Array Related

1. **Write a function which can concatenate 2 arrays. If only one array is passed it will duplicate it**

```
function mergeArray(arr1, arr2 = arr1){
   return [...arr1, ...arr2];
}
```

```
function mergeArray(arr1, arr2 = arr1){
   return arr1.concat(...arr2);
}
```

```
function mergeArray(arr1, arr2 = arr1){
   arr1.push(...arr2);
   return arr1;
}
```

2. Merge Sorted Array

You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.
Merge nums1 and nums2 into a single array sorted in non-decreasing order.
The final sorted array should not be returned by the function, but instead be stored inside the array nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

Example 1:
Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]
Explanation: The arrays we are merging are [1,2,3] and [2,5,6].
The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1.

Example 2:
Input: nums1 = [1], m = 1, nums2 = [], n = 0
Output: [1]
Explanation: The arrays we are merging are [1] and [].
The result of the merge is [1].

Example 3:
Input: nums1 = [0], m = 0, nums2 = [1], n = 1
Output: [1]
Explanation: The arrays we are merging are [] and [1].
The result of the merge is [1].
Note that because m = 0, there are no elements in nums1. The 0 is only there to ensure the merge result can fit in nums1.

```
function merge(nums1, m, nums2, n) {
   nums1.splice(m)
   for (let i = 0; i < n; i++)
      nums1.push(nums2[i])

   return nums1.sort((a, b) => a - b)
};
```

3. **Given an integer array nums and an integer val, remove all occurrences of val in nums in-place. The order of the elements may be changed. Then return the number of elements in nums which are not equal to val.**
   Consider the number of elements in nums which are not equal to val be k, to get accepted, you need to do the following things:
   - **Change the array nums such that the first k elements of nums contain the elements which are not equal to val. The remaining elements of nums are not important as well as the size of nums.**
   - **Return k.**

   Example 1:

   Input: nums = [3,2,2,3], val = 3
   Output: 2, nums = [2,2,_,_]
   Explanation: Your function should return k = 2, with the first two elements of nums being 2.
   It does not matter what you leave beyond the returned k (hence they are underscores).

   Example 2:

   Input: nums = [0,1,2,2,3,0,4,2], val = 2
   Output: 5, nums = [0,1,4,0,3,_,_,_]
   Explanation: Your function should return k = 5, with the first five elements of nums containing 0, 0, 1, 3, and 4.
   Note that the five elements can be returned in any order.
   It does not matter what you leave beyond the returned k (hence they are underscores).

```
function removeElement(nums, val) {
 let j=0
 for(let i=0;i<nums.length;i++){
    if(nums[i]!==val) nums[j++]=nums[i]
 }
 return j;
};
```

4. **Write a function which can concatenate 2 arrays and keep distinct elements in concatenated array.**

```
function uniquefromArrays(arr1, arr2){
  return [...new Set([...arr1,...arr2])]
}
```

5. **Given an integer array nums sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears only once. The relative order of the elements should be kept the same. Then return the number of unique elements in nums.**
   Consider the number of unique elements of nums to be k, to get accepted, you need to do the following things:
   - **Change the array nums such that the first k elements of nums contain the unique elements in the order they were present in nums initially. The remaining elements of nums are not important as well as the size of nums.**
   - **Return k.**

   Example 1:

   Input: nums = [1,1,2]
   Output: 2, nums = [1,2,_]
   Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.
   It does not matter what you leave beyond the returned k (hence they are underscores).

   Example 2:

   Input: nums = [0,0,1,1,1,2,2,3,3,4]
   Output: 5, nums = [0,1,2,3,4,_,_,_,_,_]
   Explanation: Your function should return k = 5, with the first five elements of nums being 0, 1, 2, 3, and 4 respectively.
   It does not matter what you leave beyond the returned k (hence they are underscores).

```
function removeDuplicates(nums) {
  for (let i = 0; i < nums.length; i++) {
       if (nums[i] === nums[i + 1]) {
          nums.splice(i, 1);
          i--;
       }
    }
  }
```

6. **Separate strings and numbers from an array in two different arrays and return [arrStrings,arrNumbers]**

```
function segregateArrays(arr) {
  const arrStrings = []
  const arrNumbers = []
  for (let item of arr) {
    if (typeof item === "number") arrNumbers.push(item)
    if (typeof item === "string") arrStrings.push(item)
  }
  return [arrStrings, arrNumbers]
}
```

7. **Given an array nums of size n, return the majority element.**
   The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

   Example 1:
   Input: nums = [3,2,3]
   Output: 3

   Example 2:
   Input: nums = [2,2,1,1,1,2,2]
   Output: 2

```
function majorityElement (nums) {
   let numMap = {}
   for (let num of nums) {
      numMap[num] = (numMap[num] || 0) +1
      if (numMap[num] >= Math.floot(nums.length / 2)) return num
   }
}
```

8. **Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.**

   Example 1:

   Input: nums = [1,2,3,4,5,6,7], k = 3
   Output: [5,6,7,1,2,3,4]
   Explanation:
   rotate 1 steps to the right: [7,1,2,3,4,5,6]
   rotate 2 steps to the right: [6,7,1,2,3,4,5]
   rotate 3 steps to the right: [5,6,7,1,2,3,4]

   Example 2:

   Input: nums = [-1,-100,3,99], k = 2
   Output: [3,99,-1,-100]
   Explanation:
   rotate 1 steps to the right: [99,-1,-100,3]
   rotate 2 steps to the right: [3,99,-1,-100]

```
function rotate (nums, k) {
    const rotations = k % nums.length;
    if (rotations === 0)
      return nums; // No rotations needed

    const removedElements = nums.splice(nums.length - rotations);
    nums.unshift(...removedElements);
};
```

9. Find the intersection, symmetric difference and asymmetric difference between two arrays

```
function intersection(arr1, arr2) {


  const set2 = new Set(arr2);
  return [...new Set(arr1)].filter(item => set2.has(item));
}

function asymmetricDiff(arr1, arr2) {
  const set2 = new Set(arr2);
  return [...new Set(arr1)].filter(item => !set2.has(item));
}

function symmetricDiff(arr1, arr2) {
  const set1 = new Set(arr1);
  const set2 = new Set(arr2);

  return [
    ...[...set1].filter(item => !set2.has(item)),
    ...[...set2].filter(item => !set1.has(item))
  ];
}
```

10. Given two 0-indexed integer arrays nums1 and nums2, return *a list* answer *of size* 2 *where:*

- **answer[0] *is a list of all distinct integers in* nums1 *which are not present in* nums2.**
- **answer[1] *is a list of all distinct integers in* nums2 *which are not present in* nums1.**

Note that the integers in the lists may be returned in any order.
Example 1:
Input: nums1 = [1,2,3], nums2 = [2,4,6]
Output: [[1,3],[4,6]]

Example 2:
Input: nums1 = [1,2,3,3], nums2 = [1,1,2,2]
Output: [[3],[]]
Constraints:

- **1 <= nums1.length, nums2.length <= 1000**
- **-1000 <= nums1[i], nums2[i] <= 1000**

```
function findDifference (nums1, nums2) {
   const result1 = [...new Set(nums1.filter((num) => !nums2.includes(num)))]
   const result2 = [...new Set(nums2.filter((num) => !nums1.includes(num)))]

   return [result1, result2]
};
```

11. Write the number of ways to empty an array

```
const arr=[1,2.3]
arr.length=0                    //Method-1
arr=[]                          //Method-2 (works for let declaration)
arr.splice(0,arr.length)        //Method-3
const emptyArray=()=>{          //Method-4
while(arr.length){
arr.pop()
}
}
```

12. Find only truthy values from an array

```
function truthyValues(arr){
return arr.filter(item=>item) //For falsy values use !
}
```

13. Find sum, average and median of elements in an array

```
function average(arr){
return arr.reduce((acc,currValue,index,{length})=>
acc+currValue/length,0)
}

function sum(arr){
return arr.reduce((acc,currValue)=>acc+currValue,0)
}

function median(arr) {
  if (arr.length % 2 !== 0) {
    return arr[Math.round(arr.length / 2) - 1]
  return (arr[Math.round(arr.length / 2) - 1] + arr[Math.round(arr.length / 2)]) / 2
  }
}
```

14. Sorting an array in ascending or descending order

```
function sortAscending(arr){
return arr.sort((a,b)=>a-b)  //For Strings a.localeCompare(b)
}

function sortDescending(arr){
return arr.sort((a,b)=>b-a)  //For strings b.localeCompare(a)
}
```

15. How to remove duplicates from an array

```
function unique(arr){
return [...new Set(arr)]
}
```

```
function unique(arr){
return arr.filter((item,index)=>arr.indexOf(item)===index)
}
```

16. Find all repeating elements in an array

```
function repeating(arr){
return […new Set(arr.filter((item,index)=>arr.indexOf(item)!==index))]
}
```

17. Find first Repeating and first unique element in an array

```
function firsUnique(arr){
for(let item of arr){
if(arr.indexOf(item)===arr.lastIndexOf(item))
return item;
}
}
```

```
function firstRepeating(arr){
for(let item of arr){
if(arr.indexOf(item)!==arr.lastIndexOf(item))
return item;
}
}
```

18. Count number of occurrences of elements in an array and find the element which has maximum occurrences if two elements have some number of occurrences return the first occurring element in an array

```
function maxOcuurences(arr) {
  let occurrences = {}
  let max = 0

  for (let char of arr) {
    occurrences[char] = (occurrences[char] || 0) + 1
    if (occurrences[char] > max) max = occurrences[char]
  }

  for (let char in occurrences) {
    if (occurrences[char] === max) return char
  }
}
```

19. Count number of occurrences of elements in an array and sort the array in the descending order according to occurrences.

```
function maxOccurencesSort(arr){
let occurrences={}

for(let item of arr){
occurrences[item]=(occurrences[item] || 0)+1
}
 return Object.keys(occurrences)
    .sort((a, b) => occurrences[b] - occurrences[a])
    .map(key => isNaN(key) ? key : Number(key))
}
```

20. Bubble Sorting

```
function bubbleSort(arr) {
for (let i = 0; i < arr.length; i++) {
for (let j = 0; j < (arr.length - i -1);j ++) {
if (arr[j] > arr[j + 1]) { // Use < for descending)
        let temp = arr[j]
        arr[j] = arr[j + 1]
        arr[j + 1] = temp
      }
    }
  }
  return arr;
}
```

21. Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this in-place without making a copy of the array.

Example 1:
Input: nums = [0,1,0,3,12]
Output: [1,3,12,0,0]

Example 2:
Input: nums = [0]
Output: [0]

```
function moveZeroes(nums) {
  let nonZeroIndex = 0;

  for (let i = 0; i < nums.length; i++) {
    if (nums[i] != 0) {
      nums[nonZeroIndex] = nums[i];
      nonZeroIndex++;
    }
  }

  for (let i = nonZeroIndex; i < nums.length; i++)
    nums[i] = 0;

};
```

22. Find the number of occurrences of minimum value and maximum value in the array

```
const min = Math.min(...arr);
minArr = arr.filter((value) => value === min);
minArr.length;

const max = Math.max(...arr);
maxArr = arr.filter((value) => value === max);
maxArr.length;
```

23. Sort the given array of objects in ascending order of their authors

```
const books = [
    { name: "Harry Potter", author: "Joanne Rowling" },
    { name: "Warcross", author: "Marie Lu" },
    { name: "The Hunger Games", author: "Suzanne Collins" },
]

books.sort((book1, book2) =>
    book2.author.localeCompare(book1.author)
)
```

24. Write a code to eliminate duplicate objects in an array where each object has an 'id' property which can be used to identify the object.

const arr = [{ id: 1, name: 'emp20'},{id: 2,name: 'emp21'},{id: 2,name: 'emp22'},{id: 3,name: 'emp24'}];

```
const unique = arr.filter(
  (items, index) =>
  arr.findIndex((item) => item.id == items.id) === index
);
```

```
const unique = [];
for (const item of arr) {
  const isDuplicate = unique.find((items) => items.id === item.id);
  if (!isDuplicate) {
    unique.push(item);
  }
}
```

25. Produce the below output:

[1,2,3,5,6]=> [[1,6],[2,5],[3,3]]

[1,2,3,4]=> [[1,4],[2,3]]

```
function combinedElements(arr){
const combo=[]
for (let i=0;i<arr.length/2;i++)
    combo.push([arr[i],arr[arr.length-i-1]])

return combo
}
```

26. Produce the below output:

[1,2]+[2,3]=[3,5]

[1,2,4]+[3,4]=[4,6,4]

```
function sumElements(arr1, arr2) {
  let result = [];
  let maxLength = Math.max(arr1.length, arr2.length);

  for (let i = 0; i < maxLength; i++) {
    let val1 = arr1[i] || 0;
    let val2 = arr2[i] || 0;
    result.push(val1 + val2);
  }

  return result;
}
```

27. Remove object based from an array on property

```
let myArray = [
  { field: "id", operator: "eq" },
  { field: "cStatus", operator: "eq" },
  { field: "money", operator: "eq" },
];

myArray = myArray.filter((item)=> item.field !== "money");
```

28. Group items on the basis of age of given array of object

```
let people = [{name: "jack",age: 21}, {name: "jill",age: 20}, {name: "john",age: 21},
                {name: "tom",age: 23}]

function groupByAge(people) {
  let group = {}

  for (let item of people) {
    if (group[item.age])
      group[item.age].push(item)
    else
      group[item.age] = [item]
  }
  return group
}
```

```
let groupByAge=Object.groupBy(people,({age})=>age)
console.log(groupByAge)
```

29. Count the occurrence of keys and convert the result into array of objects where each object belongs to one key and it's occurrence (count).

Example

```
[
   { language: 'JavaScript' }, { language: 'JavaScript' }, { language: 'TypeScript' }, { language: 'C++'
}
]
```

SHOULD BE CONVERTED TO =

```
[
{ language: 'JavaScript', count: 2 },
{ language: 'C++', count: 1 },
{ language: 'TypeScript', count: 1 }
]
```

```
function groupBy(arr) {
  let result = {}

  for (let item of arr) {
    if (result[item.language])
      result[item.language] = {
        ...result[item.language],
        count: result[item.language].count + 1
      }
    else
      result[item.language] = {
        language: item.language,
        count: 1
      }
  }
  return Object.values(result)
}
```

30. Count the occurrence of keys and convert the result into array of objects where each object belongs to one key and it's occurrence (count).

Example

```
let skillsArray = [
  { skill: 'css', user: 'Bill' },
  { skill: 'javascript', user: 'Chad' },
  { skill: 'javascript', user: 'Bill' },
  { skill: 'css', user: 'Sue' },
  { skill: 'javascript', user: 'Sue' },
  { skill: 'html', user: 'Sue' }
];
```

Convert it into result of the following form:

```
[
  { skill: 'javascript', user: [ 'Chad', 'Bill', 'Sue' ], count: 3 },
  { skill: 'css', user: [ 'Sue', 'Bill' ], count: 2 },
  { skill: 'html', user: [ 'Sue' ], count: 1 }
]
```

```
function groupBy(arr) {
  let result = {}

  for (let item of arr) {
    if (result[item.skill]) {
      result[item.skill] = {
        ...result[item.skill],
        user: [...result[item.skill].user, item.user],
        count: result[item.skill].count + 1
      }
    } else {
      result[item.skill] = {
        skill: item.skill,
        user: [item.user],
        count: 1
      }

    }
  }
  return Object.values(result)
}
```

31. Write a function that return true if every value in arr1 has corresponding value squared in array2. The frequency of values must be same.

```
function squaredValues(arr1, arr2) {
  if (arr1.length !== arr2.length) return false
  let ar1 = [...arr1].sort((a, b) => a - b)
  let ar2 = [...arr2].sort((a, b) => a - b)

  for (let i = 0; i < arr1.length; i++) {
    if (ar1[i] * ar1[i] !== ar2[i]) return false
  }
  return true
}
```

32. Find the max count of consecutive 1's in an array ?

```
function maxcOnes(arr) {
  let maxCount = 0;
  let currCount = 0;

  for (let item of arr) {
    if (item === 1) {
      currCount = currCount + 1;
      maxCount = Math.max(currCount, maxCount)
    } else
      currCount = 0;
  }
  return maxCount;
}
```

33. Given an array of integers nums and an integer target, return the indices of the two numbers that add up to the target.

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

```
function twoSum(nums, target) {
  for (let i = 0; i < nums.length; i++) {
    for (let j = i + 1; j < nums.length; j++) {
      if (nums[i] + nums[j] === target)
        return [i, j];
    }
  }
}
```

```
function twoSum(nums, target) {
  const numMap = {};
  for (let i = 0; i < nums.length; i++) {
    const complement = target - nums[i];
    if (numMap[complement])
      return [numMap[complement], i];
    numMap[nums[i]] = i;
  }
```

34. Given an array of integers nums and an integer target, return the pair of numbers that add up to the target.
input = [1, 2, 3, 4, 5, 6, 7, 8, 9]; target = 10; output = [[4, 6], [3, 7], [2, 8], [1, 9]]

```
function pairSum(input, target) {
  const set = new Set()
  const result = []
  for (let i = 0; i < input.length; i++) {
    const complement = target - input[i]
    if (set.has(complement)) result.push([complement, input[i]])
    set.add(input[i])
  }
  return result
}
```

35. Write a function sumOfThirds(arr) which should return a sum of every third number in the array, starting from the first one.

```
  function sumOfThirds(arr) {
  let sum = 0
  for (let i = 0; i < arr.length; i = i + 3)
  sum = sum + arr[i]
  return sum
}
```

36. To achieve the desired output where each element is the product of the next two elements in the input array: const arr = [3, 4, 5] //output: [20, 15, 12]

```
function product(arr){
const output = [];
for (let i = 0; i < arr.length; i++) {
  const nextIndex1 = (i + 1) % arr.length;
  const nextIndex2 = (i + 2) % arr.length;
  output.push(arr[nextIndex1] * arr[nextIndex2]);
}
return output;
}
```

37. Given an integer array nums, return an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i].

Input: nums = [1,2,3,4]
Output: [24,12,8,6]

```
function productSelf(nums) {
  const res = [1];
  let right = 1;

  for (let i = 1; i < nums.length; i++) {
    res[i] = res[i - 1] * nums[i - 1];
  }
  for (let j = nums.length - 1; j >= 0; j--) {
    res[j] *= right;
    right *= nums[j];
  }
  return res;
}
```

38. Given an integer array nums and an integer k, return true if there are two distinct indices i and j in the array such that nums[i] == nums[j] and abs(i - j) <= k.

Example 1:
Input: nums = [1,2,3,1], k = 3
Output: true

Example 2:
Input: nums = [1,0,1,1], k = 1
Output: true

Example 3:
Input: nums = [1,2,3,1,2,3], k = 2
Output: false

```
function containsNearbyDuplicate (nums, k) {
   const map = {};
   for (let i = 0; i < nums.length; i++) {
      if (i - map[nums[i]] <= k) {
         return true;
      }
      map[nums[i]] = i;
   }
   return false;
};
```

39. Given an unsorted array of integers nums, return the length of the longest consecutive elements sequence.
You must write an algorithm that runs in O(n) time.

Example 1:
Input: nums = [100,4,200,1,3,2]
Output: 4

Example 2:
Input: nums = [0,3,7,2,5,8,4,6,0,1]
Output: 9

```
function longestConsecutive (nums) {
    if (!nums.length) return 0;

    const set = new Set(nums);
    let max = 0;

    for (const num of set) {
        if (set.has(num - 1)) continue;

        let currNum = num;
        let currCount = 1;

        while (set.has(currNum + 1)) {
            currNum++;
            currCount++;
        }

        max = Math.max(max, currCount);
    }

    return max;
}
```

40. You are given a large integer represented as an integer array digits, where each digits[i] is the ith digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.
Increment the large integer by one and return the resulting array of digits.

Example 1:

Input: digits = [1,2,3]
Output: [1,2,4]
Explanation: The array represents the integer 123.
Incrementing by one gives 123 + 1 = 124.
Thus, the result should be [1,2,4].

Example 2:

Input: digits = [4,3,2,1]
Output: [4,3,2,2]
Explanation: The array represents the integer 4321.
Incrementing by one gives 4321 + 1 = 4322.
Thus, the result should be [4,3,2,2].

```
function plusOne(digits){
for (let i = digits.length - 1; i >= 0; i--){
  if (digits[i] < 9) {
    digits[i]++;
    return digits;
  } else {
    digits[i] = 0;
  }
}
digits.unshift(1);
return digits;
}
```

41. Given a non-empty array of integers nums, every element appears twice except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

Example 1:

Input: nums = [2,2,1]
Output: 1

Example 2:
Input: nums = [4,1,2,1,2]
Output: 4

Example 3:

Input: nums = [1]
Output: 1

```
function singleNumber(nums) {
   const [num]=nums.filter((letter) => nums.indexOf(letter) === nums.lastIndexOf(letter))
   return num
};
```

42. Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

Example 1:
Input: nums = [1,2,3,1]
Output: true

Example 2:
Input: nums = [1,2,3,4]
Output: false

Example 3:
Input: nums = [1,1,1,3,3,4,3,2,4,2]
Output: true

```
function containsDuplicate(nums) {
  const unique = [];
  for (const item of nums) {
  const isDuplicate = unique.find((items) => items === item);
  if (isDuplicate!==undefined)
    return true;
  unique.push(item)
}
 return false ;
}
```

43. Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.
An input string is valid if:
Open brackets must be closed by the same type of brackets.
Open brackets must be closed in the correct order.
Every close bracket has a corresponding open bracket of the same type.

Example 1:
Input: s = "()"
Output: true
Example 2:
Input: s = "()[]{}"
Output: true
Example 3:
Input: s = "(]"
Output: false

```
function isValid (s) {
    s = s.replace(/\s+/g, '');
    if (s.length % 2 != 0) return false;
    const stack = [];
    const map = {
        '(': ')',
        '[': ']',
        '{': '}'

    };
    for (let i = 0; i < s.length; i += 1) {
        if (map[s[i]]) {
            stack.push(map[s[i]]);
        } else if (s[i] !== stack.pop()) {
            return false;
        }
    }
    return stack.length === 0;
}
```

44. Given an array nums containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array.

Example 1:
Input: nums = [3,0,1]
Output: 2
Explanation: n = 3 since there are 3 numbers, so all numbers are in the range [0,3].
2 is the missing number in the range since it does not appear in nums.

Example 2:
Input: nums = [0,1]
Output: 2
Explanation: n = 2 since there are 2 numbers, so all numbers are in the range [0,2].
2 is the missing number in the range since it does not appear in nums.

```
function missingNumber(nums) {
  const length = nums.length;
  let sum = ((length + 1) * length) / 2;

  for (let i = 0; i < length; i++) {
    sum = sum - nums[i];
  }

  return sum;
}
```

45. Write a function called findMissingLetter that takes in an array of consecutive (increasing) letters as input and returns the missing letter in the array.

Examples

findMissingLetter(['a', 'b', 'c', 'd', 'f']); // => "e"
findMissingLetter(['O', 'Q', 'R', 'S']); // => "P"
findMissingLetter(['t', 'u', 'v', 'w', 'x', 'z']); // => "y"

Constraints

The input array will only contain letters in one case (lower or upper)
If no letters are in the array, return an empty string

```
function findMissingLetter(arr) {
  // Find the char code of the first letter in the array
  let start = arr[0].charCodeAt(0);

  // Loop through the array
  for (let i = 1; i < arr.length; i++) {
    // Find the char code of the current letter in the array
    const current = arr[i].charCodeAt(0);

    // If the difference between the current char code and the start char code is greater than 1, return
the letter that is missing

    if (current - start > 1) {
      // Convert the char code to a letter
      return String.fromCharCode(start + 1);
    }

    // Update the start char code
    start = current;
  }

  // If no letter is missing, return an empty string
  return '';
```

46. Write a function called sumOfEvenSquares that takes an array of numbers and returns the sum of the squares of the even numbers in the array.

Examples
sumOfEvenSquares([1, 2, 3, 4, 5]); // 20 (2^2 + 4^2)
sumOfEvenSquares([-1, 0, 1, 2, 3, 4]); // 20 (0^2 + 2^2 + 4^2)
sumOfEvenSquares([]); // 0

```
function sumOfEvenSquares(numbers) {
  const evenSquares = numbers
    .filter((num) => num % 2 === 0)
    .map((num) => num ** 2)
    .reduce((sum, square) => sum + square, 0);
  return evenSquares;
}
```

47. Write a function called calculateTotalSalesWithTax that takes in an array of product objects, along with the tax rate, and returns the total sales amount including tax.

Examples

```
calculateTotalSalesWithTax(
   [
   { name: 'Apple', price: 0.5, quantity: 10 },
   { name: 'Banana', price: 0.3, quantity: 20 },
   { name: 'Orange', price: 0.6, quantity: 15 },
 ],
 8
); // 21.6 (20 + 8% tax)
```

Constraints

- **The input array will contain at least one product object.**
- **The price and quantity values will be positive numbers.**
- **The tax rate will be a positive number less than 100.**
- **Round to 2 decimal places.**

```
function calculateTotalSalesWithTax(products,taxRate) {
   return products.reduce((acc,item)=>acc+item.quantity*item.price,0)*(1+taxRate/100);
}
```

48. Given a string of words, you need to find the highest scoring word. Each letter of a word scores points according to its position in the alphabet: a = 1, b = 2, c = 3, and so on.
You need to return the highest scoring word as a string.
If two words score the same, return the word that appears earliest in the original string.
All letters will be lowercase and all inputs will be valid.

Examples:

```
highestScoringWord('man i need a taxi up to ubud'); // 'taxi'
highestScoringWord('what time are we climbing up the volcano'); // 'volcano'
highestScoringWord('take me to semynak'); // 'semynak'
```

Constraints

- **The input string will only contain lowercase letters and spaces. It can not include numbers, special characters, or punctuation.**

```
function highestScoringWord(str) {
  const words = str.split(' ');

  const scores = words.map((word) =>
    Array.from(word).reduce(
      (score, letter) => score + letter.charCodeAt(0) - 96,
      0
    ));

  const highestScore = Math.max(...scores);
  const highestIndex = scores.indexOf(highestScore);
  return words[highestIndex];
}
```

49. You are given an array of car objects, each containing information about a car's make, model, year, and mileage. Your goal is to perform some analysis on the car mileage data using high order array methods.

Implement a function called analyzeCarMileage which takes in an array of car objects and performs the following tasks:

- **Calculate the average mileage of all cars.**
- **Find the car with the highest mileage.**
- **Find the car with the lowest mileage.**
- **Calculate the total mileage of all cars combined.**

The function should return an object containing the calculated values as properties.
Here is an object that you can use to test your function in the run file:
Examples

```
const cars = [
  { make: 'Toyota', model: 'Corolla', year: 2020, mileage: 25000 },
  { make: 'Honda', model: 'Civic', year: 2019, mileage: 30000 },
  { make: 'Ford', model: 'Mustang', year: 2021, mileage: 15000 },
];

const analysis = analyzeCarMileage(cars);
console.log(analysis);
// Output:
// {
//   averageMileage: 23333.33,
//   highestMileageCar: { make: "Honda", model: "Civic", year: 2019, mileage: 30000 },
//   lowestMileageCar: { make: "Ford", model: "Mustang", year: 2021, mileage: 15000 },
//   totalMileage: 70000
// }
```

Constraints

- **The input array cars will contain at most 100 car objects.**
- **Each car object's mileage property will be a positive integer.**
- **Result should be rounded to 2 decimal places.**

```
function analyzeCarMileage(cars) {
  // Get the total mileage of all cars by adding the mileage of each car to the sum
  const totalMileage = cars.reduce((sum, car) => sum + car.mileage, 0);

  // Get the average mileage by dividing the total mileage by the number of cars
  const averageMileage = totalMileage / cars.length;

  // Get the car with the highest mileage by comparing the mileage of each car to the highest mileage so far
  const highestMileageCar = cars.reduce(
    (highest, car) => (car.mileage > highest.mileage ? car : highest),
    cars[0]
  );

  // Get the car with the lowest mileage by comparing the mileage of each car to the lowest mileage so far
  const lowestMileageCar = cars.reduce(
    (lowest, car) => (car.mileage < lowest.mileage ? car : lowest),
    cars[0]
  );
```

```
  // Return an object with the average mileage, the car with the highest mileage, the car with the lowest mileage, and the total mileage
  return {
    averageMileage: parseFloat(averageMileage.toFixed(2)),
    highestMileageCar,
    lowestMileageCar,
    totalMileage,
  };
}
```

50. Given an array arr and a chunk size size, return a chunked array.
A chunked array contains the original elements in arr, but consists of subarrays each of length size. The length of the last subarray may be less than size if arr.length is not evenly divisible by size.
You may assume the array is the output of JSON.parse. In other words, it is valid JSON.
Please solve it without using lodash's _.chunk function.

Example 1:
Input: arr = [1,2,3,4,5], size = 1
Output: [[1],[2],[3],[4],[5]]
Explanation: The arr has been split into subarrays each with 1 element.

Example 2:
Input: arr = [1,9,6,3,2], size = 3
Output: [[1,9,6],[3,2]]
Explanation: The arr has been split into subarrays with 3 elements. However, only two elements are left for the 2nd subarray.

Example 3:
Input: arr = [8,5,3,2,6], size = 6
Output: [[8,5,3,2,6]]
Explanation: Size is greater than arr.length thus all elements are in the first subarray.

Example 4:
Input: arr = [], size = 1
Output: []
Explanation: There are no elements to be chunked so an empty array is returned.

Constraints:

- **arr is a valid JSON array**
- **2 <= JSON.stringify(arr).length <= 10^5**
- **1 <= size <= arr.length + 1**

```
function chunk (arr, size) {
   const result = [];
   for (let i = 0; i < arr.length; i += size) {
      result.push(arr.slice(i, i + size));
   }
   return result;
}
```

51. Given two arrays arr1 and arr2, return a new array joinedArray. All the objects in each of the two inputs arrays will contain an id field that has an integer value.
joinedArray is an array formed by merging arr1 and arr2 based on their id key. The length of joinedArray should be the length of unique values of id. The returned array should be sorted in ascending order based on the id key.
If a given id exists in one array but not the other, the single object with that id should be included in the result array without modification.
If two objects share an id, their properties should be merged into a single object:

- **If a key only exists in one object, that single key-value pair should be included in the object.**
- **If a key is included in both objects, the value in the object from arr2 should override the value from arr1.**

Example 1:
Input:
arr1 = [
{"id": 1, "x": 1},
{"id": 2, "x": 9}
],
arr2 = [
{"id": 3, "x": 5}
]
Output:
[
{"id": 1, "x": 1},
{"id": 2, "x": 9},
{"id": 3, "x": 5}
]
Explanation: There are no duplicate ids so arr1 is simply concatenated with arr2.

Example 2:
Input:
arr1 = [
{"id": 1, "x": 2, "y": 3},
{"id": 2, "x": 3, "y": 6}
],
arr2 = [
{"id": 2, "x": 10, "y": 20},
{"id": 3, "x": 0, "y": 0}
]
Output:
[
{"id": 1, "x": 2, "y": 3},
{"id": 2, "x": 10, "y": 20},
{"id": 3, "x": 0, "y": 0}
]
Explanation: The two objects with id=1 and id=3 are included in the result array without modifiction. The two objects with id=2 are merged together. The keys from arr2 override the values in arr1.

Example 3:
Input:
arr1 = [

{"id": 1, "b": {"b": 94},"v": [4, 3], "y": 48}
]
arr2 = [
{"id": 1, "b": {"c": 84}, "v": [1, 3]}
]
Output: [
{"id": 1, "b": {"c": 84}, "v": [1, 3], "y": 48}
]
Explanation: The two objects with id=1 are merged together. For the keys "b" and "v" the values from arr2 are used. Since the key "y" only exists in arr1, that value is taken form arr1.

Constraints:

- **arr1 and arr2 are valid JSON arrays**
- **Each object in arr1 and arr2 has a unique integer id key**
- **2 <= JSON.stringify(arr1).length <= 10^6**
- **2 <= JSON.stringify(arr2).length <= 10^6**

```
function join(arr1, arr2) {
  const map = {};

  arr1.forEach(item => {
    map[item.id] = { ...item };
  });

  arr2.forEach(item => {
    if (map[item.id])
      map[item.id] = { ...map[item.id], ...item };
    else
      map[item.id] = { ...item };
  });

  const result = Object.values(map).sort((a, b) => a.id - b.id);

  return result;
};
```

52. Write code that enhances all arrays such that you can call the array.last() method on any array and it will return the last element. If there are no elements in the array, it should return -1. You may assume the array is the output of JSON.parse.

Example 1:
Input: nums = [null, {}, 3]
Output: 3
Explanation: Calling nums.last() should return the last element: 3.

Example 2:
Input: nums = []
Output: -1
Explanation: Because there are no elements, return -1.

Constraints:

- **arr is a valid JSON array**
- **0 <= arr.length <= 1000**

```
Array.prototype.last = function () {
   if (this.length === 0)
      return -1
   return this[this.length - 1]
};
```

53. Write code that enhances all arrays such that you can call the array.groupBy(fn) method on any array and it will return a grouped version of the array.
A grouped array is an object where each key is the output of fn(arr[i]) and each value is an array containing all items in the original array which generate that key.
The provided callback fn will accept an item in the array and return a string key.
The order of each value list should be the order the items appear in the array. Any order of keys is acceptable.
Please solve it without lodash's _.groupBy function.

Example 1:
Input:
array = [
{"id":"1"},
{"id":"1"},
{"id":"2"}
],
fn = function (item) {
return item.id;
}
Output:
{
"1": [{"id": "1"}, {"id": "1"}],
"2": [{"id": "2"}]
}
Explanation:
Output is from array.groupBy(fn).
The selector function gets the "id" out of each item in the array.
There are two objects with an "id" of 1. Both of those objects are put in the first array.
There is one object with an "id" of 2. That object is put in the second array.

Example 2:
Input:
array = [
[1, 2, 3],
[1, 3, 5],
[1, 5, 9]
]
fn = function (list) {
return String(list[0]);
}
Output:
{
"1": [[1, 2, 3], [1, 3, 5], [1, 5, 9]]
}
Explanation:
The array can be of any type. In this case, the selector function defines the key as being the first element in the array.
All the arrays have 1 as their first element so they are grouped together.
{
"1": [[1, 2, 3], [1, 3, 5], [1, 5, 9]]
}

Example 3:
Input:
array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
fn = function (n) {
** return String(n > 5);**
}
Output:
{
** "true": [6, 7, 8, 9, 10],**
** "false": [1, 2, 3, 4, 5]**
}
Explanation:
The selector function splits the array by whether each number is greater than 5.

Constraints:

- **0 <= array.length <= 10^5**
- **fn returns a string**

```
Array.prototype.groupBy = function (fn) {
  const result = {}

  for (let item of this) {
    const key = fn(item)

    if (result[key])  result[key].push(item)
    else result[key] = [item]
  }

  return result
}
```

54. There are n kids with candies. You are given an integer array candies, where each candies[i] represents the number of candies the i^{th} kid has, and an integer extraCandies, denoting the number of extra candies that you have.

Return *a boolean array* result *of length* n, *where* result[i] *is* true *if, after giving the* i^{th} *kid all the* extraCandies*, they will have the greatest number of candies among all the kids, or* false *otherwise*.
Note that multiple kids can have the greatest number of candies.

Example 1:
Input: candies = [2,3,5,1,3], extraCandies = 3
Output: [true,true,true,false,true]
Explanation: If you give all extraCandies to:
- Kid 1, they will have 2 + 3 = 5 candies, which is the greatest among the kids.
- Kid 2, they will have 3 + 3 = 6 candies, which is the greatest among the kids.
- Kid 3, they will have 5 + 3 = 8 candies, which is the greatest among the kids.
- Kid 4, they will have 1 + 3 = 4 candies, which is not the greatest among the kids.
- Kid 5, they will have 3 + 3 = 6 candies, which is the greatest among the kids.

Example 2:
Input: candies = [4,2,1,1,2], extraCandies = 1
Output: [true,false,false,false,false]
Explanation: There is only 1 extra candy.
Kid 1 will always have the greatest number of candies, even if a different kid is given the extra candy.

Example 3:
Input: candies = [12,1,12], extraCandies = 10
Output: [true,false,true]

Constraints:
- **n == candies.length**
- **2 <= n <= 100**
- **1 <= candies[i] <= 100**
- **1 <= extraCandies <= 50**

```
function kidsWithCandies  (candies, extraCandies) {
  const result = []
  const max = Math.max(...candies)

   for (let i = 0; i < candies.length; i++)
      result.push(candies[i] + extraCandies >= max)

   return result
};
```

55. You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in adjacent plots.

Given an integer array flowerbed containing 0's and 1's, where 0 means empty and 1 means not empty, and an integer n, return true *if* n *new flowers can be planted in the* flowerbed *without violating the no-adjacent-flowers rule and* false *otherwise*.

Example 1:
Input: flowerbed = [1,0,0,0,1], n = 1
Output: true
Example 2:
Input: flowerbed = [1,0,0,0,1], n = 2
Output: false

Constraints:

- **1 <= flowerbed.length <= $2 * 10^4$**
- **flowerbed[i] is 0 or 1.**
- **There are no two adjacent flowers in flowerbed.**
- **0 <= n <= flowerbed.length**

```
function canPlaceFlowers (flowerbed, n) {
   for (let i = 0; i < flowerbed.length && n > 0; i++) {
      if (!flowerbed[i] && !flowerbed[i - 1] && !flowerbed[i + 1]) {
         flowerbed[i] = 1;
         n--;
      }
   }
   return n === 0;
};
```

56. Given an array of integers arr, return true *if the number of occurrences of each value in the array is unique or* false *otherwise*.

Example 1:
Input: arr = [1,2,2,1,1,3]
Output: true
Explanation: The value 1 has 3 occurrences, 2 has 2 and 3 has 1. No two values have the same number of occurrences.

Example 2:
Input: arr = [1,2]
Output: false

Example 3:
Input: arr = [-3,0,1,-3,1,1,1,-3,10,0]
Output: true

Constraints:

- **1 <= arr.length <= 1000**
- **-1000 <= arr[i] <= 1000**

```
function uniqueOccurrences (arr) {
   const map = {}
   for (let item of arr)
      map[item] = (map[item] || 0)+1

   return new Set(Object.values(map)).size === Object.values(map).length
};
```

57. Given a 0-indexed n x n integer matrix grid, *return the number of pairs* (r_i, c_j) *such that row* r_i *and column* c_j *are equal.*

A row and column pair is considered equal if they contain the same elements in the same order (i.e., an equal array).

Example 1:
Input: grid = [
[3,2,1],
[1,7,6],
[2,7,7]
]
Output: 1
Explanation: There is 1 equal row and column pair:
- (Row 2, Column 1): [2,7,7]

Example 2:
Input: grid = [
[3,1,2,2],
[1,4,4,5],
[2,4,2,2],
[2,4,2,2]]
Output: 3
Explanation: There are 3 equal row and column pairs:
- (Row 0, Column 0): [3,1,2,2]
- (Row 2, Column 2): [2,4,2,2]
- (Row 3, Column 2): [2,4,2,2]

```
function equalPairs (grid) {
   const rowMap = {}
   let count = 0

   for (let i = 0; i < grid.length; i++) {
      const row = grid[i].join(",")
      rowMap[row] = (rowMap[row] || 0) + 1
   }

   for (let i = 0; i < grid.length; i++) {
      let column = []

      for (let j = 0; j < grid.length; j++)
         column.push(grid[j][i])

      const columnStr = column.join(",");

      if (columnStr in rowMap) {
         count += rowMap[columnStr];
      }
   }
   return count;
};
```

58. Given an array of integers nums, calculate the pivot index of this array.

The pivot index is the index where the sum of all the numbers strictly to the left of the index is equal to the sum of all the numbers strictly to the index's right.
If the index is on the left edge of the array, then the left sum is 0 because there are no elements to the left. This also applies to the right edge of the array.
Return *the leftmost pivot index*. If no such index exists, return -1.
Example 1:
Input: nums = [1,7,3,6,5,6]
Output: 3
Explanation:
The pivot index is 3.
Left sum = nums[0] + nums[1] + nums[2] = 1 + 7 + 3 = 11
Right sum = nums[4] + nums[5] = 5 + 6 = 11

Example 2:
Input: nums = [1,2,3]
Output: -1
Explanation:
There is no index that satisfies the conditions in the problem statement.

Example 3:
Input: nums = [2,1,-1]
Output: 0
Explanation:
The pivot index is 0.
Left sum = 0 (no elements to the left of index 0)
Right sum = nums[1] + nums[2] = 1 + -1 = 0

Constraints:

- **1 <= nums.length <= 10^4**
- **-1000 <= nums[i] <= 1000**

```
function pivotIndex (nums) {
   const totalSum = nums.reduce((sum, num) => sum + num, 0);

   let leftSum = 0;

   for (let i = 0; i < nums.length; i++) {
      // Check if the left sum equals the right sum
      if (leftSum === totalSum - leftSum - nums[i]) {
         return i; // Return the pivot index
      }

      // Update the left sum for the next iteration
      leftSum += nums[i];
   }
    return -1;
};
```

59. There is a biker going on a road trip. The road trip consists of n + 1 points at different altitudes. The biker starts his trip on point 0 with altitude equal 0.

You are given an integer array gain of length n where gain[i] is the net gain in altitude between points i and i + 1 for all (0 <= i < n). Return *the highest altitude of a point.*

Example 1:
Input: gain = [-5,1,5,0,-7]
Output: 1
Explanation: The altitudes are [0,-5,-4,1,1,-6]. The highest is 1.
Example 2:
Input: gain = [-4,-3,-2,-1,4,3,2]
Output: 0
Explanation: The altitudes are [0,-4,-7,-9,-10,-6,-3,-1]. The highest is 0.

```
function largestAltitude(gain) {
    let maxAltitude = 0;
    let currentAltitude = 0;

    for (let i = 0; i < gain.length; i++) {
        currentAltitude += gain[i]; // Update current altitude
        maxAltitude = Math.max(maxAltitude, currentAltitude) // Update max altitude
    }

    return maxAltitude;
};
```

60. Given the array nums, for each nums[i] find out how many numbers in the array are smaller than it. That is, for each nums[i] you have to count the number of valid j's such that j != i and nums[j] < nums[i].

Return the answer in an array.

Example 1:
Input: nums = [8,1,2,2,3]
Output: [4,0,1,1,3]
Explanation:
For nums[0]=8 there exist four smaller numbers than it (1, 2, 2 and 3).
For nums[1]=1 does not exist any smaller number than it.
For nums[2]=2 there exist one smaller number than it (1).
For nums[3]=2 there exist one smaller number than it (1).
For nums[4]=3 there exist three smaller numbers than it (1, 2 and 2).
Example 2:
Input: nums = [6,5,4,8]
Output: [2,1,0,3]
Example 3:
Input: nums = [7,7,7,7]
Output: [0,0,0,0]

Constraints:

- **2 <= nums.length <= 500**
- **0 <= nums[i] <= 100**

```
function smallerNumbersThanCurrent (nums) {
   const result = []

   for (let i = 0; i < nums.length; i++) {
      const smallerNumbers = nums.filter((num) => nums[i] > num)
      result.push(smallerNumbers.length)
   }

   return result;
};
```

61. You are given an array prices where prices[i] is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.
Return *the maximum profit you can achieve from this transaction*. If you cannot achieve any profit, return 0.

Example 1:
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.
Example 2:
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max profit = 0.

Constraints:

- **1 <= prices.length <= 10^5**
- **0 <= prices[i] <= 10^4**

```
function maxProfit (prices) {
    if (prices.length === 0) return 0;

   let minPrice = Infinity; // Initialize to a very high value
   let maxProfit = 0;

   for (let i = 0; i < prices.length; i++) {
      if (prices[i] < minPrice)
         minPrice = prices[i]; // Update the minimum price
      else if (prices[i] - minPrice > maxProfit)
         maxProfit = prices[i] - minPrice; // Update the maximum profit
   }

   return maxProfit;
};
```

62. Given an array nums of n integers where nums[i] is in the range [1, n], return *an array of all the integers in the range* [1, n] *that do not appear in* nums.

Example 1:
Input: nums = [4,3,2,7,8,2,3,1]
Output: [5,6]
Example 2:
Input: nums = [1,1]
Output: [2]

Constraints:

- **n == nums.length**
- **1 <= n <= 10^5**
- **1 <= nums[i] <= n**

```
function findDisappearedNumbers (nums) {
  const seen = new Set(nums); // store all present numbers
  const result = [];

  for (let i = 1; i <= nums.length; i++) {
    if (!seen.has(i)) result.push(i); // if not in set → missing
  }

  return result;
};
```

63. You are given an integer array nums consisting of n elements, and an integer k.

Find a contiguous subarray whose length is equal to k that has the maximum average value and return *this value*. Any answer with a calculation error less than 10^{-5} will be accepted.

Example 1:
Input: nums = [1,12,-5,-6,50,3], k = 4
Output: 12.75000

Explanation: Maximum average is (12 - 5 - 6 + 50) / 4 = 51 / 4 = 12.75
Example 2:
Input: nums = [5], k = 1
Output: 5.00000

Constraints:

- **n == nums.length**
- **1 <= k <= n <= 10^5**
- **-10^4 <= nums[i] <= 10^4**

```
function findMaxAverage (nums, k) {
   let maximum = -Infinity
   for (let i = 0; i <= nums.length - k; i++) {
      let currentAverage = 0
      for (let j = i; j < i + k; j++) {
         currentAverage = currentAverage + nums[j]
      }

      currentAverage = currentAverage / k
      maximum = Math.max(currentAverage, maximum)

   }
   return maximum
};
```

```
function maxAvgSubArray(arr, k) {
  let  currentSum = 0

  for (let i = 0; i < k; i++) currentSum = currentSum + arr[i]

 let  maxAvg = currentSum / k

  for (let i = k; i < arr.length; i++) {
    currentSum = currentSum - arr[i - k] + arr[i]
    maxAvg = Math.max(maxAvg, currentSum / k)
  }
  return maxAvg
}
```

64. Write a function called maxSubarraySum that takes an array of integers and a positive integer k as input. The function should find the maximum sum of any subarray of length k using an O(n) solution by using the sliding window technique.

function maxSubarraySum(arr: number[], k: number): number
Examples
const arr1 = [2, 5, 3, 1, 11, 7, 6, 4];
const k1 = 3;
console.log(maxSubarraySum(arr1, k1)); // Output: 24

const arr2 = [-2, -5, -3, -1, -11, -7, -6, -4];
const k2 = 4;
console.log(maxSubarraySum(arr2, k2)); // Output: -9

Constraints

- **The input integer k will be between 1 and the length of the array.**

```
function maxSubarraySum(arr, k) {
  // Initialize variables to track current sum
  let currentSum = 0;

  // Calculate the sum of the first k elements
  for (let i = 0; i < k; i++) {
    currentSum += arr[i];
  }

  // Set the initial value of maxSum as currentSum
  let maxSum = currentSum;

  // Slide the window and calculate maximum sum
  for (let i = k; i < arr.length; i++) {
    // Update currentSum by subtracting the element that left the window and adding the new element
    currentSum = currentSum - arr[i - k] + arr[i];

    // Update maxSum with the maximum value between maxSum and currentSum
    maxSum = Math.max(maxSum, currentSum);
  }

  // Return the maximum sum
  return maxSum;
}
```

65. Given an integer array `nums`, find the contiguous subarray which has the largest sum and return its sum. This is known as Kadane's Algorithm. You must solve it in O(n) time.

Example Inputs & Outputs

// Example 1:
Input: [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.

// Example 2:
Input: [1]
Output: 1

// Example 3:
Input: [5,4,-1,7,8]
Output: 23

Constraints & Edge Cases

- **Array may contain both positive and negative numbers**
- **Single element array should return that element**
- **If all numbers are negative, return the largest among them**
- **Empty array should return -Infinity**

```
function maxSubArray(nums) {
  if (nums.length === 0) return -Infinity

  let currentSum = nums[0]
  let maxSum = nums[0]

  for (let i = 1; i < nums.length; i++){
    currentSum = Math.max(nums[i], currentSum + nums[i])
    maxSum = Math.max(maxSum,currentSum)
  }
  return maxSum
}
```

Recursion

66. Factorial of a number

```
function factorial(n){
if(n===0 || n===1 ){
return 1
}
return n*factorial(n-1)
}
```

67. Sum of n natural numbers

```
function sum(n){
if(n<=1){
return n
}
return n+sum(n-1)
}
```

68. Finding the nth Fibonacci term

```
function fibonacci(n){
if(n<2){
return n
}
return fibonacci(n-1)+fibonacci(n-2)
}
```

69. Find sequence of Fibonacci upto nth term

```
function fibonacci(n){
  if (n === 1) return [0];
  if (n === 2) return [0, 1];
  const sequence=fibonacci(n-1)
  sequence.push(sequence[sequence.length-1]+sequence[sequence.length-2])
  return sequence;
}
}
```

70. Implement pow(x, n), which calculates x raised to the power n (i.e., x^n).

Example 1:
Input: x = 2.00000, n = 10
Output: 1024.00000

Example 2:
Input: x = 2.10000, n = 3
Output: 9.26100

```
function myPow (x, n) {
   if (n === 0) return 1;
   if (n < 0) return myPow(1/x, n * -1);
   if (n % 2 === 0) return myPow(x * x, n / 2);
   return myPow(x * x, (n - 1) / 2) * x;
}
```

71. Write a function to deep clone an object.

```
function deepClone(obj) {
   if (obj === null || typeof obj !== 'object') {
      return obj
   }

   let clone = Array.isArray(obj) ? [] : {}

   for (let key in obj) {
      if (obj.hasOwnProperty(key)) {
         clone[key] = deepClone(obj[key])
      }
   }
   return clone;
}
```

72. Write a function flattenObject that takes a deeply nested JavaScript object and returns a new object with only one level, where the keys are the path to each value in the original object, separated by dot notation.

Input:
{a: 1, b: {c: 2, d: {e: 3}}}

Output:
{"a": 1,"b.c": 2,"b.d.e": 3}

```
function flattenObject(obj, parentKey ='', result = {}) {
    for (const key in obj) {
        if (obj.hasOwnProperty(key)) {
            const fullKey = parentKey ? `${parentKey}.${key}` : key
            const value = obj[key]

            if (value !== null && typeof value === 'object' && !Array.isArray(value))
              flattenObject(value,fullKey,result)
            else
               result[fullKey] = value
        }
    }
    return result
}
```

73. Design a flat function which flattens an array to infinite depth (Removes all the nesting of arrays)

```
function flat(arr) {
  const flatArr = [];
  for (let value of arr) {
    if (Array.isArray(value))
      flatArr.push(...flat(value));
     else
      flatArr.push(value);
  };
  return flatArr;
}
```

74. Given a multi-dimensional array arr and a depth n, return a flattened version of that array.
A multi-dimensional array is a recursive data structure that contains integers or other multi-dimensional arrays.
A flattened array is a version of that array with some or all of the sub-arrays removed and replaced with the actual elements in that sub-array. This flattening operation should only be done if the current depth of nesting is less than n. The depth of the elements in the first array are considered to be 0.
Please solve it without the built-in Array.flat method.

Example 1:
Input
arr = [1, 2, 3, [4, 5, 6], [7, 8, [9, 10, 11], 12], [13, 14, 15]]
n = 0
Output
[1, 2, 3, [4, 5, 6], [7, 8, [9, 10, 11], 12], [13, 14, 15]]

Explanation
Passing a depth of n=0 will always result in the original array. This is because the smallest possible depth of a subarray (0) is not less than n=0. Thus, no subarray should be flattened.

Example 2:
Input
arr = [1, 2, 3, [4, 5, 6], [7, 8, [9, 10, 11], 12], [13, 14, 15]]
n = 1
Output
[1, 2, 3, 4, 5, 6, 7, 8, [9, 10, 11], 12, 13, 14, 15]

Explanation
The subarrays starting with 4, 7, and 13 are all flattened. This is because their depth of 0 is less than 1. However [9, 10, 11] remains unflattened because its depth is 1.

Example 3:
Input
arr = [[1, 2, 3], [4, 5, 6], [7, 8, [9, 10, 11], 12], [13, 14, 15]]
n = 2
Output
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]

Explanation
The maximum depth of any subarray is 1. Thus, all of them are flattened.

Constraints:
- **0 <= count of numbers in arr <= 10^5**
- **0 <= count of subarrays in arr <= 10^5**
- **maxDepth <= 1000**
- **-1000 <= each number <= 1000**
- **0 <= n <= 1000**

```
function flat (arr, n) {
  // Base case: if depth n is 0, return the array as is.
  if (n === 0) {
    return arr;
  }

  return arr.reduce((result, item) => {
    if (Array.isArray(item)) {
      result.push(...flat(item, n - 1));
    } else {
      result.push(item);
    }
    return result;
  }, []);
};
```

Numbers related

75. Swap two numbers without using temporary variable

```
let num1 = 10, num2 = 20;
num1 = num1 + num2;
num2 = num1 - num2;
num1 = num1 - num2;
console.log(num1,num2)
```

```
let num1 = 10, num2 = 20;
[num1, num2] = [num2, num1];
console.log(num1,num2)
```

76. Create a function which returns a random number in the given range of values both inclusive

```
function randomNumberGeneratorInRange(rangeStart, rangeEnd) {
   return Math.floor(Math.random() * (rangeEnd - rangeStart + 1)) + rangeStart;
}
```

77. Reverse a number

```
function reverse(num) {
  let result = 0;
  while (num != 0) {
    result = result * 10;
    result = result + (num % 10);
    num = Math.floor(num / 10);
  }
  return result;
}
```

```
  const isNegative = num < 0;
  const reversed = +Math.abs(num).toString().split("").reverse().join("");
  return isNegative ? -reversed : reversed;
```

78. Armstrong Number

```
function armstrongNumber(num) {
let sum=0;
let digits=[...num.toString()]

for(let digit of digits){
sum=sum+Math.pow(digit,3)
}
return sum===num
}
```

79. Sum of digits until a single digit is returned

1234=>10=>1

```
function sumOfDigits(num){
let sum=0;
let digits=[...num.toString()]

for(let digit of digits){
sum=sum+Number(digit)
}
return sum;
}

function checkSum(sum){
let summation=sum
while(summation.toString().length>1){
summation=sumOfDigits(summation)
}
return summation;
}
```

80. Prime Number

```
function primeNumber(num) {
  if ([0, 1].includes(num)) return false;

  for (let i = 2; i < num; i++) {
    if (num % i === 0) {
      return false;
    }
  }
  return true;
}
```

81. Generate prime numbers between a range both inclusive

```
function generatePrimeNumbers(a, b) {
  const result = [];
  for (let i = a; i <= b; i++) {
    let num = i;
    let c = 0;
    if ([0, 1].includes(num)) continue;
    for (let j = 2; j < num; j++) {
      if (num % j === 0) {
        c++;
        break;
      }
    }
    if (c === 0) result.push(num);
  }
  return result;
}
```

82. Given an integer n, for every integer i <= n, the task is to print “FizzBuzz” if i is divisible by 3 and 5, “Fizz” if i is divisible by 3, “Buzz” if i is divisible by 5 or i (as a string) if none of the conditions are true.

```
function fizzBuzz(n) {
  let result = [];
  for (let i = 1; i <= n; ++i) {
    if (i % 3 === 0 && i % 5 === 0) {
      result.push("FizzBuzz");
    } else if (i % 3 === 0) {
      result.push("Fizz");
    } else if (i % 5 === 0) {
      result.push("Buzz");
    } else {
      result.push(i.toString());
    }
  }
  return result;
}
```

83. Roman to Integer conversion

```
function romToInt(num) {

   const symbol = {
     'I': 1,
     'V': 5,
     'X': 10,
     'L': 50,
     'C': 100,
     'D': 500,
     'M': 1000
   }
   let result = 0
   for (let i = 0; i < num.length; i++) {
     const curr = symbol[num[i]]
     const next = symbol[num[i + 1]]

     if (curr < next) {
       result += next - curr;
       i++;
     } else result += curr

   }
   return result
 }
```

84. Integer to Roman Conversion

```
function intToRom(num) {
  const list = ['M', 'CM', 'D', 'CD', 'C', 'XC', 'L', 'XL', 'X', 'IX', 'V', 'IV', 'I'];
  const valueList = [1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1];
  let result = '';

  while (num !== 0) {
    for (let i = 0; i < list.length; i++) {
      if (num >= valueList[i]) {
        result += list[i];
        num -= valueList[i];
        break;
      }
    }
  }
  return result;
}
```

85. Write an algorithm to determine if a number n is happy.

A happy number is a number defined by the following process:
Starting with any positive integer, replace the number by the sum of the squares of its digits.
Repeat the process until the number equals 1 (where it will stay), or it loops endlessly in a cycle which does not include 1.
Those numbers for which this process ends in 1 are happy.
Return true if n is a happy number, and false if not.
Example 1:
Input: n = 19
Output: true
Explanation:
12 + 92 = 82
82 + 22 = 68
62 + 82 = 100
12 + 02 + 02 = 1

Example 2:
Input: n = 2
Output: false

```
function isHappy (n) {
   let sum = 0;
   while (n > 0) {
      let e = n % 10;
      n = Math.floor(n / 10);
      sum += e * e;
   }
   if (sum === 1)
      return true;
   else if (sum > 1 && sum <= 4)
      return false;

   return isHappy(sum);
}
```

86. Given an integer n, return *the number of trailing zeroes in* n!.
Note that n! = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1.

Example 1:
Input: n = 3
Output: 0
Explanation: 3! = 6, no trailing zero.

Example 2:
Input: n = 5
Output: 1

Explanation: 5! = 120, one trailing zero.
Example 3:
Input: n = 0
Output: 0

Constraints:

- **0 <= n <= 10^4**

```
function trailingZeroes (n) {
   let result = 0
   while (n >= 5) {
      n = Math.floor(n / 5); // Count multiples of 5, 25, 125, etc.
      result += n; // Accumulate the count
   }
   return result;
};
```

Function Transformations

87. Write a function argumentsLength that returns the count of arguments passed to it.

Example 1:
Input: args = [5]
Output: 1
Explanation:
argumentsLength(5); // 1

One value was passed to the function so it should return 1.
Example 2:
Input: args = [{}, null, "3"]
Output: 3
Explanation:
argumentsLength({}, null, "3"); // 3

Three values were passed to the function so it should return 3.

Constraints:

- **args is a valid JSON array**
- **0 <= args.length <= 100**

```
function argumentsLength (...args) {
    return args.length
};
```

88. Given an array of functions [f1, f_2, f_3, ..., f_n], return a new function fn that is the function composition of the array of functions.

The function composition of [f(x), g(x), h(x)] is fn(x) = f(g(h(x))).
The function composition of an empty list of functions is the identity function f(x) = x.
You may assume each function in the array accepts one integer as input and returns one integer as output.

Example 1:
Input: functions = [x => x + 1, x => x * x, x => 2 * x], x = 4
Output: 65
Explanation:
Evaluating from right to left ...
Starting with x = 4.
2 * (4) = 8
(8) * (8) = 64
(64) + 1 = 65

Example 2:
Input: functions = [x => 10 * x, x => 10 * x, x => 10 * x], x = 1
Output: 1000
Explanation:
Evaluating from right to left ...
10 * (1) = 10
10 * (10) = 100
10 * (100) = 1000

Example 3:
Input: functions = [], x = 42
Output: 42
Explanation:
The composition of zero functions is the identity function

Constraints:

- **-1000 <= x <= 1000**
- **0 <= functions.length <= 1000**
- **all functions accept and return a single integer**

```
function compose (functions) {
  return function (x) {
    return functions.reduceRight((acc, fn) => fn(acc), x);
  }
};
```

89. Given a function fn, return a new function that is identical to the original function except that it ensures fn is called at most once.

- **The first time the returned function is called, it should return the same result as fn.**
- **Every subsequent time it is called, it should return undefined.**

Example 1:
Input: fn = (a,b,c) => (a + b + c), calls = [[1,2,3],[2,3,6]]
Output: [{"calls":1,"value":6}]
Explanation:
const onceFn = once(fn);
onceFn(1, 2, 3); // 6
onceFn(2, 3, 6); // undefined, fn was not called

Example 2:
Input: fn = (a,b,c) => (a * b * c), calls = [[5,7,4],[2,3,6],[4,6,8]]
Output: [{"calls":1,"value":140}]
Explanation:
const onceFn = once(fn);
onceFn(5, 7, 4); // 140
onceFn(2, 3, 6); // undefined, fn was not called
onceFn(4, 6, 8); // undefined, fn was not called

Constraints:

- **calls is a valid JSON array**
- **1 <= calls.length <= 10**
- **1 <= calls[i].length <= 100**
- **2 <= JSON.stringify(calls).length <= 1000**

```
function once (fn) {
   let count = 0

   return function(...args){
      if(!count){
        count++
        return fn(...args)
      }
      return undefined

   }
};
```

90. Given a function fn, return a memoized version of that function.

A memoized function is a function that will never be called twice with the same inputs. Instead it will return a cached value.

You can assume there are 3 possible input functions: sum, fib, and factorial.

- **sum accepts two integers a and b and returns a + b. Assume that if a value has already been cached for the arguments (b, a) where a != b, it cannot be used for the arguments (a, b). For example, if the arguments are (3, 2) and (2, 3), two separate calls should be made.**
- **fib accepts a single integer n and returns 1 if n <= 1 or fib(n - 1) + fib(n - 2) otherwise.**
- **factorial accepts a single integer n and returns 1 if n <= 1 or factorial(n - 1) * n otherwise.**

Example 1:
Input:
fnName = "sum"
actions = ["call","call","getCallCount","call","getCallCount"]
values = [[2,2],[2,2],[],[1,2],[]]
Output: [4,4,1,3,2]
Explanation:
const sum = (a, b) => a + b;
const memoizedSum = memoize(sum);
memoizedSum(2, 2); // "call" - returns 4. sum() was called as (2, 2) was not seen before.
memoizedSum(2, 2); // "call" - returns 4. However sum() was not called because the same inputs were seen before.
// "getCallCount" - total call count: 1
memoizedSum(1, 2); // "call" - returns 3. sum() was called as (1, 2) was not seen before.
// "getCallCount" - total call count: 2

Example 2:
Input:
fnName = "factorial"
actions = ["call","call","call","getCallCount","call","getCallCount"]
values = [[2],[3],[2],[],[3],[]]
Output: [2,6,2,2,6,2]
Explanation:
const factorial = (n) => (n <= 1) ? 1 : (n * factorial(n - 1));
const memoFactorial = memoize(factorial);
memoFactorial(2); // "call" - returns 2.
memoFactorial(3); // "call" - returns 6.
memoFactorial(2); // "call" - returns 2. However factorial was not called because 2 was seen before.
// "getCallCount" - total call count: 2
memoFactorial(3); // "call" - returns 6. However factorial was not called because 3 was seen before.
// "getCallCount" - total call count: 2

Constraints:

- **$0 <= a, b <= 10^5$**
- **$1 <= n <= 10$**
- **$1 <= actions.length <= 10^5$**
- **actions.length === values.length**
- **actions[i] is one of "call" and "getCallCount"**
- **fnName is one of "sum", "factorial" and "fib"**

```
function memoize(fn) {
   const cache = new Map();

   return function(...args) {
      const key = JSON.stringify(args);

      if (cache.has(key)) {
         return cache.get(key);
      }

      const result = fn(...args);
      cache.set(key, result);
      return result;
   };
}
```

91. Write a function that explains function currying

const curriedSum = curry(sum);
console.log(curriedSum(1)(2)(3)); // 6
console.log(curriedSum(1, 2)(3)); // 6
console.log(curriedSum(1)(2, 3)); // 6

```
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn(...args);
    } else {
      return function (...nextArgs) {
        return curried(...args, ...nextArgs);
      }
    }
  };
}
```

92. Write a function that explains infinite function currying

```
function sum(a) {
  return function(b) {
    if (b !== undefined)  return sum(a + b);
    return a;
  }
}
```

93. Write a function to satisfy the below statements
sum(2, 3)
sum(2)(3)

```
function sum(num1, num2) {
  if (num2 === undefined)
  return (num3) => num1 + num3
  return num1 + num2
}
```

String Related

94. String Palindrome

```
function palindrome(word){
let reverse=word.split("").reverse().join("");
return reverse===word
}
```

```
function palindrome(word){
let reverse=[...word].reverse().join("");
return reverse===word
}
```

```
function palindrome(word) {
  let reverse = ""
  for (let i = word.length - 1; i >= 0; i--) {
    reverse = reverse + word[i]
  }
  return reverse === word
}
```

```
function validatePalindrome(str) {
    const cleaned = str.replace(/[^a-z0-9]/ig, '').toLowerCase();
    return cleaned === cleaned.split('').reverse().join('');
}
```

95. Reversing a sentence with reversing order and letter of individual words.

```
function reverseSentence(sentence) {
  let reverse = sentence.split("").reverse().join("")
  return reverse
}
```

```
function reverseSentence(sentence) {
  let reverse = ""
  for(let i=sentence.length-1;i>=0;i--){
  reverse+=sentence[i];
  }
  return reverse
}
```

96. Reversing a sentence without reversing the order and with reversing each letter of individual words

```
function reverseSentence(sentence) {
  let reverse = sentence.split('').reverse().join('')
    .split(' ').reverse().join(' ')
  return reverse;
}
```

```
function reverseSentence(sentence) {
  return sentence
    .split(' ')
    .map(word => word.split('').reverse().join(''))
    .join(' ');
}
```

97. String anagram

```
function anagram(word1,word2) {
  let wr1 = word1.toLowerCase().split('').sort().join('')
  let wr2 = word2.toLowerCase().split('').sort().join('')
  return wr1===wr2
}
```

```
function anagram(str1, str2) {
  const format = str =>
    str.replace(/[^a-z]/gi, '').toLowerCase().split('').sort().join('');

  return format(str1) === format(str2);
}
```

98. Given an array of strings strs, group the anagrams together. You can return the answer in any order.

Input: strs = ["eat","tea","tan","ate","nat","bat"]
Output: [["bat"],["nat","tan"],["ate","eat","tea"]]

```
function groupAnagrams(strs) {
   let result = {};
   for (let word of strs) {
     let cleansed = word.split('').sort().join('');
     if (result[cleansed])
       result[cleansed].push(word);
     else
       result[cleansed] = [word];
   }
   return Object.values(result);
 };
```

99. Remove duplicate values (keep the ocurrence) from a string

```
function removeDuplicates(word) {
  let result = [...new Set(word.split(""))].join("")
  return result;
}
```

```
function removeDuplicates(word) {
 return [...word].filter((item,index,arr)=>arr.indexOf(item)===index).join("")
}
```

100. Remove duplicate values (do not keep any occurrence) from a string

```
function removeDuplicates(word) {
  return word
    .split("")
    .filter((letter) => word.indexOf(letter) === word.lastIndexOf(letter))
    .join("");
}
```

101. Remove all vowels from a string

```
const str = "I love JavaScript";
str.replace(/[aeiou]/gi, '');
```

```
const str = "I love JavaScript";
const vowels = ["a", "i", "e", "o", "u"]
let newStr = "";
for (let word of str) {
  if (vowels.includes(word.toLowerCase())) continue;
  newStr += word
}
```

```
const str = "I love JavaScript";
const vowels = ["a", "i", "e", "o", "u"]
let newStr = [...str].filter((letter) =>
!vowels.includes(letter.toLowerCase())).join("")
```

102. Count number of occurrences of elements in string and find the element which has maximum occurrences if two elements have some number of occurrences return the first occurring element in the string

```
function maxOccurrences(str) {
  let occurrences = {};
  let max = 0;

  for (let char of str) {
    occurrences[char] = (occurrences[char] || 0) + 1;
    if (occurrences[char] > max)
      max = occurrences[char];
  }

  for (let char of str) {
    if (occurrences[char] === max)
      return char;
  }
}
```

103. Count number of occurrences of elements in the string and sort the array in the descending order according to occurrences.

```
function maxOccurrencesSortUnique(str) {
  let occurrences = {};

  for (let char of str) {
    occurrences[char] = (occurrences[char] || 0) + 1;
  }

  return Object.keys(occurrences)
    .sort((a, b) => occurrences[b] - occurrences[a])
    .join('');
}
```

104. Find longest word in the sentence

```
function longestWord(sentence) {
  const str = sentence.split(" ")
  let longest = "";
  for (let word of str) {
    if (word.length > longest.length) {
      longest = word;
    }
  }
  return longest
}
```

105. Given two strings s and t, return true if s is a subsequence of t, or false otherwise.

A subsequence of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).

```
function isSubsequence(s, t) {
  if (s.length === 0) return true

  let sPointer = 0

  for (let i = 0; i < t.length; i++) {
    if (s[sPointer] === t[i]) sPointer++
  }
  return sPointer === s.length
}
```

106. Given two strings ransomNote and magazine, return true if ransomNote can be constructed by using the letters from magazine and false otherwise.
Each letter in magazine can only be used once in ransomNote.

Example 1:

Input: ransomNote = "a", magazine = "b"
Output: false
Example 2:

Input: ransomNote = "aa", magazine = "ab"
Output: false
Example 3:

Input: ransomNote = "aa", magazine = "aab"
Output: true

```
function canConstruct(ransomNote, magazine) {
  for (let i = 0; i < ransomNote.length; i++) {
      if (magazine.indexOf(ransomNote[i]) === -1) {
        return false
      } else {
        magazine = magazine.replace(ransomNote[i], '')
      }
   }
  return true
};
```

107. Given two strings s and t, determine if they are isomorphic.
Two strings s and t are isomorphic if the characters in s can be replaced to get t.
All occurrences of a character must be replaced with another character while preserving the order of characters. No two characters may map to the same character, but a character may map to itself.

Example 1:

Input: s = "egg", t = "add"
Output: true
Example 2:

Input: s = "foo", t = "bar"
Output: false
Example 3:

Input: s = "paper", t = "title"
Output: true

```
function isIsomorphic(s, t) {
    if(s.length !== t.length) return false;

   for(let i = 0; i < s.length; i++) {
      if(s.indexOf(s[i]) !== t.indexOf(t[i]))
         return false;
   }
   return true;
};
```

108. Given a pattern and a string s, find if s follows the same pattern.
Here follow means a full match, such that there is a bijection between a letter in pattern and a non-empty word in s.

Example 1:
Input: pattern = "abba", s = "dog cat cat dog"
Output: true

Example 2:
Input: pattern = "abba", s = "dog cat cat fish"
Output: false

```
function wordPattern(pattern, s) {
  const sArr = s.split(' ');
  if (pattern.length !== sArr.length) {
    return false
  }
  const patternMap = {};
  const sMap = {};

  for (let i = 0; i < pattern.length; i++) {
    patternMap[pattern[i]] = i;
    sMap[sArr[i]] = i
  }
  for (let i = 0; i < pattern.length; i++) {
    if (patternMap[pattern[i]] !== sMap[sArr[i]])
      return false
  }
  return true
}
```

109. Write a function to find the longest common prefix string amongst an array of strings. If there is no common prefix, return an empty string "".

Example 1:

Input: strs = ["flower","flow","flight"]
Output: "fl"

Example 2:
Input: strs = ["dog","racecar","car"]
Output: ""

```
function longestCommonPrefix(strs) {
   let prefix = ""
   if (strs === null || strs.length === 0) return prefix

   for (let i = 0; i < strs[0].length; i++) {
      const char = strs[0][i]

      for (let j = 1; j < strs.length; j++) {
         if (strs[j][i] !== char) return prefix
      }
      prefix = prefix + char
   }

   return prefix
};
```

110. Write a function to add space before every capital letter except at the start and end

```
function addSpace(sentence) {
  return sentence.replace(/([A-Z])/g, " $1").trim()
}
```

111. Write a function to produce combination of two strings as per the following pattern:

Example 1:
word1= “abcdef”
word2 = “123”
output: “a1b2c3def”

Example 2:
word1= “123456”
word2 = “abc”
output: “a1b2c3456”

```
function combo(word1, word2) {
  let result = ""
  let length = Math.min(word1.length, word2.length)

  for (let i = 0; i < length; i++) {
    result = result+ word1[i] + word2[i]
  }

  result =result +word1.slice(length)+word2.slice(length)
  return result
}
```

112. Write a program to get below output from given input ?
I/P: abbcccddddeea, O/P: 1a2b3c4d2e1a

```
function encodeString(word) {
  let count = 1
  let result = ""

  for (let i = 1; i < word.length; i++) {
    if (word[i] === word[i - 1]) count++
    else {
      result = result + count + word[i - 1]
      count = 1
    }
  }

  result = result + count + word[word.length - 1]
  return result
}
```

113. Given a string s, find the length of the longest substring without repeating characters.

Example 1:
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.

Example 2:
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.

Example 3:
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

```
function lengthOfLongestSubstring(s) {
   let start = 0, max=0, map={};
    for (let i = 0; i < s.length; i++) {
      const currentChar = s[i];
   // If the current character has been seen before and is within the current window
       if (map[currentChar] !== undefined) {
          // Move the start to the right of the previous occurrence to ensure no duplicates
          // Use Math.max to avoid moving start backward (e.g., if the previous index is outside the
current window)
          start = Math.max(start, map[currentChar] + 1);
       }
       // Update the last seen index of the current character
       map[currentChar] = i;
       // Calculate the length of the current window and update max if necessary
       max = Math.max(max, i - start + 1);
   }

   return max;

};
```

114. Write a function called titleCase that takes in a string and returns the string with the first letter of each word capitalized.

Examples:
titleCase("I'm a little tea pot"); // I'm A Little Tea Pot
titleCase('sHoRt AnD sToUt'); // Short And Stout

Constraints
You may assume that each word consists of only letters and spaces

```
function titleCase(str) {
  const words = str.toLowerCase().split(' ');

  for (let i = 0; i < words.length; i++) {
    words[i] = words[i][0].toUpperCase() + words[i].slice(1);
  }
  return words.join(' ');
}
```

115. Write a function called validateEmail that takes in a string and returns whether the string is a valid email address. For the purposes of this challenge, a valid email address is defined as a string that contains an @ symbol and a . symbol.

Examples
validateEmail('john@gmail.com'); // true
validateEmail('john@gmail'); // false

```
function validateEmail(email) {
  const emailRegex = /^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$/;
  return emailRegex.test(email);
}
```

116. Write a function called isValidIPv4 that takes a string as input and returns true if the input is a valid IPv4 address in dot-decimal format, and false otherwise. An IPv4 address should consist of four octets, with values between 0 and 255, inclusive.

Examples
isValidIPv4('1.2.3.4'); // true
isValidIPv4('123.45.67.89'); // true
isValidIPv4('1.2.3'); // false
isValidIPv4('1.2.3.4.5'); // false
isValidIPv4('123.456.78.90'); // false
isValidIPv4('123.045.067.089'); // false

Constraints

- **The input will be a single string.**

```
const isValidIPv4 = (input) => {
  const octets = input.split('.')

  if (octets.length !== 4)
    return false;

  return octets.every((octet) => {
    const num = parseInt(octet);
    return num >= 0 && num <= 255 && octet === num.toString();
  });
};
```

117. Write a function called validatePassword that takes in a string and validates it based on the following criteria:

- **The password must be at least 8 characters long.**
- **The password must contain at least one uppercase letter.**
- **The password must contain at least one lowercase letter.**
- **The password must contain at least one digit.**

Examples
validatePassword('Abc12345'); // should return true
validatePassword('password123'); // should return false (no uppercase letter)
validatePassword('Pass'); // should return false (length less than 8 characters)
validatePassword('HelloWorld'); // should return false (no digit)

Constraints

- **The input password can contain any combination of letters and digits.**
- **The input password can contain both uppercase and lowercase letters.**

```
function validatePassword(password) {

  const isLengthValid = password.length >= 8

  const hasUppercase = password
    .split('')
    .some((char) => char === char.toUpperCase() && char !== char.toLowerCase());

  const hasLowercase = password
    .split('')
    .some((char) => char === char.toLowerCase() && char !== char.toUpperCase());

  const hasDigit = password
    .split('')
    .some((char) => !isNaN(parseInt(char, 10)));

  return isLengthValid && hasUppercase && hasLowercase && hasDigit;
}
```

118. You are given two strings word1 and word2. Merge the strings by adding letters in alternating order, starting with word1. If a string is longer than the other, append the additional letters onto the end of the merged string.

Return *the merged string.*

Example 1:
Input: word1 = "abc", word2 = "pqr"
Output: "apbqcr"
Explanation: The merged string will be merged as so:
word1: a b c
word2: p q r
merged: apbqcr

Example 2:
Input: word1 = "ab", word2 = "pqrs"
Output: "apbqrs"
Explanation: Notice that as word2 is longer, "rs" is appended to the end.
word1: a b
word2: p q r s
merged: apbqrs

Example 3:
Input: word1 = "abcd", word2 = "pq"
Output: "apbqcd"
Explanation: Notice that as word1 is longer, "cd" is appended to the end.
word1: a b c d
word2: p q
merged: apbqcd

Constraints:

- **1 <= word1.length, word2.length <= 100**
- **word1 and word2 consist of lowercase English letters.**

```
function mergeAlternately (word1, word2) {
   const length = Math.min(word1.length,word2.length)
   let result = ''

   for (let i = 0; i < length; i++)
     result = result + word1[i] + word2[i]

   result =result +word1.slice(length)+word2.slice(length)

   return result
};
```

119. Given an input string s, reverse the order of the words.
A word is defined as a sequence of non-space characters. The words in s will be separated by at least one space.
Return *a string of the words in reverse order concatenated by a single space.*
Note that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.

Example 1:
Input: s = "the sky is blue"
Output: "blue is sky the"
Example 2:
Input: s = " hello world "
Output: "world hello"
Explanation: Your reversed string should not contain leading or trailing spaces.
Example 3:
Input: s = "a good   example"
Output: "example good a"
Explanation: You need to reduce multiple spaces between two words to a single space in the reversed string.

Constraints:

- **$1 <= s.length <= 10^4$**
- **s contains English letters (upper-case and lower-case), digits, and spaces ' '.**
- **There is at least one word in s.**

```
function reverseWords (s) {
    return s.trim().replace(/\s+/g," ").split(" ").reverse().join(" ");
};
```

120. Given a string s, reverse only all the vowels in the string and return it.
The vowels are 'a', 'e', 'i', 'o', and 'u', and they can appear in both lower and upper cases, more than once.

Example 1:
Input: s = "IceCreAm"
Output: "AceCreIm"
Explanation:
The vowels in s are ['I', 'e', 'e', 'A']. On reversing the vowels, s becomes "AceCreIm".

Example 2:
Input: s = "leetcode"
Output: "leotcede"

Constraints:

- **1 <= s.length <= 3 * 10^5**
- **s consist of printable ASCII characters.**

```
function reverseVowels (s) {
  let result = s.split("")
  const indexes = []
  const values = []

  for (let i = 0; i < result.length; i++) {
   if((/[aieou]/i).test(s[i])) {
      indexes.push(i)
      values.push(s[i])
    }
  }

  const valuesReversed = values.reverse()

  for (let i = 0; i < valuesReversed.length; i++) {
    result[indexes[i]] = valuesReversed[i]
  }

  return result.join("")
};
```

121. Two strings are considered close if you can attain one from the other using the following operations:

- **Operation 1: Swap any two existing characters.**
  - **For example, abcde -> aecdb**
- **Operation 2: Transform every occurrence of one existing character into another existing character, and do the same with the other character.**
  - **For example, aacabb -> bbcbaa (all a's turn into b's, and all b's turn into a's)**

You can use the operations on either string as many times as necessary.
Given two strings, word1 and word2, return true *if* word1 *and* word2 *are close, and* false *otherwise.*

Example 1:
Input: word1 = "abc", word2 = "bca"
Output: true
Explanation: You can attain word2 from word1 in 2 operations.
Apply Operation 1: "abc" -> "acb"
Apply Operation 1: "acb" -> "bca"
Example 2:
Input: word1 = "a", word2 = "aa"
Output: false
Explanation: It is impossible to attain word2 from word1, or vice versa, in any number of operations.
Example 3:
Input: word1 = "cabbba", word2 = "abbccc"
Output: true
Explanation: You can attain word2 from word1 in 3 operations.
Apply Operation 1: "cabbba" -> "caabbb"
Apply Operation 2: "caabbb" -> "baaccc"
Apply Operation 2: "baaccc" -> "abbccc"

Constraints:

- **1 <= word1.length, word2.length <= 10^5**
- **word1 and word2 contain only lowercase English letters**

```
function closeStrings (word1, word2) {
   if (word1.length !== word2.length) return false;

   const freq1 = {};
   const freq2 = {};

   for (let char of word1) {
      freq1[char] = (freq1[char] || 0) + 1;
   }

   for (let char of word2) {
      freq2[char] = (freq2[char] || 0) + 1;
   }

   const chars1 = Object.keys(freq1).sort().toString();
   const chars2 = Object.keys(freq2).sort().toString();

   if (chars1 !== chars2) return false;
```

```
    const counts1 = Object.values(freq1).sort().toString();
    const counts2 = Object.values(freq2).sort().toString();

    return counts1 === counts2;
};
```

122. Given a string s and an integer k, return the maximum number of vowel letters in any substring of s with length k.

Vowel letters in English are 'a', 'e', 'i', 'o', and 'u'.

Example 1:
Input: s = "abciiidef", k = 3
Output: 3
Explanation: The substring "iii" contains 3 vowel letters.

Example 2:
Input: s = "aeiou", k = 2
Output: 2
Explanation: Any substring of length 2 contains 2 vowels.

Example 3:
Input: s = "leetcode", k = 3
Output: 2
Explanation: "lee", "eet" and "ode" contain 2 vowels.

Constraints:

- **1 <= s.length <= 10^5**
- **s consists of lowercase English letters.**
- **1 <= k <= s.length**

```
function maxVowels (s, k) {
    let maximum = 0
    const isVowel = word => (/[aeiou]/i).test(word)

    for (let i = 0; i < k; i++) {
        if (isVowel(s[i]))
            maximum++
    }

   let currentSum = maximum;

    for (let i = k; i < s.length; i++) {
        currentSum = currentSum
         - (isVowel(s[i - k]) ? 1 : 0) + (isVowel(s[i]) ? 1 : 0)
        maximum = Math.max(currentSum, maximum)
    }
    return maximum
};
```

123. Given two strings str1 and str2, return *the largest string* x *such that* x *divides both* str1 *and* str2.

Example 1:
Input: str1 = "ABCABC", str2 = "ABC"
Output: "ABC"

Example 2:
Input: str1 = "ABABAB", str2 = "ABAB"
Output: "AB"

Example 3:
Input: str1 = "LEET", str2 = "CODE"
Output: ""

Constraints:

- **1 <= str1.length, str2.length <= 1000**
- **str1 and str2 consist of English uppercase letters.**

```
function gcdOfStrings (str1, str2) {
   if (str1 + str2 !== str2 + str1) {
      return "";
   }

   let a = str1.length;
   let b = str2.length;
   while (b !== 0) {
      const temp = b;
      b = a % b;
      a = temp;
   }
   return str1.slice(0, a);
};
```

Date Related

124. Write a program to count the number of days between two dates.

The two dates are given as strings, their format is YYYY-MM-DD as shown in the examples.

Example 1:
Input: date1 = "2019-06-29", date2 = "2019-06-30"
Output: 1

Example 2:
Input: date1 = "2020-01-15", date2 = "2019-12-31"
Output: 15

Constraints:
- **The given dates are valid dates between the years 1971 and 2100.**

```
function daysBetween(date1, date2) {
  const DAY_IN_MILLISECONDS = 1000 * 60 * 60 * 24;

  const formattedDate1 = new Date(date1);
  const formattedDate2 = new Date(date2);

  if (isNaN(formattedDate1) || isNaN(formattedDate2)) {
    throw new Error("Invalid date format");
  }

  const milliseconds = Math.abs(formattedDate2 - formattedDate1);
  const days = Math.floor(milliseconds / DAY_IN_MILLISECONDS);

  return days;
}
```

125. Given a date string in the form Day Month Year, where:

- **Day is in the set {"1st", "2nd", "3rd", "4th", ..., "30th", "31st"}.**
- **Month is in the set {"Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"}.**
- **Year is in the range [1900, 2100].**
  Convert the date string to the format YYYY-MM-DD, where:
- **YYYY denotes the 4 digit year.**
- **MM denotes the 2 digit month.**
- **DD denotes the 2 digit day.**

Example 1:
Input: date = "20th Oct 2052"
Output: "2052-10-20"

Example 2:
Input: date = "6th Jun 1933"
Output: "1933-06-06"

Example 3:
Input: date = "26th May 1960"
Output: "1960-05-26"

Constraints:

- **The given dates are guaranteed to be valid, so no error handling is necessary.**

```
function reformatDate (date) {
    const months = {
        Jan: '01', Feb: '02', Mar: '03', Apr: '04', May: '05', Jun: '06',
        Jul: '07', Aug: '08', Sep: '09', Oct: '10', Nov: '11', Dec: '12'
    };
    const partsOfDate = date.split(" ");

    const day = parseInt(partsOfDate[0]).toString().padStart(2, '0')
    const month = months[partsOfDate[1]]
    const year = partsOfDate[2]

    return `${year}-${month}-${day}`
};
```

Promises and Time

126. Given two promises promise1 and promise2, return a new promise. promise1 and promise2 will both resolve with a number. The returned promise should resolve with the sum of the two numbers.

Example 1:
Input:
promise1 = new Promise(resolve => setTimeout(() => resolve(2), 20)),
promise2 = new Promise(resolve => setTimeout(() => resolve(5), 60))
Output: 7
Explanation: The two input promises resolve with the values of 2 and 5 respectively. The returned promise should resolve with a value of 2 + 5 = 7. The time the returned promise resolves is not judged for this problem.
Example 2:
Input:
promise1 = new Promise(resolve => setTimeout(() => resolve(10), 50)),
promise2 = new Promise(resolve => setTimeout(() => resolve(-12), 30))
Output: -2
Explanation: The two input promises resolve with the values of 10 and -12 respectively. The returned promise should resolve with a value of 10 + -12 = -2.

Constraints:

- **promise1 and promise2 are promises that resolve with a number**

```
async function addTwoPromises(promise1, promise2) {
    const n1 = await promise1
    const n2 = await promise2
    return n1+n2
};
```

127. Given a positive integer millis, write an asynchronous function that sleeps for millis milliseconds. It can resolve any value.

Example 1:
Input: millis = 100
Output: 100
Explanation: It should return a promise that resolves after 100ms.

```
let t = Date.now();
sleep(100).then(() => {
  console.log(Date.now() - t); // 100
});
```

Example 2:
Input: millis = 200
Output: 200
Explanation: It should return a promise that resolves after 200ms.

Constraints:

- **1 <= millis <= 1000**

```
function sleep(millis) {
  return new Promise(resolve =>setTimeout(resolve, millis))
}
```

128. Given an asynchronous function fn and a time t in milliseconds, return a new time limited version of the input function. fn takes arguments provided to the time limited function.

The time limited function should follow these rules:

- **If the fn completes within the time limit of t milliseconds, the time limited function should resolve with the result.**
- **If the execution of the fn exceeds the time limit, the time limited function should reject with the string "Time Limit Exceeded".**

Example 1:
Input:

```
fn = async (n) => {
  await new Promise(res => setTimeout(res, 100));
  return n * n;
}
inputs = [5]
t = 50
```

Output: {"rejected":"Time Limit Exceeded","time":50}
Explanation:

```
const limited = timeLimit(fn, t)
const start = performance.now()
let result;
try {
   const res = await limited(...inputs)
   result = {"resolved": res, "time": Math.floor(performance.now() - start)};
} catch (err) {
   result = {"rejected": err, "time": Math.floor(performance.now() - start)};
}
console.log(result) // Output
```

The provided function is set to resolve after 100ms. However, the time limit is set to 50ms. It rejects at t=50ms because the time limit was reached.

Example 2:
Input:

```
fn = async (n) => {
  await new Promise(res => setTimeout(res, 100));
  return n * n;
}
inputs = [5]
t = 150
```

Output: {"resolved":25,"time":100}
Explanation:
The function resolved 5 * 5 = 25 at t=100ms. The time limit is never reached.

Example 3:
Input:

```
fn = async (a, b) => {
  await new Promise(res => setTimeout(res, 120));
  return a + b;
}
inputs = [5,10]
```

t = 150
Output: {"resolved":15,"time":120}
Explanation:
The function resolved 5 + 10 = 15 at t=120ms. The time limit is never reached.

Example 4:
Input:
fn = async () => {
throw "Error";
}
inputs = []
t = 1000
Output: {"rejected":"Error","time":0}
Explanation:
The function immediately throws an error.

```
function timeLimit (fn, t) {
   return async function(...args) {
      const timeoutPromise = new Promise((_, reject) =>
         setTimeout(() => reject("Time Limit Exceeded"), t)
      );

      // Race between the function's promise and the timeout
      return Promise.race([fn(...args), timeoutPromise]);
   };
};
```

129. Given an array of asynchronous functions functions, return a new promise promise. Each function in the array accepts no arguments and returns a promise. All the promises should be executed in parallel.

promise resolves:

- **When all the promises returned from functions were resolved successfully in parallel. The resolved value of promise should be an array of all the resolved values of promises in the same order as they were in the functions. The promise should resolve when all the asynchronous functions in the array have completed execution in parallel.**

promise rejects:

- **When any of the promises returned from functions were rejected. promise should also reject with the reason of the first rejection.**

Please solve it without using the built-in Promise.all function.

Example 1:
Input: functions = [
() => new Promise(resolve => setTimeout(() => resolve(5), 200))
]
Output: {"t": 200, "resolved": [5]}
Explanation:
promiseAll(functions).then(console.log); // [5]

The single function was resolved at 200ms with a value of 5.

Example 2:
Input: functions = [
() => new Promise(resolve => setTimeout(() => resolve(1), 200)),
() => new Promise((resolve, reject) => setTimeout(() => reject("Error"), 100))
]
Output: {"t": 100, "rejected": "Error"}
Explanation: Since one of the promises rejected, the returned promise also rejected with the same error at the same time.

Example 3:
Input: functions = [
() => new Promise(resolve => setTimeout(() => resolve(4), 50)),
() => new Promise(resolve => setTimeout(() => resolve(10), 150)),
() => new Promise(resolve => setTimeout(() => resolve(16), 100))
]
Output: {"t": 150, "resolved": [4, 10, 16]}

Explanation: All the promises resolved with a value. The returned promise resolved when the last promise resolved.

Constraints:

- **functions is an array of functions that returns promises**
- **1 <= functions.length <= 10**

```
  function promiseAll(functions) {
   return new Promise((resolve, reject) => {
      const results = [];
      let completed = 0;

      functions.forEach((fn, index) => {
         fn().then(result => {
            results[index] = result;
            completed++;

            if (completed === functions.length) {
               resolve(results);
            }
         }).catch(error => {
            reject(error);
         });
      });
   });
};
```

130. Given a function fn, an array of arguments args, and a timeout t in milliseconds, return a cancel function cancelFn.

After a delay of cancelTimeMs, the returned cancel function cancelFn will be invoked.
setTimeout(cancelFn, cancelTimeMs)
Initially, the execution of the function fn should be delayed by t milliseconds.
If, before the delay of t milliseconds, the function cancelFn is invoked, it should cancel the delayed execution of fn. Otherwise, if cancelFn is not invoked within the specified delay t, fn should be executed with the provided args as arguments.

Example 1:
Input: fn = (x) => x * 5, args = [2], t = 20
Output: [{"time": 20, "returned": 10}]
Explanation:
const cancelTimeMs = 50;
const cancelFn = cancellable((x) => x * 5, [2], 20);
setTimeout(cancelFn, cancelTimeMs);

The cancellation was scheduled to occur after a delay of cancelTimeMs (50ms), which happened after the execution of fn(2) at 20ms.

Example 2:
Input: fn = (x) => x2, args = [2], t = 100**
Output: []
Explanation:
const cancelTimeMs = 50;
const cancelFn = cancellable((x) => x2, [2], 100);**
setTimeout(cancelFn, cancelTimeMs);

The cancellation was scheduled to occur after a delay of cancelTimeMs (50ms), which happened before the execution of fn(2) at 100ms, resulting in fn(2) never being called.

Example 3:
Input: fn = (x1, x2) => x1 * x2, args = [2,4], t = 30
Output: [{"time": 30, "returned": 8}]
Explanation:
const cancelTimeMs = 100;
const cancelFn = cancellable((x1, x2) => x1 * x2, [2,4], 30);
setTimeout(cancelFn, cancelTimeMs);

The cancellation was scheduled to occur after a delay of cancelTimeMs (100ms), which happened after the execution of fn(2,4) at 30ms.

Constraints:

- **fn is a function**
- **args is a valid JSON array**
- **1 <= args.length <= 10**
- **20 <= t <= 1000**
- **10 <= cancelTimeMs <= 1000**

```
function cancellable(fn, args, t) {
  let isCancelled = false;

    // Schedule the execution of fn after t milliseconds
    const timeoutId = setTimeout(() => {
        if (!isCancelled) {
            fn(...args);
        }
    }, t);

    // Return the cancel function
    const cancelFn = () => {
        isCancelled = true;
        clearTimeout(timeoutId);
    };

    return cancelFn;
};
```

131. Given a function fn, an array of arguments args, and an interval time t, return a cancel function cancelFn.

After a delay of cancelTimeMs, the returned cancel function cancelFn will be invoked.
setTimeout(cancelFn, cancelTimeMs)
The function fn should be called with args immediately and then called again every t milliseconds until cancelFn is called at cancelTimeMs ms.

Example 1:
Input: fn = (x) => x * 2, args = [4], t = 35
Output:
[
{"time": 0, "returned": 8},
{"time": 35, "returned": 8},
{"time": 70, "returned": 8},
{"time": 105, "returned": 8},
{"time": 140, "returned": 8},
{"time": 175, "returned": 8}
]
Explanation:
const cancelTimeMs = 190;
const cancelFn = cancellable((x) => x * 2, [4], 35);
setTimeout(cancelFn, cancelTimeMs);

Every 35ms, fn(4) is called. Until t=190ms, then it is cancelled.
1st fn call is at 0ms. fn(4) returns 8.
2nd fn call is at 35ms. fn(4) returns 8.
3rd fn call is at 70ms. fn(4) returns 8.
4th fn call is at 105ms. fn(4) returns 8.
5th fn call is at 140ms. fn(4) returns 8.
6th fn call is at 175ms. fn(4) returns 8.
Cancelled at 190ms
Example 2:
Input: fn = (x1, x2) => (x1 * x2), args = [2, 5], t = 30
Output:
[
{"time": 0, "returned": 10},
{"time": 30, "returned": 10},
{"time": 60, "returned": 10},
{"time": 90, "returned": 10},
{"time": 120, "returned": 10},
{"time": 150, "returned": 10}
]
Explanation:
const cancelTimeMs = 165;
const cancelFn = cancellable((x1, x2) => (x1 * x2), [2, 5], 30)
setTimeout(cancelFn, cancelTimeMs)
Every 30ms, fn(2, 5) is called. Until t=165ms, then it is cancelled.
1st fn call is at 0ms
2nd fn call is at 30ms
3rd fn call is at 60ms
4th fn call is at 90ms
5th fn call is at 120ms
6th fn call is at 150ms

Cancelled at 165ms

Example 3:
Input: fn = (x1, x2, x3) => (x1 + x2 + x3), args = [5, 1, 3], t = 50
Output:
[
{"time": 0, "returned": 9},
{"time": 50, "returned": 9},
{"time": 100, "returned": 9},
{"time": 150, "returned": 9}
]
Explanation:
const cancelTimeMs = 180;
const cancelFn = cancellable((x1, x2, x3) => (x1 + x2 + x3), [5, 1, 3], 50)
setTimeout(cancelFn, cancelTimeMs)
Every 50ms, fn(5, 1, 3) is called. Until t=180ms, then it is cancelled.
1st fn call is at 0ms
2nd fn call is at 50ms
3rd fn call is at 100ms
4th fn call is at 150ms
Cancelled at 180ms
Constraints:

- **fn is a function**
- **args is a valid JSON array**
- **1 <= args.length <= 10**
- **30 <= t <= 100**
- **10 <= cancelTimeMs <= 500**

```
function cancellable(fn, args, t) {
    const results = []; // Array to store results
    let startTime = Date.now(); // Track the start time

    // Function to call fn, record the time and result
    const executeFn = () => {
        const currentTime = Date.now() - startTime; // Calculate elapsed time
        const result = fn(...args); // Call fn with args
        results.push({ time: currentTime, returned: result }); // Store result
    };

    // Call fn immediately
    executeFn();

    // Set up the interval to call fn every t milliseconds
    const intervalId = setInterval(executeFn, t);

    // Return the cancel function
    const cancelFn = () => {
        clearInterval(intervalId); // Stop the interval
    };

    return cancelFn;

};
```

132. Given a function fn and a time in milliseconds t, return a debounced version of that function.

A debounced function is a function whose execution is delayed by t milliseconds and whose execution is cancelled if it is called again within that window of time. The debounced function should also receive the passed parameters.
For example, let's say t = 50ms, and the function was called at 30ms, 60ms, and 100ms. The first 2 function calls would be cancelled, and the 3rd function call would be executed at 150ms.
If instead t = 35ms, The 1st call would be cancelled, the 2nd would be executed at 95ms, and the 3rd would be executed at 135ms.

Example 1:
Input:
t = 50
calls = [
{"t": 50, inputs: [1]},
{"t": 75, inputs: [2]}
]
Output: [{"t": 125, inputs: [2]}]
Explanation:
```
let start = Date.now();
function log(...inputs) {
  console.log([Date.now() - start, inputs ])
}
const dlog = debounce(log, 50);
setTimeout(() => dlog(1), 50);
setTimeout(() => dlog(2), 75);
```

The 1st call is cancelled by the 2nd call because the 2nd call occurred before 100ms
The 2nd call is delayed by 50ms and executed at 125ms. The inputs were (2).

Example 2:
Input:
t = 20
calls = [
{"t": 50, inputs: [1]},
{"t": 100, inputs: [2]}
]
Output: [{"t": 70, inputs: [1]}, {"t": 120, inputs: [2]}]
Explanation:
The 1st call is delayed until 70ms. The inputs were (1).
The 2nd call is delayed until 120ms. The inputs were (2).

Example 3:
Input:
t = 150
calls = [
{"t": 50, inputs: [1, 2]},
{"t": 300, inputs: [3, 4]},
{"t": 300, inputs: [5, 6]}
]
Output: [{"t": 200, inputs: [1,2]}, {"t": 450, inputs: [5, 6]}]
Explanation:
The 1st call is delayed by 150ms and ran at 200ms. The inputs were (1, 2).

The 2nd call is cancelled by the 3rd call
The 3rd call is delayed by 150ms and ran at 450ms. The inputs were (5, 6).

Constraints:

- **0 <= t <= 1000**
- **1 <= calls.length <= 10**
- **0 <= calls[i].t <= 1000**
- **0 <= calls[i].inputs.length <= 10**

```
function debounce (fn, t) {
  let timeout;
    return function(...args) {
        clearTimeout(timeout);
        timeout = setTimeout(() => {
            fn(...args);
        }, t);
    }
};
```

133. Write a class that allows getting and setting key-value pairs, however a time until expiration is associated with each key.

The class has three public methods:
set(key, value, duration): accepts an integer key, an integer value, and a duration in milliseconds. Once the duration has elapsed, the key should be inaccessible. The method should return true if the same un-expired key already exists and false otherwise. Both the value and duration should be overwritten if the key already exists.
get(key): if an un-expired key exists, it should return the associated value. Otherwise it should return -1.
count(): returns the count of un-expired keys.

Example 1:
Input:
actions = ["TimeLimitedCache", "set", "get", "count", "get"]
values = [[], [1, 42, 100], [1], [], [1]]
timeDelays = [0, 0, 50, 50, 150]
Output: [null, false, 42, 1, -1]
Explanation:
At t=0, the cache is constructed.
At t=0, a key-value pair (1: 42) is added with a time limit of 100ms. The value doesn't exist so false is returned.
At t=50, key=1 is requested and the value of 42 is returned.
At t=50, count() is called and there is one active key in the cache.
At t=100, key=1 expires.
At t=150, get(1) is called but -1 is returned because the cache is empty.

Example 2:
Input:
actions = ["TimeLimitedCache", "set", "set", "get", "get", "get", "count"]
values = [[], [1, 42, 50], [1, 50, 100], [1], [1], [1], []]
timeDelays = [0, 0, 40, 50, 120, 200, 250]
Output: [null, false, true, 50, 50, -1, 0]
Explanation:
At t=0, the cache is constructed.
At t=0, a key-value pair (1: 42) is added with a time limit of 50ms. The value doesn't exist so false is returned.
At t=40, a key-value pair (1: 50) is added with a time limit of 100ms. A non-expired value already existed so true is returned and the old value was overwritten.
At t=50, get(1) is called which returned 50.
At t=120, get(1) is called which returned 50.
At t=140, key=1 expires.
At t=200, get(1) is called but the cache is empty so -1 is returned.
At t=250, count() returns 0 because the cache is empty.

Constraints:

- **0 <= key, value <= 10^9**
- **0 <= duration <= 1000**
- **1 <= actions.length <= 100**
- **actions.length === values.length**
- **actions.length === timeDelays.length**
- **0 <= timeDelays[i] <= 1450**
- **actions[i] is one of "TimeLimitedCache", "set", "get" and "count"**

- **First action is always "TimeLimitedCache" and must be executed immediately, with a 0-millisecond delay**

```
var TimeLimitedCache = function() {
    this.cache = {};
};

TimeLimitedCache.prototype.set = function(key, value, duration) {
    // Check if the key already exists and is not expired
    const currentTime = Date.now();
    let isExisting = false;

    if (this.cache[key] && this.cache[key].expiry > currentTime) {
        isExisting = true; // Key exists and has not expired
    }

    // Set the key, value and expiration time
    this.cache[key] = {
        value: value,
        expiry: currentTime + duration
    };

    return isExisting;
};

TimeLimitedCache.prototype.get = function(key) {
    const currentTime = Date.now();
    if (this.cache[key] && this.cache[key].expiry > currentTime) {
        return this.cache[key].value;
    } else
        return -1;
};

TimeLimitedCache.prototype.count = function() {
    const currentTime = Date.now();
    let count = 0;
    for (let key in this.cache) {
        if (this.cache[key].expiry > currentTime)
            count++;
    }
    return count;
};
```

134. You are required to write a function that generates a list of N top brand names for a specific user based on their preferences and gender. The function should return a Promise that either resolves with an array of exactly N brand names or rejects with a custom error if there are not enough brands to fulfill the request.

Rules for Compiling the List

- **User's Liked Brands: Every user has a list of brands they like most. If the user has at least N liked brands, the function should return the first N brands from this list.**
- **Top Brands by Gender: If the user's liked brands are insufficient (i.e., fewer than N), the function should fill the remaining slots with the top brands from the list of brands most popular among users of the same gender.**
- **Error Handling: If the combination of the user's liked brands and the top brands for their gender still does not provide enough brands (i.e., fewer than N), the function should reject the Promise with a custom error indicating "Not enough data".**

function solution(U, N);

Input

- **U: A user object with the following properties:**
  - **id: The unique identifier of the user.**
  - **gender: The gender of the user (used to fetch top brands for that gender).**
- **N: The number of top brand names to return.**

Output

- **The function should return a Promise that:**
  - **Resolves with an array of exactly N brand names in the format: ["Brand1", "Brand2", ...].**
  - **Rejects with a CustomError and the message "Not enough data" if fewer than N brands are available.**

Example Scenarios
Assume the following:

- **getLikedBrands(U.id) returns:**

```
[
  { id: 1, name: "Logestyx" },
  { id: 10, name: "GladLear" }
]
```

- **getTopBrandsForGender(U.gender) returns:**

```
[
  { id: 6, name: "Burylaze Slapgait" },
  { id: 1, name: "Logestyx" },
  { id: 7, name: "Izarpure" }
]
```

Test Cases:

For N = 1:

- **The function should return a Promise that resolves with:**
  ["Logestyx"]

- **Explanation: The user's liked brands list has at least 1 brand, so the first brand is returned.**

For N = 3:

- **The function should return a Promise that resolves with:**
  ["Logestyx", "GladLear", "Burylaze Slapgait"]

- **Explanation: The user's liked brands list has 2 brands, so the remaining 1 brand is taken from the top brands for their gender.**

For N = 4:

- **The function should return a Promise that resolves with:**
  ["Logestyx", "GladLear", "Burylaze Slapgait", "Izarpure"]

- **Explanation: The user's liked brands list has 2 brands, and the top brands for their gender provide 2 additional brands.**

For N = 5:

- **The function should return a Promise that rejects with a CustomError and the message "Not enough data".**

- **Explanation: The combined list of liked brands and top brands for the user's gender has only 4 brands, which is insufficient for N = 5.**

Technical Details

- **Data Access:**
  - **Use getLikedBrands(U.id) to fetch the user's liked brands.**
  - **Use getTopBrandsForGender(U.gender) to fetch the top brands for the user's gender.**

- **Error Handling:**
  - **If either of the above functions fails or the combined list has fewer than N brands, reject the Promise with a CustomError.**

```
function getLikedBrands(userId) {
   return new Promise((resolve) => {
      setTimeout(() => {
         const likedBrands = [
            { id: 1, name: "Logestyx" },
            { id: 10, name: "GladLear" }
         ];
         resolve(likedBrands);
      }, 100);
   });
}

function getTopBrandsForGender(gender) {
   return new Promise((resolve) => {
      setTimeout(() => {
         // Mock data for top brands by gender
         const topBrands = [
            { id: 6, name: "Burylaze Slapgait" },
            { id: 1, name: "Logestyx" },
            { id: 7, name: "Izarpure" }
         ];
         resolve(topBrands);
      }, 100);
   });
}

function solution(U, N) {
   return new Promise((resolve, reject) => {
      const likedBrandsPromise = getLikedBrands(U.id);
      const topBrandsPromise = getTopBrandsForGender(U.gender);

      Promise.all([likedBrandsPromise, topBrandsPromise])
         .then(([likedBrands, topBrands]) => {

            const likedBrandNames = likedBrands.map(brand => brand.name);


            const topBrandNames = topBrands
               .map(brand => brand.name)
               .filter(name => !likedBrandNames.includes(name));

            const combinedBrands = [...likedBrandNames, ...topBrandNames];

            if (combinedBrands.length >= N) {
               resolve(combinedBrands.slice(0, N));
            } else {
               reject(new CustomError("Not enough data"));
            }
   })
         .catch(() => {
            reject(new CustomError("Not enough data"));
         });
   });
```

```
}

// Example user object
const user = {
   id: 123,
   gender: "male"
};

// Test the solution function with dummy data
solution(user, 3)
   .then((brands) => {
      console.log("Resolved:", brands);
   })
   .catch((error) => {
      console.error("Rejected:", error.message);
   });
```

135. Given a function fn and a time in milliseconds t, return a debounced version of that function.

A debounced function is a function whose execution is delayed by t milliseconds and whose execution is cancelled if it is called again within that window of time. The debounced function should also receive the passed parameters.
For example, let's say t = 50ms, and the function was called at 30ms, 60ms, and 100ms. The first 2 function calls would be cancelled, and the 3rd function call would be executed at 150ms.
If instead t = 35ms, The 1st call would be cancelled, the 2nd would be executed at 95ms, and the 3rd would be executed at 135ms.

```
function debounce(fn, delay) {
   let timer;
   return function (...args) {
      const context = this
      clearTimeout(timer)
      timer = setTimeout(() =>
         fn.apply(context, args)
         ,delay)
   }
};
```

136. Implement a throttle function that accepts a function fn and a wait time t, and returns a throttled version of fn that, when invoked repeatedly, will only actually call fn once every t milliseconds.

If multiple calls are made during the wait period, they should be ignored.

```
function throttle(fn, t) {
  let lastCallTime = 0;

  return function (...args) {
    const now = Date.now();

    if (now - lastCallTime >= t) {
      lastCallTime = now;
      fn.apply(this, args);
    }
  };
}
```

PolyFills

137. Polyfill for promise.all

```
Promise.all = (promises) => {
  const result = []
  let completed = 0

  return new Promise((resolve, reject) => {
    if (promises.length === 0) {
      return resolve([]) // handle empty input early
    }

    promises.forEach((promise, index) => {
      Promise.resolve(promise) // in case it's not already a Promise
        .then((data) => {
          result[index] = data // maintain order
          completed++
          if (completed === promises.length) {
            resolve(result)
          }
        })
        .catch((error) => {
          reject(error) // reject as soon as one fails
        })
    })
  })
}
```

138. Polyfill for promise.any

```
Promise.any = (promises) => {
  const result = []
  let errored = 0

  return new Promise((resolve, reject) => {
    if (promises.length === 0) {
      return reject(new AggregateError([], "All promises were rejected"))
    }

    promises.forEach((promise, index) => {
      Promise.resolve(promise)
        .then((data) => {
          resolve(data)
        })
        .catch((error) => {
          result[index] = error
          errored++
          if (errored === promises.length) {
            reject(new AggregateError(result, "All promises were rejected"))
          }
        })
    })
  })
}
```

139. Polyfill for promise.allSettled

```
Promise.allSettled = (promises) => {
  const result = []
  let settledCount = 0

  return new Promise((resolve, reject) => {
    if (promises.length === 0) {
      resolve(result)
    }

    promises.forEach((promise, index) => {
      Promise.resolve(promise)
        .then((value) => {
          result[index] = {
            status: "fulfilled",
            value,
          }
        })
        .catch((err) => {
          result[index] = {
            status: "rejected",
            reason: err,
          }
        })
        .finally(() => {
          settledCount++
          if (settledCount === promises.length) {
            resolve(result)
          }
        })
    })
  })
}
```

140. Promise for promise.race

```
Promise.race = (promises) => {
  return new Promise((resolve, reject) => {
    promises.forEach((promise, index) => {
      Promise.resolve(promise).then((data) => {
        resolve(data)
      }).catch(err => {
        reject(err)
      })
    })
  })
}
```

141. Polyfill for array.map

```
Array.prototype.map = function (callBack) {
  if (typeof callBack !== "function") {
    throw new TypeError(`${callBack} is not a function`)
  }
  const result = []
  for (let i = 0; i < this.length; i++) {
    if (i in this)
    result[i]= callBack(this[i],i,this)
  }
  return result;
}
```

142. Polyfill for array.filter

```
Array.prototype.filter = function (callBack) {
  if (typeof callBack !== "function") {
    throw new TypeError(`${callBack} is not a function`)
  }
  const result = []
  for (let i = 0; i < this.length; i++) {
    if (i in this && callBack(this[i], i, this)) {
      result.push(this[i])
    }
  }
  return result
}
```

143. Polyfill for array.reduce

```
Array.prototype.reduce = function (callback, initialValue) {
  if (typeof callback !== "function") {
    throw new TypeError(`${callback} is not a function`);
  }

  const array = this;
  let accumulator,
    startIndex = 0;

  if (arguments.length >= 2) accumulator = initialValue;
  else {
    while (startIndex < array.length && !(startIndex in array)) {
      startIndex++;
    }

    if (startIndex >= array.length) {
      throw new TypeError("Reduce of empty array with no initial value");
    }
    accumulator = array[startIndex];
    startIndex++;
  }

  for (let i = startIndex; i < array.length; i++) {
    if(i in array)
    accumulator = callback(accumulator, this[i], i, this);
  }
  return accumulator;
};
```

144. Polyfill for array.forEach

```
Array.prototype.forEach = function (callBack) {
  if (typeof callBack !== "function") {
    throw new TypeError(`${callBack} is not a function`)
  }
  for (let i = 0; i < this.length; i++) {
    if (i in this)
    callBack(this[i], i, this)
  }
}
```

145. Polyfill for call

```
Function.prototype.call = function(context,...args){
  if(typeof this!=='function'){
    throw new TypeError("Call must be called on a function");
  }

context = context !=null ? Object(context):globalThis


const key = Symbol('fn')

  context[key] = this
  const result = context[key](...args);
  delete context[key]
  return result
}
```

146. Polyfill for apply

```
Function.prototype.apply = function(context, args = []) {
  if (typeof this !== 'function') {
    throw new TypeError("Apply must be called on a function");
  }

  context = context!=null ? Object(context):globalThis


  const key = Symbol('fn');
  context[key] = this;

  const result = context[key](...args);

  delete context[key];
  return result;
}
```

147. Polyfill for bind

```
Function.prototype.bind = function(context, ...bindArgs) {
  if (typeof this !== 'function') {
    throw new TypeError("Bind must be called on a function");
  }

  return (...callArgs) => {
    context = context ? Object(context) : globalThis
    const key = Symbol('fn');
    context[key] = this;

    const result = context[key](...bindArgs, ...callArgs);
    delete context[key];
    return result;
  };
};
```

Classes

148. Design a Calculator class. The class should provide the mathematical operations of addition, subtraction, multiplication, division, and exponentiation. It should also allow consecutive operations to be performed using method chaining.
The Calculator class constructor should accept a number which serves as the initial value of result.

Your Calculator class should have the following methods:
add - This method adds the given number value to the result and returns the updated Calculator.
subtract - This method subtracts the given number value from the result and returns the updated Calculator.
multiply - This method multiplies the result by the given number value and returns the updated Calculator.
divide - This method divides the result by the given number value and returns the updated Calculator. If the passed value is 0, an error "Division by zero is not allowed" should be thrown.
power - This method raises the result to the power of the given number value and returns the updated Calculator.
getResult - This method returns the result.
Solutions within 10^{-5} of the actual result are considered correct.

Example 1:
Input:
actions = ["Calculator", "add", "subtract", "getResult"],
values = [10, 5, 7]
Output: 8
Explanation:
new Calculator(10).add(5).subtract(7).getResult() // 10 + 5 - 7 = 8

Example 2:
Input:
actions = ["Calculator", "multiply", "power", "getResult"],
values = [2, 5, 2]
Output: 100
Explanation:
new Calculator(2).multiply(5).power(2).getResult() // (2 * 5) ^ 2 = 100

Example 3:
Input:
actions = ["Calculator", "divide", "getResult"],
values = [20, 0]
Output: "Division by zero is not allowed"
Explanation:
new Calculator(20).divide(0).getResult() // 20 / 0

The error should be thrown because we cannot divide by zero.

Constraints:
actions is a valid JSON array of strings
values is a valid JSON array of numbers

2 <= actions.length <= $2 * 10^4$
1 <= values.length <= $2 * 10^4 - 1$
actions[i] is one of "Calculator", "add", "subtract", "multiply", "divide", "power", and "getResult"
First action is always "Calculator"
Last action is always "getResult"

```
class Calculator {
    constructor(value) {
    this.result = value;
  }

  add(value) {
    this.result += value;
    return this;
  }

  subtract(value) {
    this.result -= value;
    return this;
  }

  multiply(value) {
    this.result *= value;
    return this;
  }

  divide(value) {
    if (value === 0) {
      throw "Division by zero is not allowed";
    }
    this.result /= value;
    return this;
  }

  power(value) {
    this.result = Math.pow(this.result, value);
    return this;
  }

  getResult() {
    return this.result;
  }
}
```

149. Create a class ArrayWrapper that accepts an array of integers in its constructor. This class should have two features:

When two instances of this class are added together with the + operator, the resulting value is the sum of all the elements in both arrays.
When the String() function is called on the instance, it will return a comma separated string surrounded by brackets. For example, [1,2,3].

Example 1:

Input: nums = [[1,2],[3,4]], operation = "Add"
Output: 10
Explanation:
const obj1 = new ArrayWrapper([1,2]);
const obj2 = new ArrayWrapper([3,4]);
obj1 + obj2; // 10
Example 2:

Input: nums = [[23,98,42,70]], operation = "String"
Output: "[23,98,42,70]"
Explanation:
const obj = new ArrayWrapper([23,98,42,70]);
String(obj); // "[23,98,42,70]"
Example 3:

Input: nums = [[],[]], operation = "Add"
Output: 0
Explanation:
const obj1 = new ArrayWrapper([]);
const obj2 = new ArrayWrapper([]);
obj1 + obj2; // 0

Constraints:

0 <= nums.length <= 1000
0 <= nums[i] <= 1000
Note: nums is the array passed to the constructor

```
var ArrayWrapper = function(nums) {
    this.nums = nums
};

ArrayWrapper.prototype.valueOf = function() {
  return this.nums.reduce((sum, num) => sum + num, 0);
}

ArrayWrapper.prototype.toString = function() {
      return `[${this.nums.join(',')}]`;
}
```

150. Implement the LRUCache class:

- **LRUCache(int capacity) Initialize the LRU cache with positive size capacity.**
- **int get(int key) Return the value of the key if the key exists, otherwise return -1.**
- **void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, evict the least recently used key.**
- **The functions get and put must each run in O(1) average time complexity.**

```
class LRUCache {
    constructor(capacity) {
        this.capacity = capacity
        this.cache = new Map();
    }

    get(key) {
        if (!this.cache.has(key)) return -1
        const value = this.cache.get(key)
        this.cache.delete(key)
        this.cache.set(key, value)

        return value
    }

    put(key, value) {
        if (this.cache.has(key)) {
            this.cache.delete(key)
        }
        this.cache.set(key, value)

        if (this.cache.size > this.capacity) {
            const lruKey = this.cache.keys().next().value
            this.cache.delete(lruKey)
        }
    }
}
```

151. Design and implement a class EventEmitter that mimics the behavior of Node.js-style event emitters.

Implement the following methods:

- **on(eventName: string, callback: Function): void**

  Registers a callback for the given event.
  The callback should be called every time the event is emitted.

- **once(eventName: string, callback: Function): void**

  Registers a one-time callback for the given event.
  The callback should be called only the first time the event is emitted and then removed.

- **off(eventName: string, callback: Function): void**

  Removes the specified callback for the event.
  If the callback doesn't exist, do nothing.

- **emit(eventName: string, ...args: any[]): void**
  Triggers all callbacks associated with the given event, passing any additional arguments to the callbacks.
  If no listeners are present, do nothing.

```
class EventEmitter {
   constructor() {
      this.events = new Map()
   }

   on(eventName, callback) {
      if (!this.events.has(eventName)) {
         this.events.set(eventName,[])
      }
      this.events.get(eventName).push({callback,once:false})
   }

   once(eventName, callback) {
      if (!this.events.has(eventName)) {
         this.events.set(eventName, [])
      }
      this.events.get(eventName).push({ callback, once: true})
   }

   off(eventName, callback) {
      if (!this.events.has(eventName)) return;
      const listeners = this.events.get(eventName).filter(event => event.callback !== callback)
      this.events.set(eventName,listeners)
   }

   emit(eventName, ...args) {
      if (!this.events.has(eventName)) return;
      const listeners = this.events.get(eventName)
      const remaining = []
```

```
        for (const listener of listeners) {
            listener.callback(...args)
            if (!listener.once) {
                remaining.push(listener)
            }
        }
        this.events.set(eventName, remaining);
    }
}
```

152. Design an EventEmitter class. This interface is similar (but with some differences) to the one found in Node.js or the Event Target interface of the DOM. The EventEmitter should allow for subscribing to events and emitting them.

Your EventEmitter class should have the following two methods:

- **subscribe - This method takes in two arguments: the name of an event as a string and a callback function. This callback function will later be called when the event is emitted. An event should be able to have multiple listeners for the same event. When emitting an event with multiple callbacks, each should be called in the order in which they were subscribed. An array of results should be returned. You can assume no callbacks passed to subscribe are referentially identical.**
  The subscribe method should also return an object with an unsubscribe method that enables the user to unsubscribe. When it is called, the callback should be removed from the list of subscriptions and undefined should be returned.
- **emit - This method takes in two arguments: the name of an event as a string and an optional array of arguments that will be passed to the callback(s). If there are no callbacks subscribed to the given event, return an empty array. Otherwise, return an array of the results of all callback calls in the order they were subscribed.**

Example 1:
Input:
actions = ["EventEmitter", "emit", "subscribe", "subscribe", "emit"],
values = [[], ["firstEvent"], ["firstEvent", "function cb1() { return 5; }"], ["firstEvent", "function cb1() { return 6; }"], ["firstEvent"]]
Output: [[],["emitted",[]],["subscribed"],["subscribed"],["emitted",[5,6]]]
Explanation:
const emitter = new EventEmitter();
emitter.emit("firstEvent"); // [], no callback are subscribed yet
emitter.subscribe("firstEvent", function cb1() { return 5; });
emitter.subscribe("firstEvent", function cb2() { return 6; });
emitter.emit("firstEvent"); // [5, 6], returns the output of cb1 and cb2

```
class EventEmitter {
    constructor() {
        this.events = new Map();
    }
    subscribe(eventName, callback) {
        if (!this.events.has(eventName)) {
            this.events.set(eventName, new Set());
        }
        const subscriptions = this.events.get(eventName);
        subscriptions.add(callback);

        return {
            unsubscribe: () => {
                // Remove the callback from the subscription list
                subscriptions.delete(callback);
            }
        };
    }

    emit(eventName, args = []) {
        if (!this.events.has(eventName)) {
            return [];
        }

        const subscriptions = this.events.get(eventName);

        const results = [];
        for (const callback of subscriptions) {
            results.push(callback(...args));
        }

        return results;
    }
}
```

153. Create a singleton pattern using a class

```
class SingleTon {
  constructor(url) {
    if (SingleTon.instance) {
      throw new Error("Use getInstance() instead of new")
    }
    this.message = url
  }

  static getInstance(url) {
    if (!SingleTon.instance) {
      SingleTon.instance = new SingleTon(url)
    }
    return SingleTon.instance
  }
}
```