# Exam Preparation – 9 February 2023

## Pastry Shop

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/3018#0**](https://judge.softuni.org/Contests/Practice/Index/3018#0)

*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).** |

## Snake

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/2471#1**](https://judge.softuni.org/Contests/Practice/Index/2471#1)

*Everyone remembers the old snake games. Now is the time to create our own snake game.*

You will be given an integer **n** for the **size** of the territory with a **square** shape. On the next **n** lines, you will receive the **rows** of the territory. The snake will be placed in a **random position**, marked with the letter '**S**'. There will also be food on random positions, marked with **'\*'**. The territory may have a **lair**. The lair will have two burrows **marked** with the **letter** - '**B**'. **All of the empty positions** will be marked with **'-'**.

Each turn, you will be given a **command** for the **snake’s movement**. When the snake moves it leaves a trail marked with **'.'**

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

If the snake **moves** to **food**, it will eat the food, which will increase food quantity with one.

If it goes inside a **burrow**, it **goes out** on the **position** of the **other burrow** and then **both** burrows **disappear**. If the snake **goes out** of its territory, the game is over. The snake needs **at least** **10 food quantities** to be fed.

If **the snake** **goes outside of its territory or has eaten enough food**, the game should **end**.

### 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 snake goes out of its territory, print: "**Game over!**"
  + If the snake eats enough food, print: "**You won! You fed the snake.**"
* On the second line print, all food is eaten: "**Food eaten: {food quantity}**"
* 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** burrows, marked with - '**B**'.
* The snake position will be marked with '**S**'.
* The snake will **always** either go out of its territory or eat enough food.
* There will be no case in which the snake will go through itself.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 6  -----S  ----B-  ------  ------  --B---  --\*---  left  down  down  down  left | Game over!  Food eaten: 1  ----..  ----.-  ------  ------  --.---  --.--- | 1) left 2) down 3) down 5) down  ----S. ----.. ----.. ----..  ----B- ----.- ----.- ----.-  ------ ------ ------ ------  ------ ------ ------ ------  --B--- --S--- --.--- --.---  --\*--- --\*--- --S--- --.---  3) eat the food: '\*' (5, 2)  5) the snake goes out of its territory and the program ends |
| 7  --\*\*\*S-  --\*----  --\*\*\*--  ---\*\*--  ---\*---  ---\*---  ---\*---  left  left  left  down  down  right  right  down  left  down | You won! You fed the snake.  Food eaten: 10  --....-  --.----  --...--  ---..--  ---S---  ---\*---  ---\*--- |  |

1. **Christmas**

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/1983#2**](https://judge.softuni.org/Contests/Practice/Index/1983#2)

*Can you imagine Santa without his special bag that carries presents for so many children? Neither can we. So let's have a quick sneak peek inside it and help Santa rearrange it.*

## Preparation

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

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

### Present

First, write a Java class Present with the following fields:

* **name: String**
* **weight: double**
* **gender: String**

The class **constructor** should receive (**name, weight, and gender**).

The class also should have the methods:

* getName()
* getWeight()
* getGender()
* Override the **toString()** method in the following format:

**"Present {name} ({weight}) for a {gender}"**

**Note: Format the weight to the second digit after the decimal point!**

### Bag

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

* **color:** String
* **capacity:** int
* **data:** List<Present> **-** holds all added presents in the bag

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

Implement the following features:

* getColor()
* getCapacity()
* count() method– **returns** the **number** of presents
* add(Present present) method – **adds** an **entity** to the data **if** **there** **is** **room** for it
* remove(String name) method – removes a present by **given name,** if such **exists**,   
  and **returns boolean**
* heaviestPresent() method – returns the **heaviest** present
* getPresent(String name) method – returns the present with the **given name**
* report() method – **returns** a **string** in the following **format** (print the presents in order of appearance):
  + **"{color of Bag} bag contains:  
    {Present1}  
    {Present2}  
    (…)**"

## Constraints

* The **names** of the presents will be **always unique**.
* The **weights** of the presents will always be with **positive values**.
* You will always have a present added before receiving methods.

## Examples

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

|  |
| --- |
| Sample code usage |
| *// Initialize the repository* Bag bag = **new** Bag(**"black"**, 10);  *// Initialize entity* Present present = **new** Present(**"Doll"**, 0.4, **"girl"**);  *// Print Present* System.***out***.println(present); *// Present Doll (0.40) for a girl*  *// Add Present* bag.add(present);  *// Remove Present* bag.remove(**"Toy"**); *// false*  Present secondPresent = **new** Present(**"Train"**, 2, **"boy"**); *// Add Present* bag.add(secondPresent);  Present heaviestPresent = bag.heaviestPresent();  System.***out***.println(heaviestPresent);  *// Present Train (2.00) for a boy*  Present p = bag.getPresent(**"Doll"**);  System.***out***.println(p);  *// Present Doll (0.40) for a girl*  System.***out***.println(bag.count()); *// 2* System.***out***.println(bag.report());  *// Black bag contains: // Present Doll (0.40) for a girl // Present Train (2.00) for a boy* |

## Submission

Submit a **single .zip file**, containing a **christmas package, with the classes inside (Present, Bag, 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.