# Magic box



*Each member of your family has a magic box of items and now it's your turn, so you will buy a few.*

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

* **"First magic box is empty."**

If the second magic box becomes empty print:

* **"Second magic box 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 90, print:

* **"Wow, your prey was epic! Value: {sum of claimed items}"**

Else print:

* **"Poor prey... Value: {sum of claimed items}"**

### Input

* On the **first line**, you will receive the integers representing the **first magic box**, **separated** by a **single space**.
* On the **second line**, you will receive the integers representing the **second magic 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 prey in the format described above.

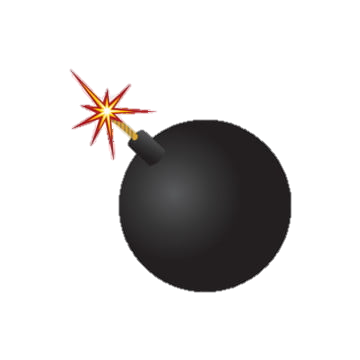
### Constraints

* All of the given numbers will be valid integers in the range **[0, 100]**.
* There won’t be a case where both magic 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 magic box is empty.**  **Poor prey... 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 magic box is empty.**  **Wow, your prey was epic! Value: 500** |  |

# Bomb



*One of the missions that new agents have to complete is called BombField. Your task is to implement the mission into a simple program.*

We get as input **the size** of the **field** in which our sapper moves. The field is **always a square**. After that, we receive the commands which represent the directions in which the sapper should move. The sapper **starts** from the **s**-position The commands will be **left/right/up/down.** If the sapper reaches the side edge of the field (left, right, up, or down), it **remains in its current position**. The possible characters that may appear on the screen are:

* **+** – regular position on the field.
* **e** – end of the route.
* **B** – bomb
* **s** – the place where the **sapper starts**

Each time when sapper finds a bomb, he deactivates it, and **replace "B" with "+"**. Keep track of the **count of the bombs**. Each time you find a bomb, you have to print the following message: **"You found a bomb!".** If the sapper **steps at the end of the route game are over (the program stops)** and you have to print the output as shown in the output section. After executing all of the commands there are only 2 possible outcomes (there are not going to be more cases):

* if you found all bombs – you win and the game ends
* if you reach the endpoint ("e"), you have to stop

Print the corresponding output depending on the case.

## Input

* **Field size** – an integer number.
* **Commands to move** the sapper – an array of strings separated by **","**.
* **The field: some of the following characters (+, e, B, s),** separated by whitespace (" ");

## Output

* There are three types of output:
  + If all of the bombs have cleared print the following output: "**Congratulations! You found all bombs!**"
  + If you reached the end, you have to stop moving and print the following line: **"END! {bombs left} bombs left on the field"**
  + If there are no more commands and none of the above cases happens, you have to print the following message: **"{bombs left} bombs left on the field. Sapper position: ({row},{col})"**

## Constraints

* The **input numbers** will be a 32-bit integer in the range [0 … 2 147 483 647].
* Allowed working time for your program: 0.1 seconds.
* Allowed memory: 16 MB.

## Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comments** |
| 5  up,right,right,up,right  + + + B +  + + + e +  + + B + +  s + + B +  + + B + + | You found a bomb!  END! 3 bombs left on the field | After executing all of the commands, the sapper moves to the endpoint. But there are some bombs left, so we print the information. |
| 4  up,right,right,right,down  + + + e  + + B +  + s + B  + + + + | You found a bomb!  You found a bomb!  Congratulations! You found all bombs! | The sapper reached the end of the field, so he remains in his position and we print the message. After all the bombs are found, we should stop the program and print the appropriate message. |

# Car Dealership



## Preparation

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

**Pay attention to the name of the package dealership, 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 repository, which stores items by creating the classes described below.

**Car**

First, write a Java class **Car** with the following public fields:

* **manufacturer: String**
* **model: String**
* **year: int**

The class **constructor** should receive the **manufacturer, model,** and **year**.You need to create the appropriate **getters and setters**. Override the **toString()** method in the following format:

**"{manufacturer} {model} ({year})"**

**Dealership**

**Next**, write a Java class **Dealership** that has **data** (**Collection**, which stores the entity **Car**). All entities inside the repository have the **same fields**. Also, the Dealership class should have those public fields:

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

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

* Field **data** – **Collection** that holds added cars
* Method add(Car car) – **adds** an **entity** to the data **if** **there** **is** an **empty cell** for the car.
* Method buy(String manufacturer, String model) – removes the car by **given manufacturer and model,** if such **exists**, and **returns boolean**.
* Method getLatestCar() – returns the **latest** car (by year) or null if have no cars.
* Method **getCar(String manufacturer, String model)** – returns the car with the **given manufacturer** and **model** or **null** if there is no such car.
* Getter getCount() – **returns** the **number** of cars.
* **getStatistics()** – **returns** a **String** in the following **format**:
  + **"** **The cars are in a car dealership {name}:  
    {Car1}  
    {Car2}  
    (…)**"

## Constraints

* The **combinations** of **manufacturers** and **models** will be **always unique**.
* The **year** of the cars will always be **positive**.
* There won't be cars of the same year.

## Examples

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

|  |
| --- |
| Sample code usage |
| // Initialize the repository  Dealership dealership = new Dealership("Autofest", 5);  // Initialize entity  Car volvo = new Car("Volvo", "XC70", 2010);  // Print Car  System.out.println(volvo); // Volvo XC70 (2010)  // Add Car  dealership.add(volvo);  // Remove Car  System.out.println(dealership.buy("Volvo", "XC90")); // false  System.out.println(dealership.buy("Volvo", "XC70")); // true  Car peugeot = new Car("Peugeot", "307", 2011);  Car audi = new Car("Audi", "S4", 2005);  dealership.add(peugeot);  dealership.add(audi);  // Get Latest Car  Car latestCar = dealership.getLatestCar();  System.out.println(latestCar); // Peugeot 307 (2011)  // Get Car  Car audiS4 = dealership.getCar("Audi", "S4");  System.out.println(audiS4); // Audi S4 (2005)  // Count  System.out.println(dealership.getCount()); // 2  // Get Statistics  System.out.println(dealership.getStatistics());  // The cars are in a car dealership Autofest:  // Peugeot 307 (2011)  // Audi S4 (2005) |

## Submission

Submit a **single .zip file**, containing the **dealership package, with the classes inside (Car, Dealership, 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.