# Exam Preparation – 17 October 2022

## Lootbox

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/2036#0**](https://judge.softuni.org/Contests/Practice/Index/2036#0)

[**https://softuni.bg/trainings/resources/video/76483/video-17-october-2022-dimo-georgiev-java-advanced-september-2022/3844**](https://softuni.bg/trainings/resources/video/76483/video-17-october-2022-dimo-georgiev-java-advanced-september-2022/3844)

*You are tired of being the only one on your team without cool items, so you decided to buy some loot boxes which have a chance to drop some cool items.*

Every purchase gives you two loot boxes and they are represented as a sequence of integers. First, you will be given **a sequence of integers, representing the first loot box**. Afterward, you will be given another **sequence of integers representing the second loot box**.

You need to start from the **first item** in the first box and **sum** it up with the last item in the second box. If the **sum** of their values is **an even number,** add the **summed** item to **your collection of claimed items** and **remove** them **both** from the boxes. Otherwise, remove the last item from the second box and add it to the last position in the first box. You need to **stop** summing items when one of the boxes becomes empty.

If the first loot box becomes empty print:

**"First lootbox is empty"**

If the second loot box becomes empty print:

**"Second lootbox is empty"**

In the end, you need to determine the quality of your claimed items. If the sum of the claimed items is equal to or greater than 100, print:

**"Your loot was epic! Value: {sum of claimed items}"**

Else print:

**"Your loot was poor... Value: {sum of claimed items}"**

### Input

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

### Output

* On the **first** line of output – print which box got empty in the format described above.
* On the **second** line – the quality of your loot in the format described above.

### Constraints

* All of the given numbers will be valid integers in the range **[0, 100]**.
* There won’t be a case where both loot boxes become empty at the same time.

### Examples

|  |  |  |
| --- | --- | --- |
| ****Input**** | ****Output**** | ****Comment**** |
| **10 11 8 13 5 6**  **0 4 7 3 6 23 3** | **Second lootbox is empty**  **Your loot was poor... Value: 42** | First, we sum 10 and 3. We get 13, which is not an even number, so we take the last item from the second box and move it to the last position in the first box. The current state of the boxes:  10 11 8 13 5 6 3  0 4 7 3 6 23  The next sum is 33 so we do the same again. On the third iteration, the sum is 16 which is an even number, so we remove both of the boxes and we add the value to our claimed items. We keep summing items until one of the boxes becomes empty. |
| **20 40 60 80 100**  **10 20 30 40 50 60** | **First lootbox is empty**  **Your loot was epic! Value: 500** |  |

## Bee

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/2531#1**](https://judge.softuni.org/Contests/Practice/Index/2531#1)

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

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

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

If the bee **moves** to a **flower**, it pollinates the flower and increases pollinated flowers with one.

If it goes to a **bonus**, the bee gets a bonus one moves forward and then the bonus **disappears**. If the bee **goes out** she can't return back and the program ends. If the bee receives the "**End**" command the program ends. The bee needs **at least** **5 pollinated flowers**.

### 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 the "**End**" command, you will receive a move command.

### Output

* On the first line:
  + If the bee goes out of its territory print: **"The bee got lost!"**
* On the second line:
  + If the bee couldn’t pollinate enough flowers print: **"The bee couldn't pollinate the flowers, she needed {needed} flowers more"**
  + If the bee successfully pollinated enough flowers print: **"Great job, the bee manage to pollinate {polinationed flowers} flowers!"**
* At the end print the matrix.

### Constraints

* The size of the **square** matrix will be between **[2…10].**
* There will **always** be **0** or **1** bonus, marked with - '**O**'.
* The bee position will be marked with '**B**'.
* There won't be a case where a bonus moves the bee out of its territory.

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 5  Bff..  ..O..  f.f.f  .....  fffff  right  right  down  left  left  down  down  right  down | The bee got lost!  Great job, the bee manage to pollinate 6 flowers!  .....  .....  ....f  .....  ..fff | 1) right 2) right 3) down 5) left  .Bf.. ..B.. ..... .....  ..O.. ..O.. ..... .....  f.f.f f.f.f f.B.f fB..f  ..... ..... ..... .....  fffff fffff fffff fffff  …  2) pollinate a flower: 'f' (0, 1)  3) step on a bonus: 'O' (1, 2) and make one more step down to: 'f' (2, 2)  4) the bee is on (4, 1), the next command she receives is down and she goes out of the field and the program ends. |
| 4  ....  .O..  ff..  f.B.  left  left  up  right  up  End | The bee couldn't pollinate the flowers, she needed 2 flowers more  .B..  ....  ....  .... |  |

1. **Openning**

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/2704#2**](https://judge.softuni.org/Contests/Practice/Index/2704#2)

## 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.