puzzle!

Mario stumbled upon a strange contraption in Bowser's latest hideout – a 4x4 grid filled with numbered blocks! He needs your help to decipher its secret. It turns out, flipping the grid horizontally reveals a hidden message!

As an example, on the left below you can see the original grid, and on the right, the horizontally flipped grid. You are to help Mario reveal the hidden message by flipping the grid that he sees.



Specifically, you need to implement the function:

```
   static int[][] flipGrid(int[][] grid)
```

which returns a

```
   int[][]
```

Which is the result of flipping the grid horizontally.

<u>Input/Output Format</u>

Input:

The first line in the test data file contains the number of test cases (*N*). After that, each set of four lines (containing 4 integers each => total 16 numbers) constitutes one test case.

Output:

For each test case, the program should output the horizontally flipped grid as discussed above.

Note:

We have provided a skeleton program that reads the number of test cases and the input strings. It also prints the output based on the `flipGrid()` method that you need to implement.

Examples:

The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**
`2`	
`10 21 22 5`	`5 22 21 10`
`3 23 4 7`	`7 4 23 3`
`11 1 8 15`	`15 8 1 11`
`2 9 14 13`	`13 14 9 2`
`9 16 95 32`	`32 95 16 9`
`27 97 45 93`	`93 45 97 27`
`36 90 36 60`	`60 36 90 36`
`80 53 9 5`	`5 9 53 80`

Practice Problem 3: Triangular Trek Through Toad Town

The Toads of Toad Town are planning a hiking adventure, but they need your help navigating the trails! The town's landmarks are interconnected with paths, forming triangles that create exciting hiking circuits. Your task is to count the number of possible triangular trails to help the Toads plan their routes.

Specifically, you need to implement the function:

```
static int countTriangles(int M, boolean[][] paths)
```

which returns a

```
int
```

which is the number of distinct triangles that exist between the landmarks. Note that each triangle should only be counted once.

Assume that the landmarks are numbered 0, 1, …, M-1. The second input to your function is a boolean *M-by-M* array that indicates whether there is a path between two landmarks (i.e., `paths[a][b] = true` ⇒ there is a path between landmarks **a** and **b**).

Help the Toads explore all the triangular trails and make their hiking adventure a success!

Input/Output Format

Input:

The first line in the test data file contains the number of test cases (*N*). After that, the next set of lines until a blank line indicates a single test case. For each test case, the first line contains *M*, and the next set of lines (until a blank line) contain the paths between the landmarks.

Output:

For each test case, the program should output the number of distinct triangles that exist between the landmarks.

Note:

We have provided a skeleton program that reads the number of test cases and the inputs. It also prints the output based on the `countTriangles()` method that you need to implement.

Examples:

The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**

`2`
`6`
`5 0`
`1 2`
`2 3`
`3 1`
`1 4`
`2 4`
`4 5`
	
	`There are 2 distinct triangles among the landmarks.`

`8`
`6 0`
`1 2`
`2 3`
`3 1`
`1 4`
`2 4`
`4 6`
`4 5`
`5 6`
	`There are 3 distinct triangles among the landmarks.`

Practice Problem 4: Princess Peach's Surprise Party

Mario and Luigi need to plan a surprise party for Princess Peach and need to keep their messages secret from her using an encryption technique. They choose to use the Vigenère Cipher, a method of encrypting alphabetic text through a simple form of polyalphabetic substitution. This technique, initially described by Giovan Battista Bellaso in 1553, was wrongly credited to Blaise de Vigenère in the 19th century due to a stronger cipher he presented in 1586.

The Vigenère Cipher operates similarly to the Caesar cipher, where each letter in the text is shifted a certain number of places down the alphabet. For example, with a shift of 3, "A" becomes "D", "B" becomes "E", and so on. The Vigenère Cipher, however, uses multiple Caesar ciphers in sequence with different shift values determined by a keyword.

For instance, if Mario and Luigi want to encrypt the message "ATTACKATDAWN" using the keyword "MUSHROOM":

Plain text: ATTACKATDAWN
Keyword: MUSHROOMMUS

Each letter in the message is shifted by an amount corresponding to the respective letter in the keyword. The shift for each letter is determined by its position in the alphabet, so if "M" represents the shift, "A" becomes "M", "B" becomes "N", and this pattern continues until the alphabet wraps around.

If the keyword letter is "M", the shift is 12 places, so "A" becomes "M", and so on. The resulting encrypted message might look completely different from the original.

Your task is to create a program that helps Mario and Luigi encrypt their messages for the surprise party planning. Specifically, you need to implement the method:

```
private static String solveVigenere(String key, String plaintext)
```

That returns a:

```
String
```

i.e., the ciphertext corresponding to the plaintext.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases (< 100). After that, each line contains one test case with two strings: the first string is the *keyword*, and the second string is the *plaintext*. Both the keyword and plaintext only contain capital letters (from A to Z) -- all numbers or punctuation marks (including white spaces) are stripped out from the message.

Output:

For each test case, you are to output the encrypted ciphertext. The exact form of the output is shown below.

Note:

We have provided a skeleton program that reads the number of test cases and the inputs. It also prints the output based on the `solveVigenere()` method that you need to implement.

Examples:

The output is shown with extra blank lines so that the output is shown aligned with the appropriate input line; those blank lines should not be present in the actual output. (The provided skeleton code handles that for you).

Input:	**Output:**
`4`	
`LEMON ATTACKATDAWN`	`Ciphertext: LXFOPVEFRNHR`
`ABCD CRYPTOISSHORTFORCRYPTOGRAPHY`	`Ciphertext: CSASTPKVSIQUTGQUCSASTPIUAQJB`
`ABCDE CRYPTOISSHORTFORCRYPTOGRAPHY`	`Ciphertext: CSASXOJUVLOSVISRDTBTTPIUEPIA`
`LUCKY COMPUTINGGIVESINSIGHT`	`Ciphertext: NIOZSECPQETPGCGYMKQFE`