# Exam Preparation – 17 June 2022

## Cooking

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/2704#0**](https://judge.softuni.org/Contests/Practice/Index/2704#0)

*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**. Afterwards, 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 ouputs:
  + "**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 value of 1 and the **last ingredient** of value 24, their **sum** is 25, so we **cook** Bread. Then we have **sum** of 50, we **cook** Cake. After that we have **sum** of 75, we **cook** Pastry. Next we have **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 them 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).** |

## Mouse and Cheese

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/3203#1**](https://judge.softuni.org/Contests/Practice/Index/3203#1)

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

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

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

If the mouse **moves** to **cheese**, it eats the cheese and increases the cheese it has eaten by one.

If it goes to a **bonus**, the mouse gets a bonus one move forward and then the bonus **disappears**. If the mouse **goes out** she can't return and the program ends. If the mouse receives "**end**" command the program ends. The mouse needs **at least 5 eaten cheeses**.

### 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 "**end**" command, you will receive a move command.

### Output

* On the first line:
  + If the mouse goes out of its territory print: **"Where is the mouse?"**.
* On the second line:
  + If the mouse couldn’t eat enough cheeses print: **"The mouse couldn't eat the cheeses, she needed {needed} cheeses more."**.
  + If the mouse has successfully eaten enough cheeses print: **"Great job, the mouse is fed {eaten cheeses} cheeses!"**.
* At the end print the matrix.

### Constraints

* The size of the **square** matrix will be between **[2…10]**.
* There will always be only one bonus, marked with '**B**.
* The mouse position will be marked with '**M**'.
* There won't be a case where a bonus moves the mouse out of its territory.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 3  M--  ccc  ---  right  right  down  down  left  end | The mouse couldn't eat the cheeses, she needed 4 cheeses more.  ---  cc-  -M- | 1) right 2) right 3) down 5) down  -M- --M --- ---  ccc ccc ccM cc-  --- --- --- --M    6) left  ---  cc-  -M- |
| 5  Mcc--  --B--  c-c-c  -----  ccccc  right  down  left  down  right  down  left  left  end | Where is the mouse?  The mouse couldn't eat the cheeses, she needed 3 cheeses more.  --c--  --B--  --c-c  -----  ccccc |  |

## University

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/2826#2**](https://judge.softuni.org/Contests/Practice/Index/2826#2)

## Preparation

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

**Pay attention to name the package (university), all the classes, their fields and methods exactly 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 which 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 which 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 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 **single .zip file**, containing **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 you program there. However there should be **main(String[] args)** method inside.