Stack & Queue - Exercises

[1. Reverse Numbers Stack 1](#_Toc145951749)

[2. Basic Stack Operations 1](#_Toc145951750)

[3. Browser 2](#_Toc145951751)

[4. Simple Calculator 3](#_Toc145951752)

[5. Decimal to Binary Converter 3](#_Toc145951753)

[6. Matching Brackets 4](#_Toc145951754)

[7. Printer Queue 4](#_Toc145951755)

[8. Balanced Parentheses 5](#_Toc145951756)

[9. Hot Potato 5](#_Toc145951757)

[10. Maximum Element 6](#_Toc145951758)

[11. Basic Queue Operations 7](#_Toc145951759)

[12. Task Scheduler 8](#_Toc145951760)

[13. Document editor 8](#_Toc145951761)

[14. Recursive Fibonacci 9](#_Toc145951762)

[15. Factorio 9](#_Toc145951763)

# Reverse Numbers Stack

Write a program that reads **N integers** from the console and **reverses them using a stack**. Use the **ArrayDeque<Integer>** class. Just put the input numbers in the stack and pop them.

|  |  |
| --- | --- |
| Input | Output |
| 1 2 3 4 5 | 5 4 3 2 1 |
| 1 | 1 |

# Basic Stack Operations

You will be given an integer **N** representing the **number of elements to push into the stack**, an integer **S** representing the **number of elements to pop from the stack,** and an integer **X**, an element **that you should check whether is present in the stack**. If it is present, print "**true**"on the console. If it's not, print the smallest element currently present in the stack.

* On the first line, you will be given **N**, **S,** and **X** separated by a single space.
* On the next line, you will be given a line of numbers **separated by one or more white spaces**.
* On a single line print, either "**true**"if **X** is present in the stack, otherwise, **print the smallest** element in the stack.
* If the stack is empty – print 0.

|  |  |
| --- | --- |
| Input | Output |
| 5 2 13  1 13 45 32 4 | true |
| 4 1 700  420 69 13 700 | 13 |
| 3 3 90  90 90 90 | 0 |

# Browser

**Write a program that takes two types of browser instructions:**

* **Normal navigation: a URL is set, given by a string;**
* **The string "back" sets the current URL to the last set URL**

**After each instruction, the program should print the current URL. If the back instruction can't be executed, print   
"no previous URLs".**

**The input ends with the "Home" command, and then you simply have to stop the program.**

* **Use ArrayDeque<>.**
* **Use String to keep the current page.**
* **Use push(), when moving to the next page.**
* **Use pop(), when going back.**

|  |  |
| --- | --- |
| Input | Output |
| <https://www.google.com/>  back  <https://www.google.com/search?q=developer+path>  <https://roadmap.sh/>  <https://roadmap.sh/backend>  back  Home | **https://www.google.com/**  **no previous URLs**  **https://www.google.com/search?q=developer+path**  **https://roadmap.sh/**  **https://roadmap.sh/backend**  **https://roadmap.sh/** |
| <https://www.google.com/>  <https://www.google.com/search?q=sirma>  back  back  <https://www.google.com/search?q=java>  back  Home | [**https://www.google.com/**](https://www.google.com/)  [**https://www.google.com/search?q=sirma**](https://www.google.com/search?q=developer+path)  [**https://www.google.com/**](https://www.google.com/)  **no previous URLs**  [**https://www.google.com/search?q=java**](https://www.google.com/search?q=java)  [**https://www.google.com/**](https://www.google.com/) |

# Simple Calculator

**Create a simple calculator that can evaluate simple expressions that will not hold any operator different from addition and subtraction. There will not be parentheses or operator precedence.**

**Solve the problem using a Stack.**

|  |  |
| --- | --- |
| Input | Output |
| 2 + 5 + 10 - 2 - 1 | **14** |
| 2 - 2 + 5 | **5** |

# Decimal to Binary Converter

**Create a simple program that can convert a decimal number to its binary representation. Implement an elegant solution using a Stack.**

**Print the binary representation back at the terminal.**

* **If the given number is 0, just print 0.**
* **Else, while the number is greater than zero, divide it by 2 and push the reminder into the stack.**

while(decimal != 0){

stack.push(decimal % 2);

decimal /= 2;

}

* **When you are done dividing, pop all reminders from the stack, which is the binary representation.**

|  |  |
| --- | --- |
| Input | Output |
| 10 | **1010** |
| 1024 | **10000000000** |

# Matching Brackets

**We are given an arithmetical expression with brackets. Scan through the string and extract each sub-expression.**

**Print the result back at the terminal.**

* **Scan through the expression searching for brackets:**
  + **If you find an opening bracket, push the index into the stack.**
  + **If you find a closing bracket, pop the topmost element from the stack. This is the index of the opening bracket.**
  + **Use the current and the popped index to extract the sub-expression.**

|  |  |
| --- | --- |
| Input | Output |
| 1 + (2 - (2 + 3) \* 4 / (3 + 1)) \* 5 | **(2 + 3)**  **(3 + 1)**  **(2 - (2 + 3) \* 4 / (3 + 1))** |
| (2 + 3) - (2 + 3) | **(2 + 3)**  **(2 + 3)** |

# Printer Queue

The printer queue is a simple way to control the order of files sent to a printer device. We know that a single printer can be shared between multiple devices. Тo preserve the order of the files sent, we can use a queue.

Write a program which takes filenames until "print" command is received. Then as output, print the filenames in the order of printing. Some of the tasks may be canceled. If you receive "cancel" you have to remove the first file to be printed. If there is no file in the queue, print "Standby".

* Use offer(), when adding the file.
* Use pollFirst(), when printing the file.

|  |  |
| --- | --- |
| Input | Output |
| Exercises.docx  cancel  cancel  Presentation.pptx  Note.txt  MyClass.java  cancel  print | **Canceled Exercises.docx**  **Stanby**  **Canceled Presentation.pptx**  **Note.txt**  **MyClass.java** |
| Presentation.pptx  cancel  Text.txt  cancel  cancel  cancel  print | **Canceled Presentation.pptx**  **Canceled Text.txt**  **Standby**  **Standby** |

# Balanced Parentheses

Create a program that checks if a given string containing different types of brackets (e.g. '{}', '[]', '()') is balanced or not using Stack. The string is balanced if the brackets open and close in the correct order.

|  |  |
| --- | --- |
| Input | Output |
| {[()]} | **True** |
| {[(])} | **False** |
| [{()()}] | **True** |
| (((([])))) | **True** |
| [{()()}{()()(}] | **False** |
| [{()()}{()()()}] | **True** |

# Hot Potato

Hot potato is a game in which children form a circle and start passing a hot potato. The counting starts with the first kid. Every nth toss, the child left with the potato leaves the game. When a kid leaves the game, it passes the potato forward. This continues repeating until there is only one kid left.

Create a program that simulates the game Hot Potato. Print every kid that is removed from the circle. In the end, print the kid that is left last.

|  |  |
| --- | --- |
| Input | Output |
| Smith Ivan Marry  2 | **Remove Ivan**  **Remove Smith**  **Last is Marry** |
| George Peter Ivan John Harry  10 | **Remove Harry**  **Remove Peter**  **Remove Ivan**  **Remove George**  **Last is John** |
| George Peter Misha Sara Carl  1 | **Remove George**  **Remove Peter**  **Remove Misha**  **Remove Sara**  **Last is Carl** |

# Maximum Element

You have an empty sequence, and you will be given **N** commands. Each command is one of the following types:

* "**1 X**" - **Push** the element **X** into the stack.
* "**2**" - **Delete** the element present at the top of the stack.
* "**3**" - **Print** the maximum element in the stack.
* The first line of input contains an integer **N**, where **1 ≤ N ≤ 105**.
* The next **N** lines contain commands.
* The element **X** will be in the range **1 ≤ X ≤ 109**.
* The **type of the command** will be in the range **1 ≤ Type ≤ 3.**
* For each command of **type** "**3**", **print the maximum element** in the stack on a new line.

|  |  |
| --- | --- |
| Input | Output |
| 9  1 97  2  1 20  2  1 26  1 20  3  1 91  3 | 26  91 |
| 7  1 81  2  1 14  2  1 14  1 47  3 | 47 |

# Basic Queue Operations

You will be given an integer N representing the number of elements to enqueue (add), an integer S representing the number of elements to dequeue (remove/poll) from the queue, and finally, an integer X, an element that you should check whether is present in the queue. If it is - prints true on the console, if it is not - print the smallest element currently present in the queue.

|  |  |
| --- | --- |
| Input | Output |
| 5 2 32  1 13 45 32 4 | true |
| 4 1 700  700 69 13 420 | 13 |
| 3 3 90  90 90 90 | 0 |

# Task Scheduler

Create a PriorityQueue to manage tasks. Each task has a name and a priority. Implement the getNextTask function to get the next task with the highest priority.

|  |  |
| --- | --- |
| Input | Output |
| Add Clean 1  Add Work 2  getNextTask  Add Exercise 3  Add Study 4  getNextTask | Work - 2  Study - 4 |
| Add Work 1  Add Study 1  getNextTask | Work - 1 |
| Add Play 1  Add Sprint 0  Add Play 10  Add Relax 8  getNextTask | Play - 10 |

# Document editor

Use two stacks to facilitate Undo and Redo.

|  |  |
| --- | --- |
| Input | Output |
| Insert("Hello")  Insert(" World")  Undo()  Redo()  End | Hello  Hello World  Hello  Hello World |
| Insert("One")  Insert(" Two")  Insert(" Three")  Undo()  Insert(" Undone")  Undo()  Redo()  End | One  One Two  One Two Three  One Two  One Two Undone  One Two  One Two Undone |

# Recursive Fibonacci

Each member of the **Fibonacci sequence** is calculated from the **sum of the two previous members**. The first two elements are 0, 1. Therefore, the sequence goes like 0, 1, 1, 2, 3, 5, 8, 13, 21, 34…

The following sequence can be generated with an array, but that's easy, so **your task is to implement it recursively**.

If the function **getFibonacci(n)** returns the nth Fibonacci number, we can express it using **getFibonacci(n) = getFibonacci(n-1) + getFibonacci(n-2)**.

However, this will never end, and a Stack Overflow Exception is thrown in a few seconds. For the recursion to stop, it has to have a "bottom". The bottom of the recursion is getFibonacci(1), and should return 1. The same goes for getFibonacci(0).

* On the only line in the input, the user should enter the wanted Fibonacci number **N** where **1 ≤ N ≤ 49.**
* The output should be the nth Fibonacci number counting from 0.

|  |  |
| --- | --- |
| Input | Output |
| 5 | 5 |
| 8 | 35 |
| 2 | 1 |

For the nth Fibonacci number, we calculate the N-1st and the N-2nd number, but for the calculation of N-1st number, we calculate the N-1-1st(N-2nd) and the N-1-2nd number, so we have a lot of repeated calculations.

If you want to figure out how to skip those unnecessary calculations, you can search for a technique called [memoization](https://en.wikipedia.org/wiki/Memoization).

# Factorio

You are creating a robotized assembly line.

Each robot has a **processing time**, the time it needs to process a product. When a **robot is free,** it should **take a product for processing** and log its name, product, and processing start time.

Each robot **processes a product coming from the assembly line**. A **product comes** from the line **each second** (so the first product should appear at [start time + 1 second]). If a product passes the line and **there is no free robot** to take it, it should be **queued at the end of the line again**.

The robots are **standing in line in order of their appearance**.

* On the first line, you will get the names of the robots and their processing times in the format:

"**robot-processTime|robot-processTime|robot-processTime**"**.**

* On the second line, you will get the starting time in the format "**hh:mm:ss**"**.**
* Next, until the "**End**" command, you will get a product on each line.

|  |  |
| --- | --- |
| Input | Output |
| R2-15|D2-10|WX78-3  8:00:00  detail  glass  wood  laptop  End | R2 - detail [08:00:01]  D2 - glass [08:00:02]  WX78 - wood [08:00:03]  WX78 - laptop [08:00:06] |
| R2-60  8:00:00  detail  glass  wood  sock  End | R2 - detail [08:00:01]  R2 - sock [08:01:01]  R2 - wood [08:02:01]  R2 - glass [08:03:01] |