Practice Problem 1: Lorapians' Palindromic Language

While exploring their new habitat, Kiri discovered a new species of animals, called *Lorapians.* Curiously, Lorapians have their own language, called *Rar,* with a unique system of phonemes that allow for palindromic words and sentences. In fact, all the words that Kiri has heard so far appear to be palindromic, when written out in the script that Lorapians use (which is luckily the same as English).

To study this language further, Kiri would like to create a program to check if a given word is a palindrome or not. She needs your help to implement this program.

The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
static boolean isPalindrome(String s)
```

which returns a

```
boolean
```

Indicating whether `s` is a palindrome or not.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be one string, consisting of characters `a-z`, and of length between 1 and 100 (inclusive).

Output:

For each test case, the program will output "Yes" or "No".

Note:

We have provided a skeleton program that reads the number of test cases and the input strings one by one. It also prints the output based on the `isPalindrome()` 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 abracadabra kayak amanaplanacanalpanama	 No Yes Yes

Practice Problem 2: Ombratrees' Patterns

After a long day of training, Neteyam went for a walk around the nearby forest. While exploring, he stumbled upon a species of plants, called Ombratrees, that were growing in an interesting pattern. Neteyam noticed that the plants had self-arranged in a series of lines such that: if the first line has *N* plants, then the second line has ⌈***N/2***⌉ = ***ceil(N/2)*** plants (i.e., smallest integer that is larger than or equal to *N/2*), the third line has ⌈***N/3***⌉ plants, and so on until the last line with a single plant. Neteyam is intrigued and wants to study this pattern further. To assist him in this endeavor, he seeks your help to calculate the total number of plants in this pattern.

For example, if *N* = *10*, then the pattern goes:
⌈10/1⌉ = 10, ⌈10/2⌉ = 5, ⌈10/3⌉ = 4, ⌈10/4⌉ = 3, ⌈10/5⌉ = 2, ⌈10/6⌉ = 2, ⌈10/7⌉ = 2, ⌈10/8⌉ = 2, ⌈10/9⌉ = 2, ⌈10/10⌉ = 1, giving us a total of 33 plants.

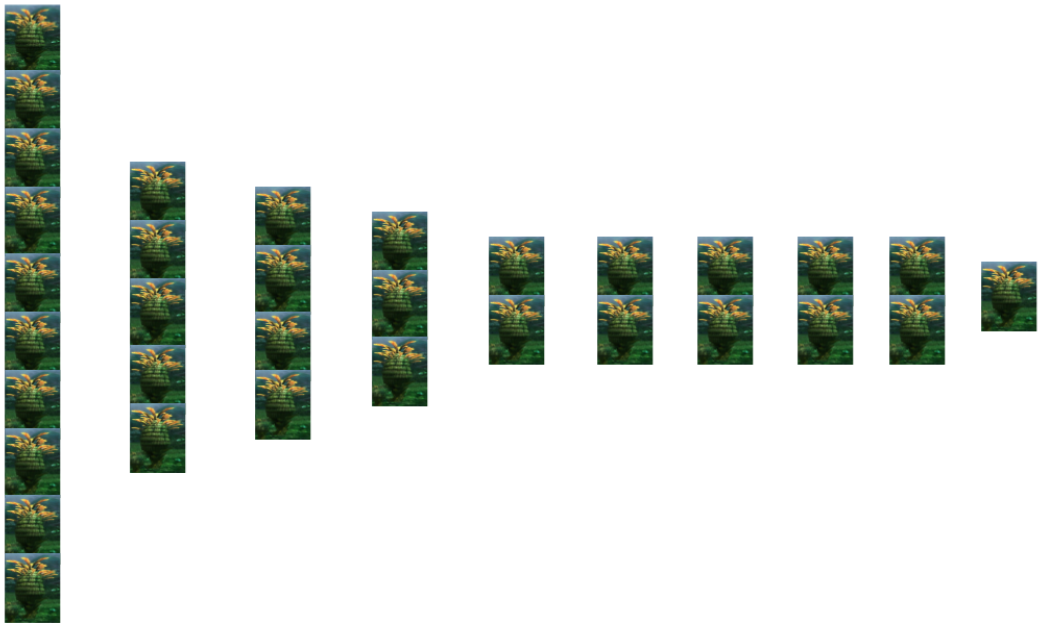
The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
static int countPlants(int N)
```

which returns an

```
int
```

that is the total number of plants.



Input/Output Format

Input:

The first line in the test data file contains the number of test cases. After that, for each test case there will be a single number (*N*) indicating the number of plants on the first line. You can assume that *1 <= N <= 10000.*

Output:

For each test case, the program will output the total number of plants.

Note:

We have provided a skeleton program that reads the number of test cases and the input numbers. It also prints the output based on the `countPlants()` 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 10 20 100	 33 80 251

Practice Problem 3: Stacked Floating Mountains

The floating mountains of Pandora present a challenge for the human scientists, especially geologists and physicists, who have been trying to understand how such structures could exist. While exploring the mountains, the scientists stumbled across interesting stacked floating mountain structures, where different mountains appeared stacked above one another, with the larger mountains being higher up in the stack. The scientists were able to calculate the size of each mountain, and they made an interesting observation: that the sizes of the mountains formed a (generalized) Fibonacci sequence.

A sequence of numbers: $x_1, x_2, ..., x_n$, is called a generalized Fibonacci sequence if, for all $i > 2$,

$$x_i = x_{i-1} + x_{i-2}$$

The standard Fibonacci sequence is simply a generalized Fibonacci sequence with $x_1 = x_2 = 1$.

An example of generalized Fibonacci sequence is:

2, 5, 7, 12, 19, … (5+2 = 7, 7+5 = 12, 12+7= 19, etc).

Your goal is to help the scientists verify this conjecture. Specifically, you are to write a program that, given a sequence of numbers, decides whether the sequence is a generalized Fibonacci sequence or not.

The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
boolean solveFibbonacciSimple(int [] sequence)
```

which returns a

```
boolean
```

Indicating whether the input is a generalized Fibonacci sequence or not.

Input/Output Format

Input:

The first line in the test data file contains the number of test cases, *n*. After that, each line contains one test case. The test case begins with the number of elements in the sequence, *k* (<= 100), and then we have *k* numbers which form the sequence. Assume all numbers are >= *0*, and that the numbers are all < *100000*.

Output:

For each test case, you are to output “YES” (if the sequence is a generalized Fibonacci sequence) or “NO” (if it is not).

Note:

We have provided a skeleton program that reads the number of test cases and the input numbers. It also prints the output based on the `solveFibbonacciSimple()` 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 6 1 1 2 3 5 8 7 1 2 2 4 6 10 16 4 2 10 12 22	 YES NO YES

Practice Problem 4: Na'vi Word Connections

Kiri has been exploring the origins of the languages spoken in different parts of Pandora, and trying to understand how different words may have originated. To do this, she would like to better understand the connections between different Na'vi words, and she would like your help with it.

Specifically, she has given you a list of Na'vi words, and you need to find the *shortest path* between two provided words. For this, you should create a graph where there is an edge between two words if either they differ in exactly one character and are of the same length, or if all the letters in the smaller word are found in the larger word (duplication doesn't matter). Then, given two specific words, you need to find the shortest path between those and return the path. If there are two paths, you should return the one that is lexicographically first. You can assume that all the words are of length 3 or 4.

For example, words "ala" and "ana" are of the same length, and differ in exactly one character, and hence there should be an edge between them. Similarly, all letters in "ana" are present in "naer", so there should be an edge between those two as well (even though "a" is present twice in the first word, and only once in the second word).

You will be given two words, **start_word** and **end_word,** and a list of Na'vi words, **listOfWords** (this list is already populated in the skeleton code provided to you). You need to find the shortest path between **start_word** and **end_word** in the graph defined by words.

The graph can be constructed as follows: iterate through each word in **listOfWords**, and for each word, compare it to all other words in words, and add an undirected edge between them if either of the conditions above is true. Once you have constructed the graph, you can use any shortest path algorithm to find the shortest path between **start_word** and **end_word**. If there are multiple shortest paths, return the lexicographically first one. For example, between two paths: "ala" → "alb" → "asb" (made-up words not in the list) and "ala" → "asa" → "asb", the first path is preferred because "alb" appears before "asa" in the lexicographical ordering.

The provided skeleton handles the input of the values and the output messages. You need to implement the method

```
ArrayList<String> findWordConnection(String start_word, String end_word)
```

which returns an

```
ArrayList<String>
```

containing the path (including start_word and end_word) if there is at least one, and null if there is no path.

Input/Output Format

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 *start_word* and *end_word.*

Output:

For each test case, you are to output “No path found” (if there is none) or the actual path.

Note:

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

Examples:

The input/output are 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:**
6 siha soha aaw nyer rux piru epay tyok tute ivu eanu amop	 Path: siha -> soha Path: aaw -> alaw -> ala -> ana -> naer -> nyer No path found between rux and piru. Path: epay -> epa -> ema -> oma -> kamo -> mok -> tok -> tyok No path found between tute and ivu. Path: eanu -> ana -> ala -> lamp -> amp -> amop