# More Exercises: Array and List Algorithms

Problems for exercises and homework for the [“Programming Fundamentals Extended” course @ SoftUni](https://softuni.bg/courses/programming-fundamentals).

You can check your solutions here: <https://judge.softuni.bg/Contests/428>

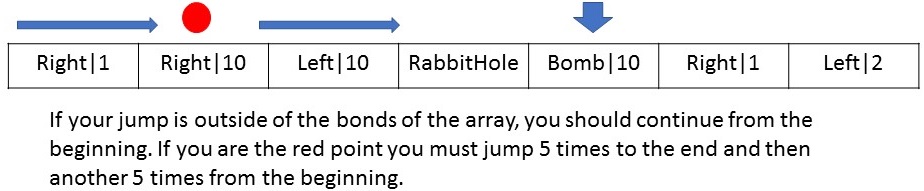
## Rabbit Hole

Rabbit hole is a term for a portal for time travel.

You are trying to go back in the past to prevent Kennedy’s assassination. But first you should find the rabbit hole. Your energy is limited and if you exhaust it, you will be tired and can’t continue the mission.

You will be given an array of strings representing different obstacles you should overcome.

* “**Left|[integer value]**” - you should jump to the left with **[integer value]** positions and decrease your energy with **[integer value];**
* “**Right|[integer value]**” - you should jump to the right with **[integer value]** positions and decrease your energy with **[integer value]**;
* “**Bomb|[integer value]**” - the bomb explodes and this element of the array should be removed, your energy should be decreased be the **[integer value]**, you should start from the beginning (**index 0**);
* “**RabbitHole**” – you have found the rabbit hole, the program should stop here, print on the console – “You have 5 years to save Kennedy!”



**Your mission begins at the first element of the array. The rabbit hole will be only one.**

The program ends when you find the rabbit hole or when your energy is gone. And if it is gone due to bomb explosion you should print “You are dead due to bomb explosion!” on the console or if it is due jumps print

“You are tired. You can't continue the mission.”.

After every move someone puts a bomb at the end of the array with **[integer value] of your current energy** (the last element is removed and the bomb is placed there, but when the last element is the “RabbitHole”, it can’t be removed and the bomb is placed after it).

### Input / Constraints

* First line – space separated string values consists the obstacles you should overcome.
* Second line – integer value representing your initial energy in range [0-100].

### Output

* If you find the Rabbit Hole you should print: “You have 5 years to save Kennedy!”;
* If you die due to bomb explosion (your energy is gone, after a bomb explosion): “You are dead due to bomb explosion!”;
* If you are tired (your energy is gone, after a jump): “You are tired. You can't continue the mission.” .

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| RabbitHole Left|100 Right|10 Bomb|10 Left|11  100 | You have 5 years to save Kennedy! |

|  |  |
| --- | --- |
| **Input** | **Output** |
| Right|100 Left|10 Bomb|11 RabbitHole Bomb|10 Left|2  80 | You are tired. You can't continue the mission. |

## Binary Search

In most of our tasks we use **linear search**, when we are looking for an element. However, another simple to implement and widely used algorithm is the **binary search**. Its idea is to work on **sorted** collections and to use the principle of **divide and conquer**.

Your task is to write two methods, which receive an array and an element, which we will be looking for. Perform the linear search on the **unmodified** collection and start from the **first** element. For the binary search **order** the collection in **ascending** **order**, so you receive a viable result. Print “**Yes**” if the element **can** be found in the collection or “**No**” if it’s **not** present.

After that print the **count** of **iterations**, which the **linear** search made in format:

“Linear search made {countOfIterations} iterations”

And the **count** of **iterations**, which **binary** search made:

“Binary search made {countOfIterations} iterations”

### Input / Constraints

* **First** line – space separated **unique** integers.
* **Second** line – the **element**, which we are searching.
* The numbers will be in the interval **[-2147483648…2147483647]**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 7 3 3 5 1 2  5 | Yes  Linear search made 4 iterations  Binary search made 2 iterations |
| 1 2 3 4 5 6 7 8  2 | Yes  Linear search made 2 iterations  Binary search made 2 iterations |
| 20 10 30 50 90 60  23 | No  Linear search made 6 iterations  Binary search made 3 iterations |

## Hints

* Read more about **linear** search [here](https://www.tutorialspoint.com/data_structures_algorithms/linear_search_algorithm.htm)
* Read more about **binary** search [here](https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm)

## Mirror Image

You will receive a collection of elements from the console. Until you receive the command “**Print**” you will receive on every line a single number - n, which will represent index from your collection. Your task is to reverse the elements from index **0** to index n-1 and from n+1 to the **last** element of the collection. You should not change he position of the element at the given index.

Example: We have the array **10 20 30 40 50 60 70 80 90**. We receive from the console the index **3**, which is the number **40**. After reversing the element left and right from the number the collection will look like this:

**30 20 10 40 90 80 70 60 50**

### Input / Constraints

* **First** line – space separated integers.
* **Next** lines – single **number**, which will represent the index, **left** and **right** from which we need to **reverse** our **array**
* The numbers will be in the interval **[-2147483648…2147483647]**
* The given **indices** will always be **inside** the array

### Output

* Print the **elements** of the array **separated** with **single** white space

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| **10 20 30 40 50 60 70 80 90**  3  5  8  Print | 60 50 80 30 20 10 40 90 70 |

|  |  |
| --- | --- |
| **Input** | **Output** |
| **Every day I am shuffling**  0  4  Print | I am shuffling Every day |

|  |  |
| --- | --- |
| **Input** | **Output** |
| **1 6 10 4 10 2**  3  Print | 10 6 1 4 2 10 |

## Japanese Roulette

Russian roulette is a game of chance where a single player places a single bullet in a revolver, spins the cylinder, points the muzzle against their head and pulls the trigger. Japanese people however are not that brave (crazy) so they **play** with **toy guns**.

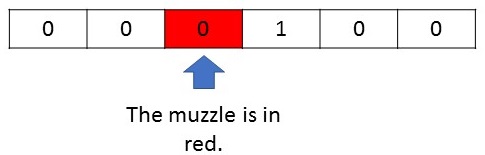
Every player can spin the cylinder with different strength. The **numeric** representation of the **strength** is with **how many places they can move the bullet while spinning**. Note that the cylinder has only **6 places** but the strength could be **much more** than **6** due to **several rotations** of the cylinder.

You are given an array of integers which represent the cylinder of the revolver where **0** means **empty** and **1** represents the **bullet**. There is **exactly** **one bullet** in the cylinder.

You will receive **another** array, this time of **strings** where **every index** is a **different player** and every **value** is the **strength** of the player and the **direction** to which the player **rotates** the cylinder.

The elements in the array will be separated by **spaces**. The **power** and the **direction** will be separated by a **comma** (”**,**”) e.g.: “**111,Left**”.

The **muzzle** **IS at** index **2** of the cylinder. If the **element** at that **position** is **1**, the **current player** **loses the game**.



**Note!** After a player **pulls the trigger** the cylinder **spins** with **one position** to the **right**.The **next player** starts spinning with the **new state** of the **cylinder**.

Assume that the order of the players is according to their places in the array. The player with index 0 shots first, index 1-second and so on…

Your task is to **go** **through all players**, and see if anyone will shoot himself with the **toy gun**.

### Input / Constraints

* First line – **6** **integers**, separated by a **space** (only 0 or 1, where 1 once and 0 five times) which represent the cylinder.
* Second line –**strings**, separated by a **space**, representing the strength and the direction to which **every player** rotates. The **strength** and **direction** for a **single player** are **separated** by a **comma**.

### Output

* If someone **loses the game** you should print on the console “Game over! Player {index} is dead.”, where **index** is the index of the player in the array. The game **ends** and so does your program.
* If no one lost, you should just print: “Everybody got lucky!”

## Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 0 0 0 0 1 0  10,Right 11,Left 140,Right 1,Left 30,Right | Game over! Player 0 is dead. |
| **Input** | **Output** |
| 0 0 1 0 0 0  1,Right 2,Left 5,Left | Game over! Player 1 is dead. |

## Increasing Crisis

You will be given **N** – an **integer**. On the next **N lines**, you will receive **sequences** of **integers**, **separated** by a **space**.

Your task is to add each sequence’s integers to a list, so that it forms an increasing sequence. If there are already elements **inside** the list, you must **find** the **right-most** element, **lower by value**, than the **first element** from the **given** **sequence**, and start **INSERTING**, the **sequence’s** **elements** at the **position**, **AFTER** the **found element**.

If the **increasing sequence** is **BROKEN**, **during** the **addition** of **new elements**, you must **IGNORE** the **element** that **breaks** it and **all others** **after it**. You must also **remove all elements** from the **list**, **AFTER** the **LAST ADDED** element.

When you process all input lines, you must print the **list’s elements**, separated by a **single space**.

### Constraints

* The list, to which you add the elements, is **initially** **empty**.
* The increasing sequence **CAN** have **EQUAL** elements.
* Do **NOT** sort the list in any order.

### Examples

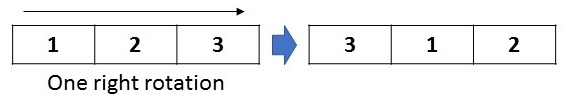
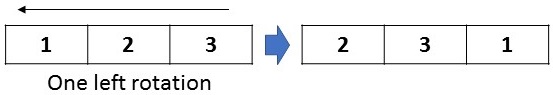
|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Comment** |
| 5  1 2 12 15 16  4 5 6  5 5 5  3 2  3 4 5 6 | 1 2 3 3 4 5 6 | The list is initially empty, so we **add** **[1, 2, 12, 15, 16]** to it.  After that we have **[4, 5, 6]**. We find the **right-most** element, **lower by value** than the **first element** (**4**), which is **2**, and we start **INSERTING** **AFTER** it.  The list is now [1, 2, 4, 5, 6, 12, 15, 16].  **+ 5 5 5 -> [1, 2, 4, 5, 5, 5, 5, 6, 12, 15, 16]**.  Of the **sequence [3, 2]** we **insert** **3**.  The **2nd element** of the sequence(**2**) **BREAKS** the **increasing sequence**, and so we **ignore** it, and we **remove** **all** elements **after** the **last added** (**3**), which results in  **[1, 2, 3]**.  Then we have **[3, 4, 5, 6] -> [1, 2, 3, 3, 4, 5, 6]** |
| **Input** | **Output** |
| 4  1 2  2 1  2 3  4 5 1 6 7 | 1 2 2 2 3 4 5 |

## Extremums \*

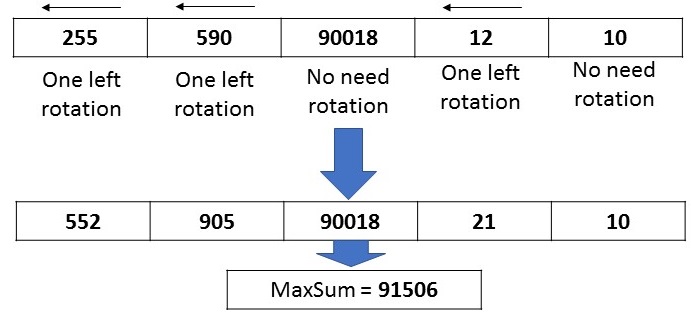
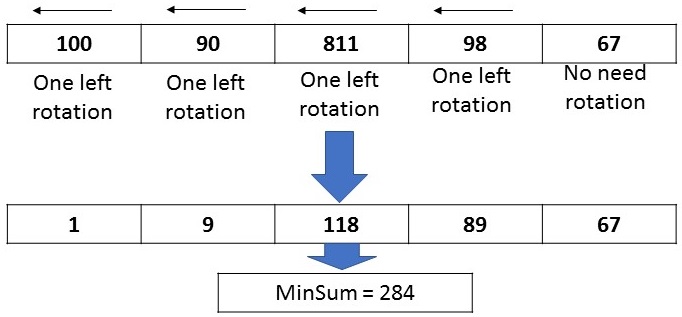
You are given a sequence of non-negative integers and a command (**Min** or **Max**).

Depending on the command, you must find the **maximum** or the **minimum** sum of **numbers** formed by the **rotations** of the **digits** of **every element** in the **sequence**.

You are allowed to use only rotations to form the numbers (**you cannot swap digits, only rotations**). These rotations are similar to the array rotations e.g.:



When you are forming the sum you should use all the elements in the sequence.



**Note!** If yourotate **900 → 009, 009** isa **valid number** which is equal to **9.**

### Input / Constraints

* On the first line you will receive **integers**, **separated** by a **space**, in **range** [**0…100000**];
* On the second line, you will receive a **string** containing the **one** of the words “**Min**” or “**Max**”.

### Output

* The numbers which form the **result** **min** / **max** **sum** separated by **comma** and **space** (‘, ’);
* A **single integer** **number** representing the **minimum** or the **maximum** **sum** of the **numbers**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 255 590 90018 12 10  Max | 552, 905, 90018, 21, 10  91506 |

|  |  |
| --- | --- |
| **Input** | **Output** |
| 100 90 811 98 67  Min | 1, 9, 118, 89, 67  284 |