# Pastry shop



*You are asked to cook pastries and cakes for the near pastry shop because your recipes are so great. You need to mix liquids with ingredients to cook the required delicacies.*

First, you will be given **a sequence of integers, representing liquids**. Afterward, you will be given another **sequence of integers representing ingredients**.

You need to start from the **first liquid** and try to mix it with the **last ingredient.** If the **sum** of their values is **equal** to **any of the items in the table below** – **cook the food corresponding** to the **value** and **remove** **both** the **liquid** and the **ingredient**. Otherwise, **remove only the liquid** and **increase** the **value** of the **ingredient by 3**. You need to **stop** combining when you have **no more liquids** or **ingredients**.

|  |  |
| --- | --- |
| **Food** | **Value needed** |
| Biscuit | 25 |
| Cake | 50 |
| Pastry | 75 |
| Pie | 100 |

### Input

* On the **first line**, you will receive the integers representing the **liquids**, **separated** by a **single space**.
* On the **second line**, you will receive the integers representing the **ingredients**, **separated** by a **single space**.

### Output

* On the **first** line of output print one of the following outputs:
  + "**Great! You succeeded in cooking all the food!**" -if you have at least

**one of each** of the **foods**, after completing combining.

* + "**What a pity! You didn't have enough materials to cook everything.**" – if you **did not** collect **one of each** of the **foods**, after completing combining.
* On the **second** line - print all liquids you have left:
  + If there are no liquids: "**Liquids left: none**"
  + If there are liquids: "**Liquids left: {liquid1}, {liquid2}, {liquid3},** **(…)**"
* On the **third** line - print all physical materials you have left:
  + If there are no items: "**Ingredients** **left: none**"
  + If there are items: "**Ingredients** **left: {ingredient}, {ingredient}, {ingredient},** **(…)"**
* Then**,** you need to print **all** AdvancedMaterials and the **amount you have of them**, ordered:
  + **"Biscuit: {amount}"**
  + **"Cake: {amount}"**
  + **"Pie: {amount}"**
  + **"Pastry: {amount}"**

### Constraints

* All of the given numbers will be valid integers in the range **[0, 100]**.
* Advanced materials **can be** crafted more than once.

### Examples

|  |  |  |
| --- | --- | --- |
| ****Input**** | ****Output**** | ****Comment**** |
| **1 25 50 50**  **50 25 25 24** | **Great! You succeeded in cooking all the food!**  **Liquids left: none**  **Ingredients left: none**  **Biscuit: 1**  **Cake: 1**  **Pie: 1**  **Pastry: 1** | The first pair is the **first liquid** with a value of 1 and the **last ingredient** of value 24, their **sum** is 25, so we **cook** Biscuit. Then we have a **sum** of 50, we **cook** Cake. After that we have a **sum** of 75, we **cook** Pastry. Next, we have a **sum** of 100, so we **craft** Pie. We have **no left liquids and/or ingredients**, so we **stop** trying to cook foods, but we **have enough** of them to **give to the pastry shop.** |
| **10 20 30 40 50**  **50 40 30 30 15** | **What a pity! You didn't have enough materials to cook everything.**  **Liquids left: none**  **Ingredients left: 39, 40, 50**  **Biscuit: 1**  **Cake: 1**  **Pie: 0**  **Pastry: 0** | **First, we take **the first given liquid** and **the last ingredient**, their **sum** is 25 and we **cook** Biscuit, **removing** **both** of them from the collections. Then, we take the **next pair** and their **sum** is 50, **cooking** Cake and again – **removing both** the liquid and the ingredient. Next, we take the **next pair** and their **sum** is 60, so we **remove the liquid** and **increase** the **ingredient's** value by 3. The next 2 pairs follow **the same scenario**, so we end up with **not enough** materials for all the food, **no liquids left,** and **some** **ingredients**, one of which is **39** (**originally 30**, **increased** its value **three** times).** |

# Formula One



You will be given an integer N for the size of the square matrix and then an integer for the count of commands. On the next **n** lines, you will receive the rows of the matrix. The player starts at a random position (the player is marked with **"P"**) and **all of the empty slots** will be filled with **"."** (dot). The goal is to reach the finish mark which will be marked with an **"F"**. On the field, there can also be bonuses and traps. Bonuses are marked with **"B"** and traps are marked with **"T"**.

Each turn you will be given commands for **the player’s movement.** If the player **goes** **out** of the matrix, he comes in from **the other side**. For example, if the player is on 0, 0 and the next command is left, he goes to the last spot on the first row.

If the player steps on a **bonus**, he should move another step forward in the direction he is going. If the player steps on a **trap**, he should move a step backward.

When the player reaches the **finish mark** or the **count of commands is reached,** the game ends.

### Input

* On the first line, you are given the integer N – the size of the square matrix.
* On the second you are given the count of commands.
* The next N lines hold the values for every row.
* On each of the next lines, you will get commands for movement directions.

### Output

* If the player reaches the finish mark, print:
  + **"Well done, the player won first place!"**
* If the player reaches the commands count and hasn’t reached the finish mark print:
  + **"Oh no, the player got lost on the track!"**
* At the end print the matrix.

### Constraints

* The size of the matrix will be between **[2…20].**
* The players will always be indicated with **"P".**
* If the player steps on the finish mark **at the same time** as his last command, he **wins** the game.
* When the final matrix is printed and the vehicle has been completed successfully, you must replace the "**F**" with the "**P**".
* Commands will be in the format of **up**, **down**, **left** or **right**.
* There won't be a case where you bypass the finish while you are on a trap or a bonus.
* A trap will never place you into a bonus or another trap and a bonus will never place you into a trap or another bonus.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 5  3  .....  .P...  .B...  ..T..  .F...  down  right  down | Well done, the player won first place!  .....  .....  .B...  ..T..  .P... | The first command is **down** so **P** moves down. On turn 1, the player steps on a bonus and does an additional step. On turn 2, the player steps on a trap and goes a step back. After each turn the field is:  1 2 3  ..... ..... .....  ..... ..... .....  .B... .B... .B...  .PT.. .PT.. ..T..  .F... .F... .P...  In turn, 3 **P** reaches the finish 'F' and wins the game. |
| 4  3  ....  .P.B  ..T.  ...F  up  up  left | Oh no, the player got lost on the track!  ....  ...B  ..T.  P..F | The first command is **up** so **P** moves up and the new player position is **(0,1).** The next command is **up** again, so the player goes **out** **of the matrix** and **comes back from the other side at (3,1).** Then the command is **left**, so the end position is **(3,0)** which is not the finish mark, so the **player has lost**. |

# Café

# 

*Now that you successfully saved money for your Café, you need to recruit some employees to work there. You are You should build a system for that.*

## Preparation

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

**Pay attention to the name of the package cafe, 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 cafe, which stores employees by creating the classes described below.

First, write a Java class **Employee** with the following properties:

* **name: String**
* **age: int**
* **country: String**

The class **constructor** should receive **name, age** and **country** and override the **ToString()** method in the following format:

**"Employee: {name}, {age} from {country}"**

**Next**, write a Java class **Cafe** that has **employees** (a collection, which stores the entity **Employee**). All entities inside the repository have the **same properties**. Also, the Cafe class should have those properties:

* **name: String**
* **capacity: int**

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

* Field **employees** – **List** that holds added Employees
* Method addEmployee(Employee employee) – **adds** an **entity** to the data **if** **there** **is** **room** for him/her.
* Method removeEmployee(String name) – removes an employee by **given name,** if such **exists**, and **returns** a **bool**.
* Method getOldestEmployee() – returns the **oldest** employee.
* Method **getEmployee(string name)** – returns the employee with the **given name**.
* Getter getCount() – **returns** the **number** of employees.
* **report()** – **returns** a **string** in the following **format**:
  + **"Employees working at Cafe {cafeName}:  
    {Employee1}  
    {Employee2}  
    (…)**"

## Constraints

* The **names** of the employees will be **always unique**.
* The **age** of the employees will always be with **positive values**.
* You will always have an employee added before receiving methods manipulating the Employees.

## Examples

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

|  |
| --- |
| Sample code usage |
| //Initialize the repository  Cafe cafe = new Cafe("Costa", 15);  //Initialize entity  Employee employee = new Employee("Alexander", 30, "Bulgaria");  //Print Employee  System.*out*.println(employee); // Employee: Alexander, 30 from Bulgaria  //Add Employee  cafe.addEmployee(employee);  //Remove Employee  System.*out*.println(cafe.removeEmployee("Employee")); //false  Employee secondEmployee = new Employee("Sara", 24, "UK");  Employee thirdEmployee = new Employee("Anna", 22, "Germany");  //Add Employee  cafe.addEmployee(secondEmployee);  cafe.addEmployee(thirdEmployee);  Employee oldestEmployee = cafe.getOldestEmployee();  Employee employeeStephen = cafe.getEmployee("Sara");  System.*out*.println(oldestEmployee); // Employee: Alexander, 30 from Bulgaria System.*out*.println(employeeStephen); //Employee: Sara, 24 from UK  System.*out*.println(cafe.getCount()); // 3  System.out.println(cafe.removeEmployee("Anna")); // true  System.*out*.println(cafe.report());  //Employees working at Cafe Costa:  //Employee: Alexander, 30 from Bulgaria  //Employee: Sara, 24 from UK |

## Submission

Submit a **single .zip file**, containing the **cafe package, with the classes inside (Employee, Cafe, 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.