# Cooking



*You are asked to cook baked foods for the near bakery 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** |
| Bread | 25 |
| Cake | 50 |
| Pastry | 75 |
| Fruit 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:
  + "**Wohoo! You succeeded in cooking all the food!**" - if you have at least

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

* + "**Ugh, 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 **alphabetically**:
  + **"Bread: {amount}"**
  + **"Cake: {amount}"**
  + **"Fruit 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** | **Wohoo! You succeeded in cooking all the food!**  **Liquids left: none**  **Ingredients left: none**  **Bread: 1**  **Cake: 1**  **Fruit pie: 1**  **Pastry: 1** | The first pair is the **first liquid** with the value of 1 and the **last ingredient** of value 24, their **sum** is 25, so we **cook** Bread. 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** Fruit 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 bakery.** |
| **10 20 30 40 50**  **50 40 30 30 15** | **Ugh, what a pity! You didn't have enough materials to cook everything.**  **Liquids left: none**  **Ingredients left: 39, 40, 50**  **Bread: 1**  **Cake: 1**  **Fruit pie: 0**  **Pastry: 0** | **First, we take **the first given liquid** and **the last ingredient**, their **sum** is 25 and we **cook** Bread, **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).** |

# Selling



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

You will be given an integer **n** for the **size** of the bakery with a **square** shape. On the next **n** lines, you will receive the **rows** of the bakery. 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** - '**O**'. **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 bakery, you **disappear** from the bakery and you are out of there. You need **at least** **50 dollars** to rent your Bakery

When **you are out of the bakery 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 bakery.**"
  + 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** - '**O**'.
* Your position will be marked with '**S**'.
* You will **always** go out of the bakery or collect enough money.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 5  SO---  -----  -----  -----  ----O  right  right | Bad news, you are out of the bakery.  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 bakery looks like this after the first command:  -----  -----  -----  -----  ----S  The second command is right. You go out of the bakery. |
| 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 bakery rich in clients.  You manage to collect **enough** money **without** **going out** of the bakery.  The clients you have sold food to have disappeared and we can see where you were when you collected the last needed money (2,2). |

# Openning

# 

*Now that you successfully saved money for your Bakery, 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 bakery, 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 bakery, 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} ({country})"**

**Next**, write a Java class **Bakery** that has **employees** (a collection, which stores the entity **Employee**). All entities inside the repository have the **same properties**. Also, the Bakery 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 add(Employee employee) – **adds** an **entity** to the data **if** **there** **is** **room** for him/her.
* Method remove(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 Bakery {bakeryName}:  
    {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 Space Station’s Employees.

## Examples

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

|  |
| --- |
| Sample code usage |
| //Initialize the repository  Bakery bakery = new Bakery("Barny", 10);  //Initialize entity  Employee employee = new Employee("Stephen", 40, "Bulgaria");  //Print Employee  System.*out*.println(employee); //Employee: Stephen, 40 (Bulgaria)  //Add Employee  bakery.add(employee);  //Remove Employee  System.*out*.println(bakery.remove("Employee name")); //false  Employee secondEmployee = new Employee("Mark", 34, "UK");  //Add Employee  bakery.add(secondEmployee);  Employee oldestEmployee = bakery.getOldestEmployee(); // Employee with name Stephen  Employee employeeStephen = bakery.getEmployee("Stephen"); // Employee with name Stephen  System.*out*.println(oldestEmployee); //Employee: Stephen, 40 (Bulgaria)  System.*out*.println(employeeStephen); //Employee: Stephen, 40 (Bulgaria)  System.*out*.println(bakery.getCount()); //2  System.*out*.println(bakery.report());  //Employees working at Bakery Barny:  //Employee: Stephen, 40 (Bulgaria)  //Employee: Mark, 34 (UK) |

## Submission

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