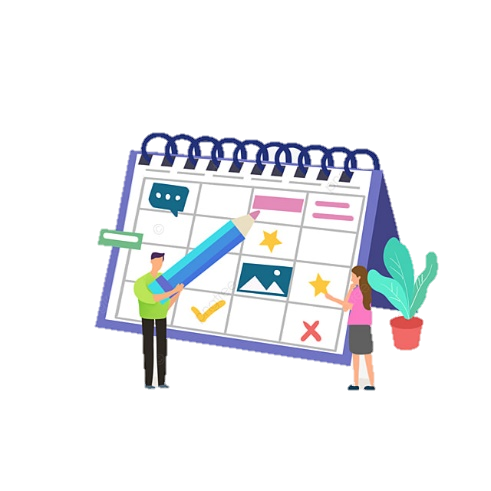
# OS Planning



*You are hired to create a program that schedules the work of an OS and avoids tasks that could harm it.*

On the **first line,** you will be given some **tasks** as **integer values**, separated by a **comma and space** **", "**. On the **second line,** you will be given some **threads as integer values**, separated by a single space**.** On the **third line**, you will receive the **integer value** of a **task** that you need to **kill**. Your job is to **stop the work of the OS as soon as you get to this task**, otherwise, your OS will crash. The **thread** that gets **first** to this task, **kills** **it**.

The **OS** **works** in the following way:

* It takes the **first given thread value** and the **last given task** **value.**
* If the **thread value** is **greater** than or **equal** to the **task** value, **the task and thread get removed**.
* If the **thread** **value** is **less** than the **task value**, the **thread** gets **removed**, but the **task** **remains**.

After you finish the needed task, print on a single line:

**"Thread with value {thread} killed task {taskToBeKilled}"**

Then print the remaining threads (**including** the **one that killed** the **task**) starting from the first on a single line, separated by a single space.

### Input

* **On the first line,** you will receive the **tasks**, separated by **", "**.
* **On the second line,** you will the **threads**, separated by a single space.
* **On the third line**, you will receive a **single integer** – a **value of the task** to be killed.

### Output

* Print the **thread that killed the task** and the **task itself** in the format given above.
* Print the **remaining threads** starting **from the first** on a single line, separated by a single space.

### Constraints

* The needed task will always be with a **unique** value.
* You will **always have enough threads to get to the needed task.**

### Examples

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 20, 23, 54, 34, 90  150 64 20 34  54 | Thread with value 20 killed task 54  20 34 | First, the thread with a value of 150 is taken and the task with a value of 90. The thread has a bigger value, so both thread and task get removed. Next, thread 64 finishes task 34 and both get removed. Then thread 20 gets to task 54 and kills it. |
| 33, 12, 15, 40, 45, 60  30 20 53 67 84 90  40 | Thread with value 90 killed task 40  90 | Thread 30 takes task 60, but the task has greater value, so the thread gets removed. Then thread 20 takes task 60 and the same happens – the thread gets removed. Then the same happens with thread 53. After that, thread 67 takes task 60 and finishes it. Then thread 84 finishes task 45. Finally, thread 90 gets to task 40, which should be killed and the program stops. |

# Python



*One of the candidates-intern must solve the task - "Python". He must write a Java code that calculates wheater the python wins or loses the game.*

Here are the rules of the game:

The game starts with a **python length of 1**.

We get as input **the size** of the **screen** in which our python moves. The screen is **always square**. After that, we receive the commands which represent the directions in which the python should move.The python **starts** from **s**-position The commands will be: "**left**"**,** "**right**", "**up**", "**down**"**.** If the python reaches the side edge of the screen (left or right), it goes to the **opposite side of the same row**. If the python reaches the top/bottom edge of the screen it goes on the **opposite side of the same column**. The possible characters that may appear on the screen are:

* **\*** – that is a regular asterisk; it does nothing
* **e** – represents an enemy.
* **f** – this is the food
* **s** – the place where the **game starts**

Each time you eat a piece of food your **length increases by one**. Keep track of the length, because **in case you win you have to print it**. If you **step on an enemy the game is over (the python stops moving)** and you have to print the output as shown in the output section. After executing all of the commands there are 3 possible outcomes:

* you have eaten all the food and you win
* you get killed by an enemy
* there is still some food to be eaten

Print the corresponding output depending on the case.

## Input

* **Length** of the screen side – an integer number.
* **Commands to move** the python – an array of strings separated by "**,**".

## Output

* There are three types of output:
  + If all of the food is eaten print the following output: **"You win! Final python length is {length}"**
  + If there are no left commands and there is still some food to be eaten: **"You lose! There is still {left food} food to be eaten."**
  + If you step on the enemy the game is over and you print **"You lose! Killed by an enemy!"**

## 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, right, up  \* \* e \* \*  \* \* \* f \*  \* f \* \* \*  s \* \* \* \*  \* \* e \* \* | You win! Final python length is 3 | After executing all of the commands the python has eaten all of the food and it is still alive. |
| 4  right, right, right, right, right, down, right  \* s \* \*  \* \* e \*  \* f \* f  \* \* \* f | You lose! Killed by an enemy! | The python moves 2 times to the right, then it goes off-screen and appears on the left side. Then it makes 2 more moves to the right, it goes down, steps on an enemy and the game is over. |
| 6  down, left, left, down, right, right, right, right, right  \* \* \* \* s \*  \* e f \* \* \*  f \* \* \* \* \*  \* \* \* f e \*  \* e \* \* \* \*  \* \* \* \* \* \* | You lose! There is still 1 food to be eaten. | The python survived but there is still 1 piece of food that the python could not eat. |

# Grooming Salon



## Preparation

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

**Pay attention to the name of the package groomingSalon, 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.

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

* **name: String**
* **age: int**
* **owner: String**

The class **constructor** should receive **all fields.** You need to create the appropriate **getters and setters**. The class should override the **toString()** method in the following format:

**"{name} {age} - ({owner})"**

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

* **capacity: int**

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

* Field **data** – **List** that holds added pets
* Method add(Pet pet) – **adds** an **entity** to the data **if** **there** **is** an **empty place** inthegrooming salon for the pet.
* Method remove(String name) – removes the pet by **given name,** if such **exists**, and **returns boolean**.
* Method **getPet(String name, String owner)** – returns the pet with the **given name** and **owner** or **null if no such pet exists**.
* Getter getCount – **returns** the **number** of pets.
* **getStatistics()** – **returns** a **String** in the following **format**:
  + **"** **The grooming salon has the following clients:  
    {name} {owner}  
    {name} {owner}**

**(…)**"

## Constraints

* The **combinations** of **names** and **owners** will **always be unique**.
* The **age** of the pets will always be **positive**.

## Examples

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

|  |
| --- |
| Sample code usage |
| // Initialize the repository  GroomingSalon salon = new GroomingSalon(20);  // Initialize entity  Pet dog = new Pet("Ellias", 5, "Tim");  // Print Pet  System.out.println(dog); // Ellias 5 - (Tim)  // Add Pet  salon.add(dog);  // Remove Pet  System.out.println(salon.remove("Ellias")); // true  System.out.println(salon.remove("Pufa")); // false  Pet cat = new Pet("Bella", 2, "Mia");  Pet bunny = new Pet("Zak", 4, "Jon");  salon.add(cat);  salon.add(bunny);  // Get Pet  Pet pet = salon.getPet("Bella", "Mia");  System.out.println(pet); // Bella 2 - (Mia)  // Count  System.out.println(salon.getCount()); // 2  // Get Statistics  System.out.println(salon.getStatistics());  // The grooming salon has the following clients:  //Bella Mia  //Zak Jon |

## Submission

Zip all the files in the project folder except the **bin** and **obj** folders.

Submit a **single .zip file**, containing **groomingSalon package, with the classes inside (Pet, GroomingSalon, 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.