# Flower Wreaths

*You want to go to a flowers wreath competition but to participate you have to make at least 5 flower wreaths.*

You will be given **two sequences of integers, representing roses and lilies**. You need to start making wreaths **knowing that one wreath needs 15 flowers**. Your goal is to make **at least 5 flower wreaths**.

You will start crafting from the **last lilies** and the **first roses**. If the **sum** of their values is **equal** to **15** – **create one wreath** and **remove** them. If the sum is **bigger than 15**, just **decrease** the value of the **lilies** **by 2.** If the sum is **less than 15 you have to store them for later and remove them**. You need to **stop** combining when you have **no more roses or lilies**. In the end, if you have **any stored flowers you should make as many wreaths as you can with them**.

### Input

* On the **first line**, you will receive the integers representing the **lilies**, **separated** by **", "**.
* On the **second line**, you will receive the integers representing the **roses**, **separated** by "**,** ".

### Output

* Print whether you have succeeded in making **at least 5 wreaths**:
  + **"You made it, you are going to the competition with {count of wreaths} wreaths!"**
  + **"You didn't make it, you need {wreaths needed} wreaths more!"**

### Constraints

* All of the given numbers will be valid integers in the range **[0, 120]**.
* Don't have a situation with a negative number.

### Examples

|  |  |
| --- | --- |
| ****Input**** | ****Output**** |
| **10, 15, 2, 7, 9, 13**  **2, 10, 8, 12, 0, 5** | **You made it, you are going to the competition with 5 wreaths!** |
| ****Comment**** | |
| We start with the last lilies (13) and the first roses (2) -> 13 + 2 = 15 -> 15 = 15 So we create one wreath and remove them bouth.  Next we have 9 + 10 = 19 -> 19 > 15 so we decrease the lilies by 2 -> 7 + 10 = 17 and we decrease the liles by 2 -> 5 + 10 = 15 and we create one more wreath and remove them.  Next, we have 7 + 8 = 15. We create one more wreath and remove them.  Next, we have 2 + 12 = 14 -> 14 < 15 so we have to store theire sum for later and remove theme.  Next, we have 15 + 0 = 15 so we create one more wreath.  And last we have 10 + 5 = 15, we create one more wreath and stop mixing because we don’t have any flowers left.  Now we have a total of 5 wreaths and we also have 14 flowers left but we cant create a wreath because 14 < 15. | |

|  |  |
| --- | --- |
| ****Input**** | ****Output**** |
| **10, 5, 3, 7, 8**  **5, 10, 8, 7, 6** | **You didn't make, you need 1 wreaths more!** |
| ****Comment**** | |
| We start with 8 + 5 = 13 -> 13 < 15 -> we have to store their sum for later and remove them.  Next, we have 7 + 10 = 17 -> we decrease the lilies by 2 -> 5 + 10 = 15 -> 15 = 15 and we create one wreath.  Next, we have 3 + 8 = 11 -> 11 < 15 -> we store their sum for later and remove them.  Next, we have 5 +7 = 12 -> we store their sum for later and remove them.  Next, we have 10 + 6 = 16 -> 16 > 15 we decrease the lilies by 2 -> 8 + 6 = 14 and we store their sum for later and remove them.  We stop crafting because we don’t have any flowers left and we have 1 wreath and 50 stored flowers. We create 3 more wreaths because 3 \* 15 = 45 -> 50 – 45 = 5 -> 5 < 15. | |

# Bee

You will be given an integer **n** for the **size** of the bee territory with a **square** shape. On the next **n** lines, you will receive the **rows** of the territory. The bee will be placed in a **random position**, marked with the letter '**B**'. On random positions, there will be flowers, marked with **'f'**. There may also be a **bonus** on the territory. There will always be only one bonus. It will be **marked** with the **letter** - '**O**'. **All of the empty positions** will be marked with **'.'**.

Each turn, you will be given a **command** for the **bee’s movement**.

The commands will be: "**up**", "**down**", "**left**", "**right**", "**End**".

If the bee **moves** to a **flower**, it pollinates the flower and increases pollinated flowers with one.

If it goes to a **bonus**, the bee gets a bonus one moves forward and then the bonus **disappears**. If the bee **goes out** she can't return back and the program ends. If the bee receives the "**End**" command the program ends. The bee needs **at least** **5 pollinated flowers**.

### Input

* On the first line, you are given the integer **n** – the size of the **square** matrix.
* The **next n lines** hold the values for every **row**.
* On each of the next lines, until you receive the "**End**" command, you will receive a move command.

### Output

* On the first line:
  + If the bee goes out of its territory print: **"The bee got lost!"**
* On the second line:
  + If the bee couldn’t pollinate enough flowers print: **"The bee couldn't pollinate the flowers, she needed {needed} flowers more"**
  + If the bee successfully pollinated enough flowers print: **"Great job, the bee manage to pollinate {polinationed flowers} flowers!"**
* At the end print the matrix.

### Constraints

* The size of the **square** matrix will be between **[2…10].**
* There will **always** be **0** or **1** bonus, marked with - '**O**'.
* The bee position will be marked with '**B**'.
* There won't be a case where a bonus moves the bee out of its territory.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 5  Bff..  ..O..  f.f.f  .....  fffff  right  right  down  left  left  down  down  right  down | The bee got lost!  Great job, the bee manage to pollinate 6 flowers!  .....  .....  ....f  .....  ..fff | 1) right 2) right 3) down 5) left  .Bf.. ..B.. ..... .....  ..O.. ..O.. ..... .....  f.f.f f.f.f f.B.f fB..f  ..... ..... ..... .....  fffff fffff fffff fffff  …  2) pollinate a flower: 'f' (0, 1)  3) step on a bonus: 'O' (1, 2) and make one more step down to: 'f' (2, 2)  4) the bee is on (4, 1), the next command she receives is down and she goes out of the field and the program ends. |
| 4  ....  .O..  ff..  f.B.  left  left  up  right  up  End | The bee couldn't pollinate the flowers, she needed 2 flowers more  .B..  ....  ....  .... |  |

# Vet Clinic

## Preparation

Download the skeleton provided in Judge. **Do not** change the **packages**!

**Pay attention to name the package parking, all the classes, their fields, and methods the same way they are presented in the following document. It is also important to keep the project structure as described.**

## Problem description

Your task is to create a repository, which stores items by creating the classes described below.

First, write a Java class **Pet** with the following fields:

* **name: String**
* **age: int**
* **owner: String**

The class **constructor** should receive a **name, age,** and **owner.** You need to create the appropriate **getters and setters**. The class should override the **toString()** method in the following format:

**"{name} {age} ({owner})"**

**Next**, write a Java class **Clinic** that has **data** (a collection, which stores the Pets). All entities inside the repository have the **same fields**. Also, the **Clinic** class should have those fields:

* **capacity: int**

The class **constructor** should receive **capacity**, also it should initialize the **data** with a new instance of the collection**.** Implement the following features:

* Field **data** – **List** that holds added pets
* Method add(Pet pet) – **adds** an **entity** to the data **if** **there** **is** an **empty cell** for the pet.
* Method remove(String name) – removes the pet by **given name,** if such **exists**, and **returns boolean**.
* Method **getPet(String name, String owner)** – returns the pet with the **given name** and **owner** or **null if no such pet exists**.
* Method getOldestPet() – returns the oldest Pet.
* Getter getCount – **returns** the **number** of pets.
* **getStatistics()** – **returns** a **String** in the following **format**:
  + **"The clinic has the following patients:  
    {name} {owner}  
    {name} {owner}**

**(…)**"

## Constraints

* The **combinations** of **names** and **owners** will **always be unique**.
* The **age** of the pets will always be **positive**.

## Examples

This is an example of how the **Clinic** class is **intended to be used**.

|  |
| --- |
| Sample code usage |
| // Initialize the repository  Clinic clinic = new Clinic(20);  // Initialize entity  Pet dog = new Pet("Ellias", 5, "Tim");  // Print Pet  System.out.println(dog); // Ellias 5 (Tim)  // Add Pet  clinic.add(dog);  // Remove Pet  System.out.println(clinic.remove("Ellias")); // true  System.out.println(clinic.remove("Pufa")); // false  Pet cat = new Pet("Bella", 2, "Mia");  Pet bunny = new Pet("Zak", 4, "Jon");  clinic.add(cat);  clinic.add(bunny);  // Get Oldest Pet  Pet oldestPet = clinic.getOldestPet();  System.out.println(oldestPet); // Zak 4 (Jon)  // Get Pet  Pet pet = clinic.getPet("Bella", "Mia");  System.out.println(pet); // Bella 2 (Mia)  // Count  System.out.println(clinic.getCount()); // 2  // Get Statistics  System.out.println(clinic.getStatistics());  //The clinic has the following patients:  //Bella Mia  //Zak Jon |

## Submission

Zip all the files in the project folder except the **bin** and **obj** folders

Submit a **single .zip file**, containing **vetClinic package, with the classes inside (Pet, Clinic, and the Main class)**, there is no specific content required inside the Main class e. g. you can do any kind of local testing of your program there. However, there should be a **main(String[] args)** method inside.