# Bouquets



*You want to go to a bouquets competition but to participate you have to make at least 5* *bouquets.*

You will be given **two sequences of integers, representing daffodils and tulips.** You need to start making bouquets **knowing that one bouquet needs 15 flowers**. Your goal is to make **at least 5 bouquets.**

You will start crafting from the **last tulips** and the **first daffodils**. If the **sum** of their values is **equal** to **15** – **create one bouquet** and **remove** them. While the sum is **bigger than 15**, keep **decreasing** the value of the **tulips 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 daffodils or tulips**. In the end, if you have **any stored flowers you should make as many bouquets as you can with them.**

### Input

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

### Output

* Print whether you have succeeded in making **at least 5 bouquets**:
  + **"You made it! You go to the competition with {count of bouquets} bouquets!"**
  + **"You failed... You need more {number} bouquets."**

### 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 go to the competition with 5 bouquets!** |
| ****Comment**** | |
| We start with the last tulips (13) and the first daffodils (2) -> 13 + 2 = 15 -> 15 = 15 So we create one bouquet and remove them bouth.  Next we have 9 + 10 = 19 -> 19 > 15 so we decrease the tulips by 2 -> 7 + 10 = 17 and we decrease the tulips by 2 -> 5 + 10 = 15 and we create one more bouquet and remove them.  Next, we have 7 + 8 = 15. We create one more bouquet 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 bouquet.  And last we have 10 + 5 = 15, we create one more bouquet and stop mixing because we don’t have any flowers left.  Now we have a total of 5 bouquets and we also have 14 flowers left but we cant create a bouquet because 14 < 15. | |

|  |  |
| --- | --- |
| ****Input**** | ****Output**** |
| **10, 5, 3, 7, 8**  **5, 10, 8, 7, 6** | **You failed... You need more 1 bouquets.** |
| ****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 tulips by 2 -> 5 + 10 = 15 -> 15 = 15 and we create one bouquet.  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 tulips 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 bouquet and 50 stored flowers. We create 3 more bouquets because 3 \* 15 = 45 -> 50 – 45 = 5 -> 5 < 15. | |

# Cooking journey



*You successfully started your cooking journey, so now you need to sell the products from your basket in the pastry shop* *to collect your price.*

You will be given an integer **n** for the **size** of the pastry shop with a **square** shape. On the next **n** lines, you will receive the **rows** of the pastry shop. You will be placed in a **random position**, marked with the letter '**S**'. On random positions, there will be clients, marked with a **single digit**. There **may** also be **pillars**. Their **count** will be either **0** or **2** and they are **marked** with the **letter** - '**P**'. **All of the empty positions** will be marked with **'-'**.

Each turn, you will be given **commands** for **your movement**. Move commands will be: "**up**", "**down**", "**left**", "**right**". If you **move** to a **client**, you **collect** **the price** **equal** to the **digit** **there** and the client **disappears**. If you move to a **pillar**, you move on to the **position** of the **other pillar** and then **both** pillars **disappear**. If you **go** **out** of the pastry shop, you **disappear** from the pastry shop and you are out of there. You need **at least** **50 dollars** to rent your Pastry shop.

When **you are out of the pastry shop** **or you collect enough money,** the program **ends**.

### 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, you will get a move command.

### Output

* On the first line:
  + If the player goes to the void, print: "**Bad news! You are out of the pastry shop.**"
  + If the player collects enough star power, print: "**Good news! You succeeded in collecting enough money!**"
* On the second line print, all-star power collected: "**Money: {money}**"
* At the end print the matrix.

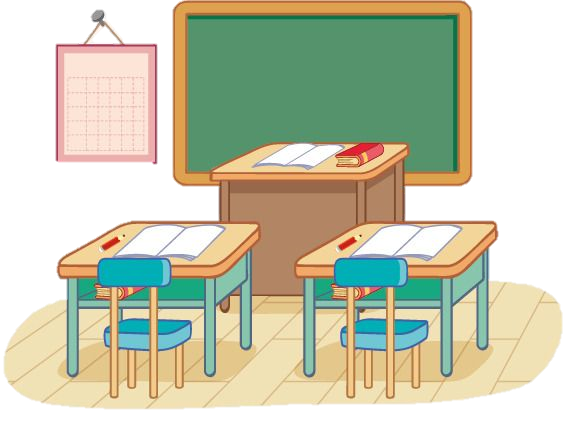
### Constraints

* The size of the **square** matrix will be between **[2…10].**
* There will **always** be **0** or **2** pillars, marked with the **letter** - '**P**'.
* Your position will be marked with '**S**'.
* You will **always** go out of the pastry shop or collect enough money.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 5  SP---  -----  -----  -----  ----P  right  right | Bad news, you are out of the pastry shop.  Money: 0  -----  -----  -----  -----  ----- | The first command is right. You move to **one of the pillars** and then **appear** on the other side of it (4,4).  The pastry shop looks like this after the first command:  -----  -----  -----  -----  ----S  The second command is right. You go to the pastry shop. |
| 6  S98---  99----  666666  ------  --77--  -6-6-6  right  right  down  left  left  down  right  right | Good news! You succeeded in collecting enough money!  Money: 53  ------  ------  --S666  ------  --77--  -6-6-6 | Here we have **no** pillars and a pastry shop rich in clients.  You manage to collect **enough** money **without** **going out** of the pastry shop.  The clients you have sealed food to have disappeared and we can see where you were when you collected the last needed money (2,2). |

# University



## Preparation

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

**Pay attention to the name of the package (university), 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 above.**

**Problem Description**

Your task is to create a repository that stores departments by creating the classes described below.

### Student

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

* **firstName: String**
* **lastName: String**
* **bestSubject: String**

The class **constructor** should receive (**firstName, lastName,** and **bestSubject**).

The class also should have the methods:

* getFirstName()
* getLastName()
* getBestSubject()
* Override the **toString()** method in the following format:

**"Student: {firstName} {lastName}, {bestSubject}"**

### University

**Next**, write a **Java** class University that has **students** (a collection that stores the entity **Student**). All entities inside the repository have the **same public fields**. Also, the University class should have those fields:

* **capacity:** int
* **students:** List<Student> **-** holds all added students in the university

The class **constructor** should receive (**capacity**), also it should initialize the **students** with a new instance of the collection**.**

Implement the following features:

* getCapacity()
* getStudents()
* getStudentCount() method– **returns** the **number** of **students in the university**
* registerStudent(Student student) method – **adds** an **entity** to the students **if** **there** **is** **room** for it
  + Returns **"Added student {firstName} {lastName}"** if the student is **successfully added**
  + Returns **"Student is already in the university"** if the student is already in the university
  + Returns **"No seats in the university"** if the university is full
* dismissStudent(Student student) method – **removes the student**
  + Returns **"Student not found"** if the student is not in the university
* **getStudent(String firstName, String lastName)** method **-** returns the student with the **given names**.
* **getStatistics()** – **returns** a **String** in the following **format**:
  + **"==Student: First Name = {firstName}, Last Name = {lastName}, Best Subject = {bestSubject}   
     ==Student: First Name = {firstName}, Last Name = {lastName}, Best Subject = {bestSubject}**

**(…)**"

## Constraints

* The **combinations** of **names** will **always be unique**.
* The **capacity** will always be **a positive** **number**.

## Examples

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

|  |
| --- |
| Sample code usage |
| *// Initialize the repository* University university = new University(10); *// Initialize entities* Student student = new Student("John", "Smith", "Astrology");  Student studentTwo = new Student("Anna", "Cameron", "Geometry");  Student studentThree = new Student("Samy", "Johnson", "Algebra");  Student studentFour = new Student("Rihanna", "Fenty", "Music");  Student studentFive = new Student("Ellie", "Goulding", "Music"); *// Print Student* System.*out*.println(student);  *// Student: John Smith, Astrology*  *// Register Student* String register = university.registerStudent(student);  System.out.println(university.getCapacity()); // 10  System.*out*.println(register); *// Added student John Smith* String registerTwo = university.registerStudent(studentTwo);  String registerThree = university.registerStudent(studentThree);  String registerFour = university.registerStudent(studentFour); *// Dismiss Student* String dismissed = university.dismissStudent(student);  System.*out*.println(dismissed); *// Removed student John Smith* String dismissedTwo = university.dismissStudent(studentFive);  System.*out*.println(dismissedTwo); *// Student not found // Get Student* System.*out*.println(university.getStudent("Rihanna", "Fenty")); *// Student: Rihanna Fenty, Music*  *System.out.println(university.getStudentCount()); // 3*  *System.out.println(university.getStatistics());*  *//==Student: First Name = Anna, Last Name = Cameron, Best Subject = Geometry*  *//==Student: First Name = Samy, Last Name = Johnson, Best Subject = Algebra*  *//==Student: First Name = Rihanna, Last Name = Fenty, Best Subject = Music* |

## Submission

Submit a **single .zip file**, containing the **university package, with the classes inside (Student, University, 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.