# Exam Preparation – 13 June 2024

## **Offroad Challenge**

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/4355#0**](https://judge.softuni.org/Contests/Practice/Index/4355#0)

*John is quite an avid off-road fan. He bought a new jeep and made the necessary improvements to it. John is ready for new off-road adventures and can't wait to get started. In this challenge, he must save his fuel very carefully…*

There will be **two sequences of integers**. The first sequence will represent the **initial fuel** and the second - **additional consumption index** due to thin air at high altitudes, hence higher fuel consumption. There will also be a third **sequence of integers**, representing values equal to the necessary **amount of fuel needed** to reach the corresponding altitude in the challenge.

Your task is to take the last **fuel quantity** from the **fuel sequence** and the firstindexfrom **the additional consumption index sequence**. **Subtract the values** and **check the result**.

* The corresponding altitude is reached if the calculated result is bigger or equal to the first element from the **needed amount of fuel** sequence. You need to **remove both the fuel and the consumption** indexfrom their sequences as well as the **needed amount of fuel** index from their sequence.
* If the calculated result **is smaller or not equal** to the first element from **the needed amount of fuel** sequence, the corresponding altitude is not reached, movement cannot continue, and the program should end.

### **Input**

* The first line will represent the initial **fuel** – **integers**, separated by a **single space.**
* The second line will represent the **consumption indexes** that decrease initial **fuel** – **integers**, separated by a **single space**.
* The third line will represent the **quantities** needed to reach the corresponding altitude – **integers**, separated by a **single space**.

### **Output**

* On the **first** or the next **n** lines, output the corresponding message on the console from the following options:
* If John **reaches the altitude**, print the message:
* **"John has reached: Altitude 1"**
* **…**
* **"John has reached: Altitude {n}"**
* If John **fails to reach the altitude**, print the message:
* **"John did not reach: Altitude {n}"**
* On the **next** lines, based on whether he reached the top or not, print the following on the console in the specified format:
* If John **doesn't have enough fuel to reach the top** but **has reached some altitude**, display the following messages:
* **"John failed to reach the top.**

**Reached altitudes: Altitude 1, … Altitude {n}"**

* If John **does not have enough fuel to reach the top** and **has not reached any altitude**, print:
  + **"John failed to reach the top.**

**John didn't reach any altitude."**

* If John manages to **reach all the altitudes**, he will reach the top. End the program and **print on the console** the following message:
  + **"John has reached all the altitudes and managed to reach the top!"**

### **Constraints**

* All the given numbers will be valid **integers** in the range **[1, 200].**
* All sequences always consist of **four** elements.
* There will be **no negative input**.

### **Examples**

|  |  |
| --- | --- |
| **Input** | **Output** |
| **200 90 40** 100  20 **40 30 50**  **50 60 80 90** | John has reached: Altitude 1  John did not reach: Altitude 2  John failed to reach the top.  Reached altitudes: Altitude 1 |
| **Comment** | |
| We start by taking the last fuel quantity from the fuel sequence (**100**) and the first **additional consumption index from the consumption index fuel sequence** (**20**). The result from subtraction is **100 - 20 = 80**. After that, we check if the sum equals or exceeds the **first amount of needed fuel**. The **result (80)** is more than the **needed fuel (50)** for this altitude, so the **altitude is reached.** As the altitude is reached, we **remove an element from every sequence**. We continue with the next altitude to do the same and as a result, we have 40 – 40 = 0. The needed fuel is **60**, we do not have enough fuel to reach the current altitude, so the challenge for John ends here. | |
| **Input** | **Output** |
| 40 66 123 100  10 30 70 33  40 55 77 100 | John has reached: Altitude 1  John has reached: Altitude 2  John did not reach: Altitude 3  John failed to reach the top.  Reached altitudes: Altitude 1, Altitude 2 |
| **Comment** | |
| Here we take the last **fuel** quantity and like in the previous case subtract the **consumption index** from the fuel and continue forward until the result is equal to or greater than the required fuel otherwise the program stops. | |
| **Input** | **Output** |
| 199 190 100 100  20 40 30 50  50 60 70 80 | John has reached: Altitude 1  John has reached: Altitude 2  John has reached: Altitude 3  John has reached: Altitude 4  John has reached all the altitudes and managed to reach the top! |
| **Comment** | |
| Here all altitudes are conquered, and John successfully reaches the top. | |

## **The Gambler**

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/4438#1**](https://judge.softuni.org/Contests/Practice/Index/4438#1)

You will be given an integer **n** for the size of the **game board** (square shape). On the next **n** lines, you will receive the rows of the **board**. The gambler will start at a **random** position, marked with the letter '**G**' and have an **initial** **'entering the game'** amount of **100$**.

On each turn, **until command 'end'** is received, you will receive commands for the **direction**, in which the gambler should move. The commands will be "**up**", "**down**", "**left**" and "**right**".

* If a position with **'-' (dash)** is reached, it means that the field is empty and the gambler awaits its next direction.
* If the **position marked** with the letter **'W'** is reached the gambler takes it and adds **100$** to his amount
* If the **position marked** with the letter **'P'** (penalty) is reached **decrease** the gambler's total amount by **200$**
* If the **position marked** with the letter **'J'** is reached the gambler **wins** the jackpot and adds **100000$** to his amount and the game ends.
* If **the gambler leaves** the **boundaries of the board or** his **total amount** becomes **equal to or drops** below **0** (zero)**, he loses everything** and you should stop the program.

The current gambler position should be marked with **'G'**

When the gambler leaves a position marked with a letter it should be replaced by **'-' (dash)**

The program **ends** when one of **these four events** occurs:

* the gambler **leaves** the board boundaries
* command **'end'** is received
* the gambler's total winning amount is **equal to or drops below 0(zero)**
* the position **marked** with **'J'** is reached

### **Input**

* On the first line, you are given the integer **n** – the size of the matrix (board).
* The **next n lines** hold the values for every **row**.
* On each of the next lines, you will get a direction command.

### **Output**

* If you win the jackpot on the first and second lines print:
* **"You win the Jackpot!**

**End of the game. Total amount: {amount}$"**

* If you do not win the jackpot print:
* **"End of the game. Total amount: {amount}$"**
* If you leave the boundaries of the matrix or the gambler's amount becomes 0(zero) or below print:
* **"Game over! You lost everything!"**

### **Constraints**

* The **square matrix** (board) size will be between **[4…10].**
* Gambler's starting position will always be marked with '**G**'.
* There will always be a field marked with **'W'** and it may appear more than once.
* There will be always one field marked with '**J**'.
* There will always be one or two fields marked with **'P'**.
* You will always receive enough commands to end the game.
* Finally if **you** **have any amount print** the **matrix.**

### **Examples**

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 4  W-GW  W--W  --P-  ----  down  down  end | **Game over! You lost everything!** | The movement starts from position **[0,2]** after receiving the command "**down**" the gambler moves to position **[1,2]** where there is a **'-'** (dash) field - nothing is happening. The next command is "**down**" again, the gambler lands on a **'P'** (penalty) field and since he has to pay **200$** his sum becomes negative (100 – 200 = -100) and therefore loses it. The game ends. |
| 4  G---  WWWW  P---  PJ--  right  right  right  down  left  left  end | **End of the game. Total amount: 400$**  **----**  **WG--**  **P---**  **PJ--** |  |
| 4  ---G  W-W-  ---P  --JW  left  down  down  down  right  end | **You win the Jackpot!**  **End of the game. Total amount: 100200$**  **----**  **W---**  **---P**  **--GW** |  |

# 03. Chasing Sharks

**Link:** [**https://judge.softuni.org/Contests/Practice/Index/4543#2**](https://judge.softuni.org/Contests/Practice/Index/4543#2)

*Sharks have been around for at least 420 million years. They are the top predators in the oceans. Let's find out why.*

## **Preparation**

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

**Pay attention to name the package sharkHaunt, 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 that stores sharks by creating the classes described below.

## **Shark**

First, write a class **Shark** with the following properties:

* **kind: String**
* **length: int**
* **food: String**
* **habitation: String**

The class **constructor** should receive **kind, length, food and habitation**. You need to create the appropriate **getters and setters**. All shark kinds will be **unique.** It is guaranteed that there **will be no duplicates** of kinds.

Override the **toString()** method in the following format:  
**The {kind} is {length} centimeters long, eats {food} and inhabits {habitation}.**

## **Basin**

**Next**, write a class **Basin** that has **data** (a List which stores the entity **Shark**). All entities inside the repository have the **same properties**. The **Basin** class should have those **properties**:

* **name: String**
* **capacity: int**
* **sharks: List<Shark>**

The class **constructor** should receive **name** and **capacity.** Also, it should initialise the **sharks** with a new **collection** instance.Implement the following features:

* **Method addShark(Shark shark)** – **adds** an **entity** to the data **if** a spacefor it, otherwise print: **"This basin is at full capacity!"**
* **Method removeShark(String kind)** – removes a shark by **given kind,** if such **exists**, and **returns boolean** (true if it is removed, otherwise – false)
* **Method getLargestShark()**– **returns** the **largest shark** by **length** in the given basin
* **Method getShark(String kind)** – **returns** the **shark** with the **given kind,** otherwise – returns **null**
* **Method getCount** – **returns** the **count** of **sharks** in the given basin
* **Method getAverageLength –** **returns** integer - the average length of **the sharks** in the given basin
* **Method report()** – **returns** a **string** in the following **format** (print the sharks in **order of addition**):
  + **"Sharks in {basin name}:  
    The {kind} is {length} centimeters long, eats {food} and inhabits {habitation}.  
    The {kind} is {length} centimeters long, eats {food} and inhabits {habitation}.  
     (…)"**

## **Constraints**

* The **kind** and **length** of the shark will always be **unique**.
* You will always have a shark added before receiving methods manipulating the Basins's shark.

## **Examples**

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

|  |
| --- |
| **Sample code usage** |
| //Initialize the repositories (Basin)  Basin pacific = **new** Basin("Pacific Ocean", 6);  Basin atlantic = **new** Basin("Atlantic Ocean", 2);  Basin ganges = **new** Basin("Ganges River", 1);  //Initialize entities (Shark)  Shark tigerShark = **new** Shark("Tiger shark", 300, "mammals", "saltwater");  Shark whaleShark = **new** Shark("Whale shark", 1200, "zooplankton", "saltwater");  Shark dwarfShark = **new** Shark("Dwarf lantern shark", 20, "shrimp", "saltwater");  Shark bullShark = **new** Shark("Bull shark", 330, "dolphins", "freshtwater");  Shark gangesShark = **new** Shark("Ganges shark", 178, "fish", "freshwater");  //Add Shark  pacific.addShark(tigerShark);  pacific.addShark(whaleShark);  pacific.addShark(dwarfShark);  atlantic.addShark(bullShark);  ganges.addShark(gangesShark);  //Remove Shark  System.***out***.println(pacific.removeShark("Carpet shark")); //false  System.***out***.println(atlantic.removeShark("Lemon shark")); //false  System.***out***.println(atlantic.removeShark("Bull shark")); //true  //Get the largest shark  System.***out***.println(pacific.getLargestShark().getKind()); //Whale shark  //Get the average length of sharks in given basin  System.***out***.println(pacific.getAverageLength()); //506  System.***out***.println(atlantic.getCount()); //0  //Get a report for the given basin  System.***out***.println(pacific.report());  System.***out***.println(ganges.report());  //Sharks in Pacific Ocean:  //The Tiger shark is 300 centimeters long, eats mammals and inhabits saltwater.  //The Whale shark is 1200 centimeters long, eats zooplankton and inhabits saltwater.  //The Dwarf lantern shark is 20 centimeters long, eats shrimp and inhabits saltwater.  //Sharks in Ganges River:  //The Ganges shark is 178 centimeters long, eats fish and inhabits freshwater. |

## **Submission**

Submit **single .zip file**, containing **sharkHaunt** package, **with the classes inside** (**Basin** and **Shark** 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 **main(String[] args)** method inside.