**Java Script Problem Statements**

1. Write a function that takes in an array of integers and returns the integers that are either **palindromes** or **almost-palindromes**. An **almost-palindrome** is any integer that can be rearranged to form a palindrome.

For example, the numbers 677 and 338 are both **almost-palindromes**, since they can be rearranged to form 767 and 383, respectively.

**Examples**

palindromeSieve([443, 12, 639, 121, 3232]) ➞ [443, 121, 3232]

// Since 443 => 434; 121 is a palindrome; 3232 => 2332 or 3223

palindromeSieve([5, 55, 6655, 8787]) ➞ [5, 55, 6655, 8787]

// Single-digit numbers are automatically palindromes.

palindromeSieve([897, 89, 23, 54, 6197, 53342]) ➞ []

2. Create a function that takes an array and return the most frequently occuring element contained within it.

### Examples

findFrequent([3, 7, 3]) ➞ 3

findFrequent([null, "hello", true, null]) ➞ null

findFrequent([false, "up", "down", "left", "right", true, false]) ➞ false

3. This robot roams around a 2D grid. It starts at (0, 0) facing North. After each time it moves, the robot rotates 90 degrees clockwise. Given the amount the robot has moved each time, you have to calculate the robot's final position.

To illustrate, if the robot is given the movements 20, 30, 10, 40 then it will move:

* 20 steps North, now at (0, 20)
* 30 steps East, now at (30, 20)
* 10 steps South. now at (30, 10)
* 40 steps West, now at (-10, 10)

...and will end up at coordinates (-10, 10).

### Examples

trackRobot(20, 30, 10, 40) ➞ [-10, 10]

trackRobot() ➞ [0, 0]

// No movement means the robot stays at (0, 0).

trackRobot(-10, 20, 10) ➞ [20, -20]

// The amount to move can be negative.

4. An input string can be completed if additional letters can be added and no letters need to be taken away to match the word. Furthermore, the order of the letters in the input string must be the same as the order of letters in the final word.

Create a function that, given an input string, determines if the word can be completed.

### Examples

canComplete("butl", "beautiful") ➞ true

// We can add "ea" between "b" and "u", and "ifu" between "t" and "l".

canComplete("butlz", "beautiful") ➞ false

// "z" does not exist in the word beautiful.

canComplete("tulb", "beautiful") ➞ false

// Although "t", "u", "l" and "b" all exist in "beautiful", they are incorrectly ordered.

canComplete("bbutl", "beautiful") ➞ false

// Too many "b"s, beautiful has only 1.

5. You are playing a game of JavaScript & Jackalopes with your friends and need to roll dice as part of the game. None of you actually own dice, but you do have a computer handy!

You'll be given a string representing the number of dice to roll, how many faces each die has, and a "modifier" to apply to the final result after adding up all the dice. For example, rolling a single six-sided die with no modifier might be represented by the string "1d6" — one die with six sides and values ranging from 1 through 6. If you wanted to add 2 to the result of rolling the same die, you might represent that as "1d6+2".

Create a function that takes a string representing a set of dice to be rolled as an argument, and returns an array of two numbers representing the minimum and maximum possible values that could be achieved.

### Examples

diceRange("1d6") ➞ [1, 6]

// If a modifier is not given, assume that nothing will be

// added to/subtracted from the results.

diceRange("1d6+2") ➞ [3, 8]

diceRange("d6") ➞ [1, 6]

// If a number of dice is not provided, assume only one is

// being rolled.

diceRange("d6-2") ➞ [-1, 4]

// If a modifier is negative, the resulting values may be

// negative as well.

diceRange("2d6") ➞ [2, 12]

diceRange("2d6-1") ➞ [1, 11]

// The modifier should be added to/subtracted from the

// final result after rolling all the dice and adding up their

// results, not applied to each roll!

diceRange("0d6+1") ➞ [1, 1]

// If you roll no dice, the result will only be whatever the

// modifier's value is with no randomness.

6. Write a sorting function that takes in an array of names and sorts them **by last name** either alphabetically (ASC) or reverse-alphabetically (DESC).

### Examples

sortContacts([

"John Locke",

"Thomas Aquinas",

"David Hume",

"Rene Descartes"

], "ASC") ➞ [

"Thomas Aquinas",

"Rene Descartes",

"David Hume",

"John Locke"

]

// Aquinas (A) < Descartes (D) < Hume (H) < Locke (L)

sortContacts([

"Paul Erdos",

"Leonhard Euler",

"Carl Gauss"

], "DESC") ➞ [

"Carl Gauss",

"Leonhard Euler",

"Paul Erdos"

]

// Gauss (G) > Euler (EU) > Erdos (ER)

sortContacts([], "DESC") ➞ []

sortContacts(null, "DESC") ➞ []

sortContacts(undefined, "DESC") ➞ []

7. Practice Regular Expressions Concepts

Email iD verification using regular expression. Need to accept valid Email ID.

8. Create a function that takes in an array of grass heights and a **variable** sequence of lawn mower cuts and outputs the array of successive grass heights.

If **after a cut**, any single element in the array reaches zero or negative, return "Done" instead of the array of new heights.

### Examples

cuttingGrass([5, 6, 7, 5], 1, 2, 1)

➞ [[4, 5, 6, 4], [2, 3, 4, 2], [1, 2, 3, 1]]

cuttingGrass([4, 4, 4, 4], 1, 1, 1, 1)

➞ [[3, 3, 3, 3], [2, 2, 2, 2], [1, 1, 1, 1], "Done"]

cuttingGrass([8, 9, 9, 8, 8], 2, 3, 2, 1)

➞ [[6, 7, 7, 6, 6], [3, 4, 4, 3, 3], [1, 2, 2, 1, 1], "Done"]

cuttingGrass([1, 0, 1, 1], 1, 1, 1) ➞ ["Done", "Done", "Done"]