

For JavaScript Theoretical questions:

"<https://github.com/sudheerj/javascript-interview-questions>"

For Reactjs theory questions:

<https://github.com/sudheerj/reactjs-interview-questions/tree/master/.github/workflows>

For Google, Facebook, Microsoft coding challenges: <https://youtube.com/c/KevinNaughtonJr>

Jest: <https://github.com/sapegin/jest-cheat-sheet>

<https://plainenglish.io/blog/50-javascript-output-questions>

<https://github.com/priya42bagde/javascript-interview-questions?organization=priya42bagde&organization=priya42bagde#what-is-json-and-its-common-operations> ---> IMP

<https://learnersbucket.com/javascript-sde-cheat-sheet/>

<https://javascript.info/>

<https://frontenddeveloperinterview.netlify.app/>

```
=====
=====
=====
```

Code 1: Remove Duplicate characters from String

```
function removeDuplicateCharacters() {
  var string='priya riya supriya'
  let result= string.split("").filter((item, index, arr)=> {
    return arr.indexOf(item) == index;
  }).join("");
  return result;
}
console.log(removeDuplicateCharacters());
```

```
=====
=====
=====
```

Code 2: Remove Duplicate characters from array of element and find the count of an elements using set object

```
var arr = [55, 44, 55,67,67,67,67,8,8,8,8,8,65,1,2,3,3,34,5];
var unique = [...new Set(arr)]
console.log(unique) //output: [55, 44, 67, 8, 65, 1, 2, 3, 34, 5]
console.log(unique.length) //output: 10
```

```
=====
=====
=====
```

Code 3: Remove Duplicate characters from array of element using filter

```
var myArray = ['a', 1, 'a', 2, '1'];
var unique = myArray.filter((value, index, arr) => arr.indexOf(value) === index);
```

```
=====
=====
```

=====

Code 4:String reverse without reversing of individual words (Array of elements can be reverse with reverse() method but for string it is won't possible so required to split and then join()).

```
function removeDuplicates(){
  var string ="India is my country"
  let result = string.split("").reverse().join("").split(' ').reverse().join(' ')
  return result
}
console.log(removeDuplicates())
output = "aidnl si ym yrtnuoc"
```

```
var reverseWords = function(s) {
  let res = "";
  let word = "";
  for (let c of s) {
    if (c === ' ') {
      res += word + c;
      word = "";
    } else {
      word = c + word;
    }
  }
  return res + word;
};
console.log(reverseWords("priya bagde"))
```

=====

=====

=====

Code 5:String reverse with reversing of individual words

```
function withoutReverse(){
  var string ="India is my country"
  let result = string.split("").reverse().join("")
  return result
}
console.log(withoutReverse())
output = "yrtnuoc ym si aidnl"
```

```
=====
=====
=====
Code 6:String reverse without using inbuilt function
```

```
function Reverse(){
  var string ="India is my country";
  var result="";
  for( var i=string.length-1; i>=0 ; i-- ) {
    result=result+string[i] }
  return result
}
console.log(Reverse())
output = "yrtnuoc ym si aidnI"
```

```
=====
=====
=====
Code 7: Find factorial of user input number
```

```
const number = parseInt(prompt('Enter a positive integer: '));
if (number < 0) { console.log('Error! Factorial for negative number does not exist.')}
else if (number === 0) { console.log(`The factorial is 1.`)}
else {
  let fact = 1;
  for (i = 1; i <= number; i++) {
    fact *= i;
  }
  console.log(`The factorial is ${fact}.`);
}
```

```
=====
=====
=====
Code 8:Anagram
```

```
function checkStringsAnagram() {
  var a="Army";
  var b="Mary"
  let str1 = a.toLowerCase().split("").sort().join("");
  let str2 = b.toLowerCase().split("").sort().join("");
  if(str1 === str2){
    console.log("True");
  }
  else {
    console.log("False");
  }
}
```

```
=====
=====
=====
Code 9: Swapping of 2 numbers with third variable
```

```
let a=10;
let b=20;
let c;
  c=a;
  a=b;
  b=c;
console.log (a,b,c)
```

```
=====
=====
=====
Code 10: Swapping of 2 numbers without third variable
```

```
let a=10;
let b=20;
  a=a+b //30
  b=a-b //10
  a=a-b //20
console.log (a,b)
```

```
=====
=====
=====
Code 11: To check the string or number is palindrome or not( ex: 121, madam, anna) using
reverse method
```

```
function checkPalindrome(){
  const string = "anmna"
  let arr= string.split("").reverse().join("")
  //console.log(arr)
  if (string==arr){
    console.log("Palindrome")
  }
  else{
    console.log("Not Palindrome")
  }
}
checkPalindrome()
```

```
=====
```

```
=====
```

```
=====
```

Code 12: To check the string or number is palindrome or not(ex: 121, madam, anna) using diving length by 2 and then comparing

```
function checkPalindrome(){
  const string = "12321"
  let len = string.length;
  for (i=0; i<len/2;i++){
    if (string[i]!==string[len-1-i]){
      console.log("Not Palindrome")
    }
    else{
      console.log(" Palindrome")
    }
  }
}
checkPalindrome()
```

```
=====
```

```
=====
```

```
=====
```

Code 13: To find longest word from a string using (for of) /*for(var i=0; i>=num; i++) means iterate by indexing*/ /*for (var word of words) means iterate by an elements not by indexing*/

```
function longestWord(){
  let string = "supriya is a masooooooooom good girl"
  var words= string.split(' ')
  var longest=""
  for(var word of words){
    console.log(word)
    if (word.length > longest.length)
    {
      longest=word;
    }
  }
  return longest.length
}
longestWord()
```

```
-----
```

```
function longestWord(){
  let string = "supriya is a hahahahaha good girl"
  var arr= string.split(' ')
  var longest=""
```

```

for(var i=0; i<arr.length; i++){

    if (arr[i].length > longest.length)
    {
        longest=arr[i];
    }
}
return longest
}
console.log(longestWord())
=====
=====
=====

```

Code 14: To find longest word from a string using functions

```

function findLongestWord() {
    var str = "Priya is a goog girl and having hardworking skill"
    var longestWord = str.split(' ').sort((a, b) => {return b.length - a.length }); //in desc order //from
greater to smallest word
    console.log(longestWord[0]);
    console.log(longestWord[0].length);
}
findLongestWord();
=====
=====
=====

```

Code 15: To find longest word from a string using custom code

```

function longest() {
    var str ="Priya is a good girl and having hardworking skills"
    var words = str.split(' ');
    var longest = "";
    for (var i = 0; i < words.length; i++) {
        if (words[i].length > longest.length) {
            longest = words[i];
        }
    }
    console.log(longest)
    return longest;
}
longest();

```

=====
=====
=====
Code 16: To find longest common string from array of strings

```
function longestCommonString(){
  array=["go","google","gosh"]
  var arr = array.sort()
  var i=0;
  while(arr[0].length>0 && arr[0].charAt(i)===arr[arr.length-1].charAt(i)){
    i++;
  }
  console.log(arr[0].substring(0,i)) // "go"
  return arr[0].substring(0,i)
}
longestCommonString()
```

```
function longestCommonString(){
  array=["got","google","gosh"]
  var arr = array.sort()
  var i=0;
  while(arr[0].length>0 && arr[0][i]===arr[arr.length-1][i]){
    i++;
  }
  console.log(arr[0].slice(0,i))
}
longestCommonString()
```

```
function longestCommonString(){
  let array=["go","google","gosh"]
  var arr = array.sort((a,b)=>a.length-b.length)
  let result = ""
  for(let i=0; i<arr[0].length; i++){
    if(arr[0][i]===arr[arr.length-1][i]){
      result+=arr[0][i]
    }
  }
  return result
}
console.log(longestCommonString())
```

=====
=====
=====
Code 17: To find vowels and its count in a given string

```
function vowelCounts(){  
  vowels=["a","i","e","o","u"]  
  var str ="priya"  
  count=0;  
  for(var letter of str.toLowerCase())  
  {  
    if(vowels.includes(letter))  
    {  
      count++;  
      console.log(letter)  
    }  
  }  
  console.log(count)  
  return count  
}  
vowelCounts()
```

=====
=====
=====

Code 18: To find character occurrence from the string

```
function characterOccurance(str,letter){  
  let count =0;  
  for(var i=0; i<str.length-1; i++){  
    if(str.charAt(i)==letter)  
    {  
      count++  
    }  
  }  
  console.log(count)  
  return count  
}  
characterOccurance("priyapri", "p")
```

=====
=====
=====

Code 19: To find a first pair whose sum is zero

```
function getSumPairZero(array)  
{  
  for(let number of array)
```



```

    {
      for(let i=1; i<array.length; i++)
      {
        if(number+array[i]==0)
        {
          return [number, array[i]]
        }
      }
    }
  }
}
const result = getSumPairZero([-5,-4,-3,-2,-1,0,1,2,3,4,5])
console.log(result)

```

```

-----
function getSumPairZero(array)
{
  for(let j=0; j<array.length;j++)
  {
    for(let i=1; i<array.length; i++)
    {
      if(array[j]+array[i]==0)
      {
        return [array[j], array[i]]
      }
    }
  }
}

```

```

const result = getSumPairZero([-5,-4,-3,-2,-1,0,1,2,3,4,5])
console.log(result)

```

```

=====
=====
=====

```

Code 20: To find a first pair whose sum is zero using indexing //Firstly do a sort here

```

function getSumPairZero(array)
{
  let left = 0;
  let right = array.length-1;
  while(left<right)
  {
    sum = array[left]+array[right]
    if(sum===0){
      return [array[left],array[right]]
    }else if(sum>0){
      right--;
    }
  }
}

```

```

    }else{
        left++;
    }
}
}
const result = getSumPairZero([-5,-4,-3,-2,-1,0,2,4,6,8])
console.log(result)
=====
=====
=====

```

Code 21: To find the largest pair of the 2 elements using indexing with unsorted elements

```

function largestPairSumofTwo(numbers){
    const num = numbers.sort((a, b) => b - a);
    console.log(num)
    return num[0] + num[1];
}
const result = largestPairSumofTwo([9,7,8,4,5,6,1,2,3])
console.log(result)
=====
=====
=====

```

Code 22: To find the largest pair of the 2 elements using indexing with sorted elements

```

function largestPairSumofTwo(num){
    return num[num.length-1] + num[num.length-2];
}
const result = largestPairSumofTwo([1,2,3,4,5,6,7,8,9])
console.log(result)
=====
=====
=====

```

Code 23: To find the index of an element from an array

```

const letters = ['a', 'b', 'c']
const index = letters.indexOf('b')
console.log(index) // `1`
=====
=====
=====

```

Code 24: Fibonacci Series (0,1,1,2,3,5,8,13....)

```

function fibonacciSeries(){
    const number = parseInt(prompt('Enter the number of terms: '));
    let n1 = 0, n2 = 1, nextTerm, arr=[]
    arr.push(n1)
    arr.push(n2)

```

```

    for (let i = 1; i <= number; i++)
    {
        console.log(n1);
        nextTerm = n1 + n2;
        n1 = n2;
        n2 = nextTerm;
        arr.push(nextTerm)
    }
    return arr
}
console.log(fibonacciSeries())
=====
=====
=====

```

Code 25: Fibonacci Series (0,1,1,2,3,5,8,13....) where keeping in array

```

function listFibonacci(n) {
    var arr = [0, 1]
    for (var i = 1; i < n; i++)
        arr.push(arr[i] + arr[i - 1])

    return arr
}
console.log(listFibonacci(4))
-----

```

```

function listFibonacci(n) {
    var arr = [0, 1]
    for (var i = 0; i < n; i++)
        arr.push(arr[i] + arr[i + 1])

    return arr
}
console.log(listFibonacci(4))
=====
=====
=====

```

Code 26: Finding a missing elements in an array and then add with existing elements. (-1 means if elements not found then it will return always -1 as per rule)

```

function missingElement(){
    var a = [1,2,5]
    var missing = [];
    for (var i = 1; i <= 6; i++)
    {
        if (a.indexOf(i) == -1)

```

```

    {
        missing.push(i);
    }
}
console.log(missing) //missing array
console.log(a.concat(missing).sort()); //actual+missing elements
}
missingElement()

```

```

=====
=====
=====

```

Code 27: Find the missing no. in an array

```

function missing(arr) {
    var x = 0;
    for (var i = 0; i < arr.length; i++) {
        x = x + 1;
        if (arr[i] != x) {
            return(x); //9
        }
    }
}

```

```

missing([1, 2, 3, 4, 5, 6, 7, 8, 10])

```

```

function missing(arr) {
    for (var i = 0, x=1; i < arr.length; x++,i++) {
        if (arr[i] != x) { //index value comparing with pointer
            return x; //9
        }
    }
}

```

```

console.log(missing([1, 2, 3, 4, 5, 6, 7, 8, 10]))

```

```

=====
=====
=====

```

Code 28: Sorting of an string/character

```

function sorting(arr) {
    return arr.sort()
}
console.log(sorting(["d","g","y","e","r","p"]))

```

```

=====
=====
=====

```

Code 29: Sorting of an number

```
function sorting(arr) {
  return arr.sort((a,b)=>{return a-b})
}
console.log(sorting([1,23,34,2,76,78])) //[1, 2, 23, 34, 76, 78]
```

```
=====
=====
=====
```

Code 30: To check if given number is prime or not

```
function isPrime(num) {
  if(num < 2) return false;
  for (let k = 2; k < num; k++){
    if( num % k == 0){ return false}
  }
  return true;
}
console.log(isPrime(17)) //true
```

```
=====
=====
=====
```

Code 31: To print all the numbers from 2 to 100

```
for (let i = 2; i <= 100; i++) {
  let flag = 0;
  for (let j = 2; j < i; j++) {
    if (i % j == 0) {
      flag = 1;
      break;
    }
  }
  if (i > 1 && flag == 0) {
    console.log(i);
  }
}
```

```
-----
for (let i = 2; i <= 100; i++)
{
  let flag = 0;
  for (let j = 2; j < i; j++) { //2<2 //2<3 //3<4
    if (i % j == 0) {
      flag = 1;
      break;
    }
  }
  if (i > 1 && flag == 0)
```

```

    {
        document.write(i+ "</br>");
    }
}

```

```

=====
=====
=====

```

Code 32: To find unique values from 2 arrays and keep into one array.

```

function uniqueElements(arr1,arr2){
    let arr =[...arr1,...arr2];
    let array =[...new Set(arr)]
    console.log(array)
}
uniqueElements([1,2,3,4,4],[2,3,4,5,6])

```

```

=====
=====
=====

```

Code 33: Find first duplicate element from an array

```

function firstDuplicate() {
    let arr = [1,2,2,5,5];
    let data = {};
    for (var item of arr) {
        if (data[item]) {
            return item
        } else {
            data[item] = item
            console.log(data[item])
        }
    }
    return -1
}

```

```

console.log(firstDuplicate())

```

```

=====
=====
=====

```

Code 34: Write a program that prints the numbers from 1 to 100. But for multiples of three, print "Fizz" instead of the number, and for the multiples of five, print "Buzz".

For numbers which are multiples of both three and five, print "FizzBuzz"

```

for (var i=1; i <= 20; i++)
{
    if (i % 15 == 0)
        console.log("FizzBuzz");
    else if (i % 3 == 0)

```

```

        console.log("Fizz");
    else if (i % 5 == 0)
        console.log("Buzz");
    else
        console.log(i);
}

```

```

=====
=====
=====

```

Code 35: Uppercase of each first letter of a words

```

function upperCaseFirsstLetter(){
    var string ="India is my country";
    var words = string.toLowerCase().split(" ")
    for( var i=0; i<words.length; i++) {
        words[i]=words[i][0].toUpperCase() + words[i].slice(1) //slice is used here to give all the
letters except first letter.
    }
    return words.join(" ")
}
console.log(upperCaseFirsstLetter())

```

```

=====
=====
=====

```

Code 36: Uppercase of each first letter of a words using map function

```

function upperCaseFirsstLetter(){
    var string ="India is my country";
    var words = string.toLowerCase().split(" ").map((ele)=>{
        return ele[0].toUpperCase() + ele.slice(1)
    })
    return words.join(" ")
}
console.log(upperCaseFirsstLetter())

```

```

=====
=====
=====

```

Code 37: To check ending of the string with given character/s using inbuild function

```

function confirmEnding(str,target){
    return str.endsWith(target) //true
}
console.log(confirmEnding("priya","a"))

```

```

=====
=====
=====

```

Code 38: To check ending of the string with given character/s using custom

```
function confirmEnding(str,target){  
    return str.substr(-target.length)===target  
}  
console.log(confirmEnding("priya","a"))
```

```
=====
```

Code 39: To find the largest elements from the 2 dimensional array

```
function largestFromArray(arr){  
    var max=[];  
    for(var i=0; i<arr.length;i++){  
        var tempMax =arr[i][0] //first elements of the 4 internal arrays i,e(1,5,45,89  
        for(var j=0; j<arr[i].length; j++){  
            var currElement = arr[i][j];  
            if(currElement>=tempMax){  
                tempMax = currElement  
            }  
        }  
        max.push(tempMax)  
    }  
    console.log(max)  
    return max;  
}
```

```
largestFromArray([[1,2,3,4],[5,6,7,9],[45,76,2,1],[89,90,87,9]])
```

```
=====
```

Code 40: To find the largest elements from the 2 dimensional array in another way

```
function largestFromArray(arr){  
    var max=[0,0,0,0];  
    for(var i=0; i<arr.length;i++){  
        for(var j=0; j<arr[i].length; j++)  
        {  
            if(arr[i][j]>=max[i]){  
                max[i] = arr[i][j]  
            }  
        }  
    }  
    console.log(max)  
    return max;  
}
```

```
largestFromArray([[1,2,3,4],[5,6,7,9],[45,76,2,1],[89,90,87,9]])
```



```
=====
=====
=====
Code 41: Print string n times using inbuilt function
```

```
function repeatStrinNumTimes(str, num){
  if (num<1) return ""
  return str.repeat(num)
}
console.log(repeatStrinNumTimes("priya",3))
=====
=====
=====
```

```
=====
Code 42: Print string n times in custom way
```

```
function repeatStrinNumTimes(str, num){
  var final="";
  if(num<0) return ""
  for(var i=0; i<num;i++)
  {
    final=final+str
  }
  return final
}
console.log(repeatStrinNumTimes("priya",3))
=====
=====
=====
```

```
=====
Code 43:Print string n times in custom way
```

```
function repeatStrinNumTimes(str, num){
  if(num<0) return ""
  if(num===1) return str
  return str+ repeatStrinNumTimes(str, num-1)
}
console.log(repeatStrinNumTimes("priya",3))
=====
=====
=====
```

```
=====
Code 44: Truncate the string
```

```
function truncateString(str, num){
  if(num<=3) return str.slice(0,num)
  return str.slice(0,num-3)+"..." //retuen only 4 digits thats why subtracted from 3
}
console.log(truncateString("priyabagde",2)) //pr
console.log(truncateString("priyabagde",4)) //p... //retuen only 4 digits
```

```
=====
=====
=====
Code 45: Converting one dimensional array into n dimensional array using slice
```

```
function chunkArrayInGroup(arr, size){
  var group=[]
  while(arr.length>0){
    group.push(arr.slice(0, size))
    arr = arr.slice(size)
  }
  return group
}
console.log (chunkArrayInGroup(['a','b','c','d'],2)) //[["a", "b"], ["c", "d"]]
```

```
=====
=====
=====
Code 46: Converting one dimensional array into n dimensional array using splice
```

```
function chunkArrayInGroup(arr, size){
  var group=[]
  while(arr.length>0){
    group.push(arr.splice(0, size))
  }
  return group
}
console.log (chunkArrayInGroup(['a','b','c','d'],2)) //[["a", "b"], ["c", "d"]]
```

```
=====
=====
=====
Code 47: To find only truthy values
```

```
function removeFalseValue(arr){
  var truth = []
  for (var item of arr){
    if(item){
      truth.push(item)
    }
  }
  return truth
}
console.log(removeFalseValue(["priya", 0 , "", false, null,undefined, "ate", Nan ,9 ]))
//[ "priya","ate",9]
```

Code 49: To find only truthy values using filter

```
function removeFalseValue(arr){
  return arr.filter((item)=>{
    return item})
}
console.log(removeFalseValue(["priya", 0 ,"" , false, null,undefined, "ate", 9 ]))
```

```
=====
=====
=====
```

Code 50: Checking all letters of second words should present in first word, in the same order using include function

```
function characterPresent(arr){
  var first = arr[0].toLowerCase()
  var second = arr[1].toLowerCase()
  for (var letter of second){
    if(!first.includes(letter)){
      return false
    }
  }
  return true
}
console.log(characterPresent(["hello","hey"]))
```

```
=====
=====
=====
```

Code 51: Checking all letters of second words should present in first word, in the same order using indexOf without indexing i.e for-of loop

```
function characterPresent(arr){
  var first = arr[0].toLowerCase()
  var second = arr[1].toLowerCase()
  for (var letter of second){
    if(first.indexOf(letter)== -1){ //-1 means not found in array
      return false
    }
  }
  return true
}
console.log(characterPresent(["hello","he"]))
```

```
-----
```

```
function characterPresent(arr){
  var first = arr[0].toLowerCase()
  var second = arr[1].toLowerCase()
  for (var i=0; i<second.length; i++){
```

```

    if(!first.includes(second[i])){ //-1 means not found in array
        return false
    }
}
return true
}
console.log(characterPresent(["hello", "he"]))
=====
=====
=====

```

Code 52: Checking all letters of second words should present in first word, in the same order using indexOf with indexing

```

function characterPresent(arr){
    var first = arr[0].toLowerCase()
    var second = arr[1].toLowerCase()
    for (var i=0; i<second.length; i++){
        if(first.indexOf(second[i])=== -1){ //-1 means not found in array
            return false
        }
    }
    return true
}
console.log(characterPresent(["hello", "he"]))
=====
=====
=====

```

Code 53: Unique values only from 2 arrays

```

function diffArrayElement(arr1, arr2){
    var result =[]
    for(var i=0; i<arr1.length; i++){
        if(arr2.indexOf(arr1[i]) === -1){
            result.push(arr1[i])
        }
    }
    for(var j=0; j<arr2.length; j++){
        if(arr1.indexOf(arr2[j]) === -1){
            result.push(arr2[j])
        }
    }
    return result
}
console.log(diffArrayElement([1,2,3,4], [2,3,4,5])) //[1,5]
=====

```

```
=====
=====
Code 54: Unique values only from 2 arrays
```

```
function diffArrayElement(arr1, arr2){
var combine = arr1.concat(arr2)
return combine.filter( (num)=>{
  if(arr1.indexOf(num)== -1 || arr2.indexOf(num)== -1 ) return num
})
}
console.log(diffArrayElement([1,2,3,4], [2,3,4,5])) [1,5]
```

```
=====
=====
Code 55: Remove Duplicates from 2 arrays using Set
```

```
function uniquefromArrays(arr1, arr2){
  let arr = [...arr1, ...arr2]
  let unique = [...new Set(arr)];
  return unique
}
console.log(uniquefromArrays([1,2,3,4], [2,3,4,5])) //[1,2,3,4,5]
```

```
=====
=====
code 56: Sum of all numbers from start to end given number
```

```
function sumFromStartToEnd(arr){
  var start = Math.min(arr[0], arr[1])
  var end = Math.max(arr[0], arr[1])
  sum =0
  for(var i= start; i<=end; i++){
    sum+=i
  }
  return sum
}
console.log(sumFromStartToEnd([1,4]))
```

```
=====
=====
code 57: Remove or Delete elements from an array using various ways
```

```
Way 1: Removing Elements from End of a JavaScript Array
```

```
var ar = [1, 2, 3, 4, 5, 6];
ar.length = 4; // set length to remove elements
console.log( ar ); // [1, 2, 3, 4]
```

Way 2: Removing Elements from Beginning of a JavaScript Array

```
var ar = ['zero', 'one', 'two', 'three'];
ar.shift(); // returns "zero"
console.log( ar ); // ["one", "two", "three"]
```

Way 3: Using Splice to Remove Array Elements in JavaScript

```
var list = ["bar", "baz", "foo", "qux"];
list.splice(0, 2); // Starting at index position 0, remove two elements ["bar", "baz"] and
retains ["foo", "qux"].
```

Way 4: Removing Array Items By Value Using Splice

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0];
for( var i = 0; i < arr.length; i++){
    if ( arr[i] === 5) {
        arr.splice(i, 1);
    }
} // [1, 2, 3, 4, 6, 7, 8, 9, 0]
```

OR

```
var arr = [1, 2, 3, 4, 5, 5, 6, 7, 8, 5, 9, 0];
for( var i = 0; i < arr.length; i++){
    if ( arr[i] === 5) {
        arr.splice(i, 1);
        i--;
    }
} // [1, 2, 3, 4, 6, 7, 8, 9, 0]
```

Way 5: Using the Array filter Method to Remove Items By Value

```
var array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0];
var filtered = array.filter(function(value, index, arr){
    return value > 5;
}); //filtered => [6, 7, 8, 9]
```

Way 6: Making a Remove Method

```
function arrayRemove(arr, value) {
    return arr.filter(function(ele){
        return ele !== value;
    });
}
var result = arrayRemove(array, 6); // result = [1, 2, 3, 4, 5, 7, 8, 9, 0]
```

Way 7: Explicitly Remove Array Elements Using the Delete Operator

```

var ar = [1, 2, 3, 4, 5, 6];
delete ar[4]; // delete element with index 4
console.log( ar ); // [1, 2, 3, 4, undefined, 6]

```

Code 58 : Spiral Matrix Printing | Print the elements of a matrix in spiral form

```

var input = [[1, 2, 3, 4],
             [5, 6, 7, 8],
             [9, 10, 11, 12],
             [13, 14, 15, 16]];
function run(input, result) {
    // add the first row to result
    result = result.concat(input.shift());
    console.log("res1", result) //[1, 2, 3, 4] //[1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5, 6, 7]
    console.log("in1", input)  //[5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]] // [[10, 11]]

    // add the last element of each remaining row
    input.forEach(function(rightEnd) {
        result.push(rightEnd.pop());
    });
    console.log("res2", result) //[1, 2, 3, 4, 8, 12, 16] //[1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5, 6, 7,
11]
    console.log("in2", input)  //[5, 6, 7], [9, 10, 11], [13, 14, 15]] // [[10]]

    // add the last row in reverse order
    result = result.concat(input.pop().reverse());
    console.log("res3", result) //[1, 2, 3, 4, 8, 12, 16, 15, 14, 13] //[1, 2, 3, 4, 8, 12, 16, 15, 14, 13,
9, 5, 6, 7, 11, 10]
    console.log("in3", input)  //[5, 6, 7], [9, 10, 11]]

    // add the first element in each remaining row (going upwards)
    var tmp = [];
    input.forEach(function(leftEnd) {
        tmp.push(leftEnd.shift());
    });
    console.log("res4", result) //[1, 2, 3, 4, 8, 12, 16, 15, 14, 13]
    console.log("in4", input)  //[6, 7], [10, 11]]

    result = result.concat(tmp.reverse());
    console.log("temp", tmp)    //[9, 5]
    console.log("res5", result) //[1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5]
    console.log("in5", input)  //[6, 7], [10, 11]]

```

```

    //again start the function
    return run(input, result);
}
console.log('result', run(input, [])); // [1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10]
=====
=====
=====

```

Code 59: Currying function i.e sum of multiple argument functions //inner function can access outer function variables but outer functions can't able to access inner function.

```

function sum(a){
  return function sum(b){
    return function sum(c){
      return function sum(d){
        return a+b+c+d;
      }
    }
  }
}
console.log(sum(1)(2)(3)(4))

```

OR

```

const sum = (a) => (b) => (c) => (d) => a+b+c+d // using ES6
console.log(sum(1)(2)(3)(4))
=====
=====
=====

```

Code 60: Find SUM, PRODUCT AND AVERAGE of the numbers //accumulation means collection

```

let arr=[1,2,3,4,5]
let sum = arr.reduce((accum, curr) =>{
  return accum + curr;
})
console.log(sum) //15

```

OR

```

let sum = arr.reduce((accum, curr) =>{
  return accum + curr;
},5) // can set initial value as 5 also
console.log(sum) //20

```



```

let product = arr.reduce((accum, curr) =>{
  return accum * curr;
})
console.log(product)//120

```

```

let average = arr.reduce((accum, curr, index, array) =>{
  let total = accum + curr;
  if(index === array.length-1){
    return total/array.length
  }
  return total
})
console.log(average)//3

```

```

=====
=====
=====

```

Code 61: Convert 2D/3D array into 1D using reduce function and inbuilt function i.e flat

```

const arr = [
  ['a','b'],
  ['c','d'],
  ['e','f'],
]
const flatArr = arr.reduce((accum, curr)=>{
  return accum.concat(curr)
})
console.log(flatArr) //["a", "b", "c", "d", "e", "f"]

```

OR

```

const arr = [
  ['a','b'],
  ['c','d'],
  ['e',['f','g']],
]
console.log(arr.flat(2)) //["a", "b", "c", "d", "e", "f"] //bydefault 1 hota h as a argument

```

OR

```

const arr = [
  ['a','b'],
  ['c','d'],
  ['e',['f',['g','h']]],
]

```

```

]
console.log(arr.flat(3)) //["a", "b", "c", "d", "e", "f", "g", "h"]
=====
=====
=====

code 62: Reverse of a nuber using converting into string
function reverseNumber(input){
return(
    parseFloat(input.toString().split("").reverse().join(""))*Math.sign(input)
)
}
console.log(reverseNumber(123)) //321
=====
=====
=====

code 63: Reverse of a nuber
function reverseNumber(input){
var result=0;
while(input!=0){ //123 //12 //1
    result = result *10; //0*10=0 //3*10=30 // 32*10 =320
    result = result + (input%10) //give reminder // 0+3=3 // 30+2=32 //320+1=321
    input = Math.floor(input/10) //12 //1
    // console.log("in", input)
}

    return result
}
console.log(reverseNumber(123)) //321
=====
=====
=====

code 64: Check Armstrong Number
function CheckArmstrongNum(num){ //153
    var temp = num;
    var result =0;
    var a;
    while(temp>0){ //153 //15 //1
        a= temp%10; //3 //5 //1
        temp= parseInt(temp/10) //15 // 1
        result= result+a*a*a //0+3*3*3 // 27+ 5*5*5 // 27+ 5*5*5 +1*1*1
    }
    if(result==num){
        return true
    }
}

```

```

    }
    return false
}
console.log(CheckArmstrongNum(153)) //3*3*3 + 5*5*5 + 1*1*1
=====
=====
=====

```

code 65: To find the closest number in an array

```

const needle = 5;
const numbers = [1, 10, 7, 2, 4, 9];
numbers.sort((a, b) => {
    return Math.abs(needle - a) - Math.abs(needle - b);
})
console.log(numbers[0]);
=====
=====
=====

```

code 66: To find the second largest number

```

function secondLargestNum(arr){
    return arr.sort((a, b)=> b - a )[1]
}
console.log(secondLargestNum(['1', '2', '3', '4', '9']))
=====
=====
=====

```

code 67: To check whether particular word/number present in sentence or not using inbuilt function

```

function wordInSentence(str){
    return str.includes("world"); //true
}
console.log(wordInSentence("Hello world, welcome to the universe."))
OR
var nums =[0,1,3,5,6,7,8,9,7]
console.log(nums.includes(9)) //true
OR
var item=3
console.log(nums.some(x => x === item)) //true
=====
=====
=====

```

code 68: To check whether particular word/number present in sentence or not using custom function

```

function checkValueExist(arr, item){

```

```

var status = "Not Exist"
for(var i=0; i<arr.length; i++){
  if(arr[i]===item){
    status = "Exist"
    break;
  }
}
return status
}
console.log(checkValueExist(['priya', 'riya', 'supriya'], 'priya'))
=====
=====
=====

```

code 69: To check wheather property exist or not in object

```

let student ={
  name : "priya",
  age: 20
}
console.log('name' in student)
OR
console.log(student.hasOwnProperty('name'))
=====
=====
=====

```

code 70: To dlete the property of an object

```

let student ={
  name : "priya",
  age: 20,
  city: "pune"
}
delete student.age;
console.log(student)
OR
delete student['name']
console.log(student)
=====
=====
=====

```

code 71: To find the length of the array in custom way

```

function findLength(arr){
  var len =0;
  while(arr[len]!==undefined){
    len++
  }
}

```

```
}
return len;
}
console.log(findLength([50,60,70,80,90]))
```

OR

```
function findLength(arr){
    return arr.length;
}
console.log(findLength([50,60,70,80,90]))
```

```
=====
=====
=====
```

code 72: Star Pattern

```
for(var i=1; i<=5;i++){ //use to create new row
    for(var j=i; j<=5; j++){ //use to add in existing row
        document.write("**")
    }
    document.write("<br/>")
}
```

**

*

```
=====
=====
=====
```

code 73: Star Pattern

```
for(var i=1; i<=5;i++){ //use to create new row
    for(var j=1; j<=5; j++){ //use to add in existing row
        document.write("**")
    }
    document.write("<br/>")
}
```

```
=====
=====
=====
```

code 74: Star Pattern

```

for(var i=1; i<=5;i++){ //use to create new row
  for(var j=i; j<=5; j++){ //use to add in existing row
    document.write(i)
  }
  document.write("<br/>")
}

```

```

11111
2222
333
44
5

```

```

=====
=====
=====

```

code 75: Star Pattern

```

for(var i=1; i<=5;i++){ //use to create new row
  for(var j=i; j<=5; j++){ //use to add in existing row
    document.write(j)
  }
  document.write("<br/>")
}

```

```

12345
2345
345
45
5

```

```

=====
=====
=====

```

code 76: Star Pattern

```

for(var i=1; i<=5;i++){ //use to create new row
  for(var j=1; j<=i; j++){ //use to add in existing row
    document.write("*")
  }
  document.write("<br/>")
}

```

```

*
**
***
****
*****

```

```

=====
=====

```

=====

code 77: To find the square root

```
var num = [4, 9, 16, 25, 36]
```

```
var result = num.map(Math.sqrt)
```

```
console.log(result) //[2,3,4,5,6]
```

=====

=====

=====

code 78: Make alternate character to upper case

```
function alternateText(str){
```

```
  var char = str.toLowerCase().split("")
```

```
    for(var i=0; i < char.length; i=i+2){
```

```
      char[i]=char[i].toUpperCase()
```

```
    }
```

```
  return char.join("")
```

```
}
```

```
console.log(alternateText("Priya Bagde")) //"PrlyA BaGdE"
```

OR

```
let alt = "Priya Bagde"
```

```
alt = alt.split("")
```

```
  .map((letter,index)=>(index%2)==0 ? letter.toUpperCase(): letter.toLowerCase())
```

```
  .join("")
```

```
console.log(alt) //"PrlyA BaGdE"
```

=====

=====

=====

code 79: To find the negative values in an array or 2D Array

```
function countNegative(arr){
```

```
  let count = 0;
```

```
  for(let i=0;i<arr.length; i++){
```

```
    for(let j=0; j<arr[i].length; j++){
```

```
      if(arr[i][j]<0){
```

```
        count++
```

```
      }
```

```
    }
```

```
  }
```

```
  return count;
```

```
}
```

```
console.log(countNegative([[1,-1],[-1,-1]]))
```

=====

=====

=====

code 80: Find first repeating character with its index from an array

```

function firstRepeatingIndex(arr){
  let count = {};
  for(let i=0;i<arr.length; i++){
    if(count[arr[i]])
    {
      console.log("character", arr[i])
      console.log("index", count[arr[i]])
      return count[arr[i]] //if exist
    }
    else
    {
      count[arr[i]]=i //if not exist keep at count
    }
    console.log("count", count)
  }
  return count
}
firstRepeatingIndex([1,0,2,3,4,4,5,7,7])

```

```

=====
=====
=====

```

code 81: To find all the subsets of the set

```

function generateSubsets (arr) { //[1,2]
  let subsets = [];
  for (const item of arr)
  {
    const tempSubsets = [...subsets];
    console.log("tempSubsets",tempSubsets) //[[]][[1]]
    for (const currSubset of tempSubsets)
    {
      subsets.push([...currSubset, item]);
      console.log("subsets",subsets) //not came//[1], [1,2]]
    }
    subsets.push([item]);
    console.log("subsets1",subsets) //[1][[1], [1,2],[2]]
  }
  subsets.push([]);
  console.log("subsets2",subsets) //[1], [1, 2], [2], []
  return subsets;
}
generateSubsets([1, 2]);
OR

```



```

function generateSubsets (arr) {
  let subsets = [];
  for (const item of arr) //[1,2]
  {
    const tempSubsets = [...subsets]; //[[]][[1]]
    for (const currSubset of tempSubsets)
    {
      subsets.push([...currSubset, item]); //not came//[1], [1,2]]
    }
    subsets.push([item]); //[1]//[1], [1,2],[2]]
  }
  subsets.push([]); //[1], [1, 2], [2], []
  return subsets;
}
generateSubsets([1, 2]);
OR
function findAllSubsetsofGivenSet(arr)
{
  var result= arr.reduce((subsets, value) => subsets.concat(subsets.map(set => [value,...set])),
    [[]]) //[] is used to pass initial value
  return result
}
console.log(findAllSubsetsofGivenSet([8,9]));

```

```

function findAllSubsets(arr){
  var result = []
  for(var item of arr){
    let tempSub = [...result]
    for(var curr of tempSub){
      result.push([...curr, item])
    }
    result.push([item])
  }
  result.push([])
  return result
}

```

```

console.log(findAllSubsets([1,2]))

```

```

=====
=====
=====

```

Code 82: To find the maximum repetition of the character in a string

```

function maxRepeating(str)

```

```

{
  let count = 0;
  let character = str[0];
  for (let i=0; i<str.length; i++)
  {
    let tempCount = 1;
    for (let j=i+1; j<str.length; j++)
    {
      if (str[i] == str[j]) //if a is equal to a
        tempCount++; //use to find out the counts of character i.e a
    }
    if (tempCount > count)
    {
      count = tempCount;
      character = str[i];
    }
  }
  console.log(count, character)
  return character;
}
maxRepeating("aaaabbaacccccccccccccccde");
=====
=====
=====

```

Code 83: To find all the missing numbers from an array

```

function MissingElements(arr)
{
  for(let i = 0; i < arr.length; i++)
  {
    if (arr[i] - i != arr[0]) //1-0==1 //2-1==1 //6-2!=1 //checking for consecutive numbers
    {
      while (arr[0] < arr[i] - i) //1<4 //2<4 //3<4 //finding missing numbers
      {
        console.log(i + arr[0]); //2+1 //3+1 //3+1
        arr[0]++; //2 //3 //4
      }
    }
  }
}
MissingElements([1,2,6]); //3,4,5

```

```

function MissingElements(arr)
{

```

```

for(let i = 0; i < arr.length; i++)
{
    if (arr[0] !== arr[i] - i)
    {
        while (arr[0] < arr[i] - i)
        {
            console.log(arr[0]+i);
            arr[0]++;
        }
    }
}

```

MissingElements([1,2,6])

```

=====
=====
=====

```

Code 84: Adding an elements to the array when elements are consecutive numbers

```

const as = [1,2,3,4];
for (let index = 5; index <= 10; ++index) {
    as.push(index);
}
console.log(as); //[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

```

```

=====
=====
=====

```

Code 85: Create a new array by adding one to each elements of the existing array

```

function plusOne(arr){
    var output=[]
    for (let i = 0; i < arr.length; ++i) {
        output.push(arr[i]+1);
    }
    return output
}
console.log(plusOne([1,2,3,4]));

```

```

=====
=====
=====

```

Code 86: To find kth smallest or largest element in an array

```

function findKthSmallestOrLargest(arr, num) {
    arr.sort(function(a, b) { return a - b});
    console.log(arr)
    console.log("kth smallest", arr[num- 1]) //kth smallest
    console.log("kth largest", arr[arr.length-num]) //kth smallest
}

```

```

};
console.log(findKthSmallestOrLargest([2,1,4,3,6,5,7], 3)); //kth is 3rd //3,5
=====
=====
=====

```

Code 87: sort by frequency of the letters

```

function frequencySort(str) {
  let map = {}
  for (const letter of str) {
    map[letter] = (map[letter] || 0) + 1; //to count the occurrence
  };
  console.log(map) //{a: 2,A: 2,b: 3,B: 3,c: 1,C: 1}
  let res = "";
  let sorted = Object.keys(map).sort((a, b) => map[b] - map[a])
  console.log("sorted", sorted)// ["b", "B", "a", "A", "c", "C"]
  for (let letter of sorted) {
    for (let count = 0; count < map[letter]; count++) {
      res += letter
      console.log(res)
    }
  }
  return res;
};
console.log(frequencySort("cCaaAAbbbbBBB")); //"bbbBBBaaAAcC"

```

```

function frequencySort(str) {
  let map = {}, res = "", sortedArr;
  for (const letter of str)map[letter]=(map[letter] || 0) + 1;
  sortedArr = Object.keys(map).sort((a, b) => map[b] - map[a]);
  for (let letter of sortedArr) {
    for (let count = 0; count < map[letter]; count++) {
      res += letter
    }
  }
  return res;
};
console.log(frequencySort("cCaaAAbbbbBBB"));

```

Code 88: To find the OCCURANCE of the character

```

function frequencySort(str) {

```

```

let map = {}
for (var i=0; i<str.length; i++)
{
    map[str[i]] = map[str[i]] ? map[str[i]]+1 : 1; //Adding an element to the object, if already
present then incrementing by 1
}
console.log(map)////{"c":1, "C":1, "a":2, "A":2, "b":3, "B":3}
};
frequencySort("cCaaAAbbbbBBB");
OR
function frequencySortArr(arr) {
    let map = {}
    arr.forEach((element)=>{map[element] = map[element]+1 || 1 }) // will get occurrence of the
number
    return [...arr].sort((a,b)=> map[b]-map[a])
};
console.log(frequencySortArr([2,5,67,89,2,3,4,4,4])); // [4,4,4,2,2,5,67,89,3]
=====
=====
=====

```

Code 89: Permutation // Need to debug

```

let perm= (str, result)=> {
    if(str.length==0){console.log("result", result)} //let //lte //elt //etl //tle //tel

    for(var i=0; i<str.length; i++){
        let rest= str.substring(0,i)+ str.substring(i+1)
        // console.log("rest", rest) //et//t//"" //e//"" //lt//t//"" //l//"" //le//e//"" //l//""
        console.log("finalresult",result+str[i]) //l//le//let //lt//lte //e//el//elt //et//etl //t//tl//tle //te//tel
        perm(rest, result+str[i])
    }
}
perm("let","");
// "result" "let"
// "result" "lte"
// "result" "elt"
// "result" "etl"
// "result" "tle"
// "result" "tel"
=====
=====
=====

```

Code 90: To find the power of x

```

var r = 1, i = 1;

```

```

var b = 2; e = 3;
while(i <= e) //1<3//2<3//3=3
{
    r *= b; //1*2//2*2//4*2
    i++;
}
console.log(r) //8

```

OR

```

let number = 2;
let exponent = 3;
console.log( number ** exponent);
console.log( Math.pow(number, exponent));

```

```

=====
=====
=====

```

Code 91: To find even and odd number by user input

```

const number = prompt("Enter a number: ");
if(number % 2 == 0) {
    console.log("The number is even.");
}
else {
    console.log("The number is odd.");
}

```

```

=====
=====
=====

```

Code 92: Grouping of an Anagram

```

let collectAnagrams = (words) => {
    let anagrams = {}
    let collectedAnagrams = []
    for (let word of words)
    {
        let sortedWord = word.split("").sort().join("") //arrange every word in alphabetical order
        anagrams[sortedWord] = anagrams[sortedWord] || [] //console.log(".",anagrams) //creating
keys
        anagrams[sortedWord].push(word) // assigning exact values to keys //console.log("..",
anagrams)
    }
    console.log(anagrams)
    for (let item in anagrams)
    {
        collectedAnagrams.push(anagrams[item]) // add their values as subarrays of the
collectedAnagrams array
    }
}

```

```

    }
    return collectedAnagrams
}
console.log(collectAnagrams(['bag', 'gab', 'foo', 'abg', 'oof', 'ofo'])) //["bag", "gab", "abg"], ["foo",
"oof", "ofo"]

```

```

=====
=====
=====

```

Code 93: Sort an array of an element by parity means even then odd elements

```

function sortByParity(arr){
  let even =[]
  let odd =[]
  let result=[]
  for(let i=0; i<arr.length; i++){
    if(arr[i]%2 ===0 ) even.push(arr[i])
    else odd.push(arr[i])
  }
  result = even.concat(odd)
  return result
}
console.log(sortByParity([1,2,3,4,5,6,7,8,9]))

```

```

=====
=====
=====

```

Code 94: Move all the zeroes at the end of an elements

```

var array = [1,0,2,0,0,9,0,6,7];
array.sort(function(a, b) {
  if(a==0 && b!=0)
    return 1;
  else if (a!=0 && b==0)
    return -1;
  else
    return 0;
});
console.log(array)
OR
var moveZeroes = function(arr) {
  for ( var i = 0; i < arr.length-1; i++)
  {
    if(arr[i] === 0) { //if place x here then move x last to the array
      var temp = arr.splice(i, 1);
      console.log(temp[0])
      arr.push(temp[0]);
    }
  }
}

```

```

    }
  }
  return arr;
};
console.log(moveZeroes([1,0,4,8,6,0,8,3,39,0])) //[1,4,8,6,8,3,39,0,0,0]

```

OR

```

var moveZeroes = function(arr) {
  for ( var i = 0; i < arr.length-1; i++)
  {
    if(arr[i] === 0) {
      arr.splice(i, 1);
      arr.push(0);
    }
  }
  return arr;
};
console.log(moveZeroes([1,0,4,8,6,0,8,3,39,0]))

```

```

=====
=====
=====

```

Code 95: Print consecutive numbers

```

function range(num)
{
  var result =[]
  for(var i =0; i<num; i++)
  {
    result.push(i)
  }
  return result
};
console.log(range(10)); //[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

```

```

=====
=====
=====

```

Code 96: 4 Ways to empty an array

```

var arrayList = ['a', 'b', 'c', 'd', 'e', 'f'];
console.log("1", arrayList = [])
console.log("2", arrayList.length = 0)
console.log("3", arrayList.splice(0, arrayList.length))
var result= function(){while(arrayList.length) {
  arrayList.pop();
}}
console.log("4", arrayList)

```



```
=====
```

```
=====
```

```
=====
```

Code 97: Create a function to calculate the sum of all the numbers in a jagged array

```
function sumArray(ar)
{
    var sum = 0;
    for(var el of ar)
    {
        if (Array.isArray(el))
        {
            el = sumArray(el); //recursion
        }
        sum += el;
    }
    return sum;
}
console.log(sumArray([1, 2, [3, [4], [5, 6]], [7]])) //28
```

```
=====
```

```
=====
```

```
=====
```

Code 98: To check weather perfect number or not

```
function is_perfect(number)
{
    var temp = 0;
    for(var i=1;i<=number/2;i++)
    {
        if(number%i === 0)
        {
            console.log(i) //1,2,4,7,14
            temp += i;
        }
    }
    if(temp === number && temp !== 0)
    {
        console.log("It is a perfect number.");
    }
    else
    {
        console.log("It is not a perfect number.");
    }
}
is_perfect(28);
```

```
=====
=====
=====
Code 99: Number of days between 2 dates calculation
```

```
date1 = "2020-01-01", date2 = "2020-01-30"
```

```
function daysBetweenDates (date1, date2) {
  const days = (new Date(date2) - new Date(date1)) / (1000 * 60 * 60 * 24)
  return days
}
console.log(daysBetweenDates(date1,date2))
=====
=====
=====
```

```
=====
Code 100: To find todays date
```

```
var today = new Date();
var date = today.getFullYear()+'-'+(today.getMonth()+1)+'-'+today.getDate();
console.log(date)
=====
=====
=====
```

```
=====
Code 100: String Compression (Microsoft, Amazon etc)
```

```
function stringCompression (str) {
  if (str.length ==0) {
    console.log('Please enter valid string. ');
    return;
  }
  var output = "";
  var count = 0;
  for (var i = 0; i < str.length; i++)
  {
    count++;
    if (str[i] != str[i+1]) //if a is not equal to b
    {
      output += str[i] + count; //a+4
      count = 0; //for b it will start from zero
    }
  }
  console.log(output);
}
stringCompression(""); //Please enter valid string.
stringCompression('aaaa'); //a4
stringCompression('aaaabbc'); //a4b2c1
stringCompression('aaaabbcaabb'); //a4b2c1a2b2
```

```

-----
function stringCompression (str) {
  var output = "";
  var count = 0;
  for (var i = 0; i < str.length; i++){
    count++;
    if (str[i] != str[i+1]) {
      output += str[i] + count;
      count = 0;
    }
  }
  console.log(output);
}
stringCompression('aaaab');

```

Code 102: Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.

A subtree of a binary tree tree is a tree that consists of a node in tree and all of this node's descendants. The tree tree could also be considered as a subtree of itself.

Given tree s:

```

  3
 / \
4   5
/ \
1  2

```

Given tree t:

```

  4
 / \
1  2

```

```

-----
class Node {
  constructor(val) {
    this.data = val;
    this.left = null;
    this.right = null;
  }
}
var root1,root2;
root1 = new Node(26); // TREE 1
root1.right = new Node(3);
root1.right.right = new Node(3);
root1.left = new Node(10);

```

```

    root1.left.left = new Node(4);
    root1.left.left.right = new Node(30);
    root1.left.right = new Node(6);
    root2 = new Node(10);           // TREE 2
    root2.right = new Node(6);
    root2.left = new Node(4);
    root2.left.right = new Node(30);
function areIdentical(root1, root2) //to check for same
{
    if (root1 == null && root2 == null)
        return true;
    if (root1 == null || root2 == null)
        return false;
    return (root1.data == root2.data && areIdentical(root1.left, root2.left) &&
areIdentical(root1.right, root2.right));
}
function isSubtree(T, S) //main function
{
    if (S == null)
        return true;
    if (T == null)
        return false;
    if (areIdentical(T, S))
        return true;
    return isSubtree(T.left, S) || isSubtree(T.right, S);
}
console.log(isSubtree(root1, root2))
=====
=====
=====

```

Find triplets whose sum is zero

```

function findTriplets(arr, n) {
    arr.sort();
    for (var i = 0; i < arr.length; i++) {
        var j = i + 1,
            k = arr.length - 1;
        while (j < k) {
            if (arr[i] + arr[j] + arr[k] < n) {
                j++;
            } else if (arr[i] + arr[j] + arr[k] > n) {
                k--;
            } else {
                console.log(arr[i] + "," + arr[j] + "," + arr[k]);
            }
        }
    }
}

```

```

        j++;
        k--;
    }
}
}
return true;
}
var arr = [-1, -4, -9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
findTriplets(arr, 0);

```

```

function findTriplets(arr, n) {
    arr.sort((a, b) => a - b);
    for (let i = 0; i < arr.length - 2; i++) {
        for (let j = i + 1; j < arr.length - 1; j++) {
            for (let k = j + 1; k < arr.length; k++) {
                if (arr[i] + arr[j] + arr[k] === 0) {
                    console.log(`${arr[i]}, ${arr[j]}, ${arr[k]}`);
                }
            }
        }
    }
}
let arr = [1, 2, 5, 3, -2, 0, 1, -1, 5, 6, -2, -1];
findTriplets(arr, 0);

```

Convert Array into object:-

```

const arr = ["John", "Peter", "Sally", "Jane"];
const updatedArr={...arr};
console.log(updatedArr)//{0: 'John', 1: 'Peter', 2: 'Sally', 3: 'Jane'}

```

3 main use-cases of #map() function:-

- 1 . Used for rendering a list of data to the Dom in React
2. Used to modify an element in array depending on some requirement, which means calling a function on each item in Array
3. Used to convert String to Array

103. Find common elements:

```
function commonElements(a, b){
  return a.filter((item) => b.includes(item))
}
A = [1,2,3,4,5];
B = [4, 5, 6, 7, 8];
console.log(commonElements(A, B)); //[4,5]
```

```
=====
=====
=====
```

```
=====
=====
=====
```

```
=====
=====
=====
```

```
=====
=====
=====
```

JAVASCRIPTS Inbuilt Functions:

Code : JAVASCRIPT substr concept

```
var sentence = "I'm priya and having sounds knowledge."
console.log(sentence.substr(0,5)) //(startigIndex, NoOfCharatersWants-->take 1 less)//I'm p
console.log(sentence.substr(2,5)) //m pri
console.log(sentence.substr(2)) //m priya and having sounds knowledge.
console.log(sentence.substr(4)) //priya and having sounds knowledge.
console.log(sentence.substr(-4)) //Negative goes From ending of the string //dge.
console.log(sentence.substr(-5)) //edge.
```

```
=====
=====
=====
```

Code : JAVASCRIPT slice concept i.e, it doesn't change the original array

```
var sentence = "I'm priya and having sounds knowledge."
console.log(sentence.slice(0,5)) //"I'm p"
console.log(sentence.slice(2,5)) //"m p"
console.log(sentence.slice(2)) //"m priya and having sounds knowledge."
console.log(sentence.slice(4)) //"priya and having sounds knowledge."
```

```
console.log(sentence.slice(-4)) //"dge."
console.log(sentence.slice(-5)) //"edge."
```

```
var sentence =['a','b','c','d']
console.log(sentence.slice(0,2)) //[ 'a','b' ]
console.log(sentence) //[ 'a','b','c','d' ]
```

Code : JAVASCRIPT splice concept i.e, it changes the original array

```
var sentence =['a','b','c','d']
console.log(sentence.splice(0,2)) //[ 'a','b' ]
console.log(sentence) //[ 'c','d' ]
```

Code: JAVASCRIPT indexOf concept

```
var greeting = "Hello"
console.log(greeting.indexOf("e")) //1
```

Code: JAVASCRIPT split concept

```
var name ="Priya Bagde"
console.log(name.split("")) //[ "P","r","i","y","a"," ","B","a","g","d","e" ]
console.log(name.split(" ")) //[ "Priya","Bagde" ]
```

Code: JAVASCRIPT join concept

```
var arr = ['a','b','c','d','e']
console.log(arr.join()) //"a,b,c,d,e"
```

Code: JAVASCRIPT join concept

```
var arr = ['a','b','c','d','e']
for(var i=0; i<arr.length; i++){
  console.log(arr[i])
```

```
}
"a"
"b"
"c"
```

"d"

"e"

```
=====
=====
=====
```

Advance Interview Concepts

1. Closures-

A closure is the combination of a function and the lexical environment within which that function was declared.

OR

When inner function can have access to the outer function variables and parameter.

The return statement does not execute the inner function - function is only executed only when followed by ()parathesis, but rather than returns the entire body of the function.

Uses/advantages of closures:

- event handlers
- callback functions
- Encapsulation: can store data in separate store
- Object data privacy
- Module Design Pattern
- Currying
- Functions like once
- memoize
- setTimeouts
- Iterators
- maintaining state in async world

Disadvantages of closures:

1. Creating function inside a function leads to duplicate in memory and cause slowing down the application means use only when required privacy.
2. As long as the clousers are active, the memory can't be garbage collected means if we are using clousers in ten places then untill all the 10 process complete it hold the memory and can overcome to set closure to Null.

```
const outerFunction =(a)=>{
  let b=10;
  const innerFunction =()=>{
    let sum = a+b;
    console.log(sum)
  }
}
```



```

    }
    innerFunction()
  }
  outerFunction(5)// 15

```

```

const outerFunction =(a)=>{
  let b=10;
  const innerFunction =()=>>{
    let sum = a+b;
    console.log(sum)
  }
  return innerFunction
}
outerFunction(5) //output :
()=>{
  let sum = a+b;
  console.log(sum)
}

```

```

const outerFunction =(a)=>{
  let b=10;
  const innerFunction =()=>>{
    let sum = a+b;
    console.log(sum)
  }
  return innerFunction
}
let inner = outerFunction(5)
console.log(inner)
inner() //15

```

```

=====
=====
=====

```

Prototype:

Prototype allow to easily define methods to all the instances of object. It stored in the memory once but every object instances can access it.

1. Javascript is a prototype based language, so, whenever we are creating a function using javascript, javascript engine adds a prototype property inside a function, Prototype property is basically an object (also known as Prototype object), where we can attach methods and properties in a prototype object, which enables all the other objects to inherit these methods and properties.
2. We are creating prototype in constructor function. All the intances of objects can able to

access properties and methods from constructor function.

3. The prototype is an object that is associated with every functions and objects by default in JavaScript, where function's prototype property is accessible and modifiable and object's prototype property (aka attribute) is not visible.

4. object's prototype property is invisible. Use `Object.getPrototypeOf(obj)` method instead of `__proto__` to access prototype object.

5. prototype is useful in keeping only one copy of functions for all the objects (instances).

6. An Object has a prototype. A prototype is also an object. Hence Even it may have its own prototype object. This is referred to as prototype chain.

<A>Several Types:

1. `Object.prototype`- It is a prototype OBJECT of object(construction function where it will inherit all properties of `Object.prototype`).

Prototype Object of `Object.prototype` is NULL.

2. `Array.prototype`-Prototype Object of `Array.prototype` is `Object.prototype` and Prototype Object of `Object.prototype` is NULL.

3. `Function.prototype`

4. Example-

```
var person = function(name){
    this.name = name;
}
person.prototype.age = 21;
var piya = new person("Piya");
var priya = new person("Priya");
console.log(piya.age) //21
console.log(priya.age) //21
```

Purpose/Use of prototype:

1) to find properties and methods of an object

2) to implement inheritance in JavaScript

<C>Difference between Prototype and `__proto__`:

1. In reality, the only true difference between prototype and `__proto__` is that the former is a property of a class constructor,

while the latter is a property of a class instance.

2. Instances have `__proto__`, classes have prototype.

3. Instances of a constructor function use `__proto__` to access the prototype property of its constructor function.

4. `__proto__` is invisible property of an object. It returns prototype object of a function to which it links to.

5. `__proto__` is Deprecated.

6. Example:

```
function Person(name){
```

```
    this.name = name
};
var eve = new Person("Eve");
eve.__proto__ == Person.prototype //true
eve.prototype //undefined
```

7. Example:

```
function Person() {
    this.name = 'John'
}
// adding property
Person.prototype.name = 'Peter';
Person.prototype.age = 23
const person1 = new Person();
console.log(person1.name); // John
console.log(person1.age); // 23
```

```
=====
=====
=====
```

CSS Positions:

1. Static: HTML elements are positioned static by default. Static positioned elements are not affected by the top, bottom, left, and right properties.

Impact of margin or padding. Object can't move. It is always positioned according to the normal flow of the page.

2. Relative: Object can move. It is positioned relative to its normal position. If want gap from its actual placed position then use it. It work with left, right, top, bottom properties.

3. Fixed: Not allow to scroll up or down. is positioned relative to the viewport, which means it always stays in the same place even if the page is scrolled.

The top, right, bottom, and left properties are used to position the element. The element is positioned relative to the browser window

4. Absolute: it is work with relative i.e, w.r.t parent. It is positioned relative to the nearest positioned ancestor (instead of positioned relative to the viewport, like fixed). if an absolute positioned element has no positioned ancestors, it uses the document body, and moves along with page scrolling.

5. Sticky: An element with position: sticky; is positioned based on the user's scroll position. Internet Explorer does not support sticky positioning.

You must also specify at least one of top, right, bottom or left for sticky positioning to work. Use for to create menu.

```
=====
=====
```

=====

Time based Event:

SetTimeout:

1. allows us to run a FUNCTION ONCE, after the interval of time
2. setTimeout() executes the passed function after given time. The id_value returned by setTimeout() function is stored in a variable and it's passed into the clearTimeout() function to clear the timer.
3. Syntax- let timerId = setTimeout(function, milliseconds, [arg1], [arg2], ...)
4. Don't make a mistake by adding brackets () after the function otherwise gives undefined and nothing will be scheduled.
5. Example-
let timerId = function sayHi(phrase, who) {
 console.log(phrase + who);
}
setTimeout(sayHi, 1000, "Hello", "John");

SetInterval:

1. allows us to run a function repeatedly, starting after the interval of time, then repeating continuously at that interval.
2. setInterval() executes the passed function for the given time interval. The number id value returned by setInterval() function is stored in a variable and it's passed into the clearInterval() function to clear the interval.
3. Syntax-
4. Example
let timerId = function sayHi(phrase, who) {
 console.log(phrase + who);
}
setInterval(sayHi, 1000, "Hello", "John");

ClearTimeout:

1. This method is used to cancel a setTimeout(). Inside the method you have to reference the timeoutID.
clearTimeout(timerId)

ClearInterval:

1. This method is used to cancel a setInterval(). Inside the method you have to reference the intervalID.
clearInterval(timerId)

=====

=====

=====

Debouncing and Throttling in JavaScript: using in search box, scrolling or resize the window size

1. Create impact on performance of your website, but also prevent unnecessary API calls and load on the server.
2. Debouncing and throttling techniques are used to limit the number of times a function can execute. ke button click, mouse move, search bar, window scrolling and window resize allow the user to decide when to execute.
3. The main difference between throttling and debouncing is that throttling executes the function at a regular interval, while debouncing executes the function only after some cooling period.
4. Example: If you're scrolling, throttle will slowly call your function while you scroll (every X milliseconds). Debounce will wait until after you're done scrolling to call your function.

Throttling-

Throttling is a technique in which, no matter how many times the user fires the event, the attached function will be executed only once in a given time interval.

Debouncing-

No matter how many times the user fires the event, the attached function will be executed only after the specified time once the user stops firing the event.

The Debounce technique allow us to "group" multiple sequential calls in a single one.

```
var debounced_version = _.debounce(doSomething, 200);
```

```
$(window).on('scroll', debounced_version);
```

```
debounced_version.cancel();
```

```
=====
=====
=====
```

CALL, APPLY and BIND method: These methods allow us to write a function once and invoke it in a different context. They all attach this into a function (or object) and the difference is in the function invocation. Call and apply are pretty interchangeable. Just decide whether it's easier to send in an array or a comma separated list of arguments. I always remember which one is which by remembering that Call is for comma (separated list) and Apply is for Array. Bind is a bit different.

It returns a new function. Call and Apply execute the current function immediately. The main concept behind all this methods is Function burrowing.

CALL:

1. It is predefined javascript method.
2. An object can use a method belonging to another object.
3. Call invokes the function and allows you to pass in arguments one by one.

APPLY:

1. Apply invokes the function and allows you to pass in arguments as an array.

BIND:

1. We can bind an object to a common function, so that the function gives different results when

its need.

2. It takes an object as an first argument and creates a new function.

Example1:

```
const people={
  fullName: function(){
    return this.firstName+" "+this.lastName;
  }
}
const person1={
  firstName: "Priya",
  lastName:"Bagde"
}
console.log(people.fullName.call(person1)); //Priya Bagde
console.log(people.fullName.apply(person1)); //Priya Bagde
let bound = people.fullName.bind(person1)
console.log(bound()) //Priya Bagde
```

Example2:

```
const obj = {name:"Priya"}
let greeting = function(a,b){
  return a+" "+this.name+" "+b;
}
console.log(greeting.call(obj, "Hello", "How are you?"));
console.log(greeting.apply(obj, ["Hello", "How are you?"]));
let test=greeting.bind(obj);
console.log(test("Hello", "How are you?"))
```

```
=====
=====
=====
```

Hoisting:

1. To move all of the variable and function declarations at the top of the current scope.
2. Hoisting is JavaScript's default behavior of moving declarations to the top.
3. A variable can be used before it has been declared.
4. Note: JavaScript only hoists declarations, not the initializations.
5. JavaScript allocates memory for all variables and functions defined in the program before execution.
6. Due to the concept of hoisting in JavaScript, we can call a function even before we define the function definition in our program's code.
7. Variables defined with let and const are hoisted to the top of the block, but not initialized. Let and const are also hoisted but we cant access them until they are assigned because they are in Temporal dead zone. To avoid Temporal deadzone we need to declare let and const to the top of our program!

***when const gets hoisted it will be defined undefined, and further processing will get assigned the value, this violated the meaning of const, this is the fundamental reason why const doesn't get hoisted and let just got tagged along with this feature.

-- Can able to access variable and function before initialise. Can able to access it without any error.

-- console.log(getName); without parenthesis -- print the body of function.

-- if we remove var x=7 then it will get as reference error: x is not defined because x is not present at all and we are trying to access it.

-- with arrow and function expression

-- callstack

-- console.log(greetingName) //print the body of a function

JavaScript Hoisting refers to the process whereby the interpreter appears to move the declaration of functions, variables or classes to the top of their scope, before execution of the code.

This is a complete textbook type definition (thanks to MDN Docs for this), but what does it mean in simple words.

Hoisting simply means that we can access variables, before their initialisation & functions, before their declaration.

it gives us an advantage that no matter where functions and variables are declared, they are moved to the top of their scope

regardless of whether their scope is global or local

Examples:

1. Hoisting Function: If i write below code then JS compiler auto move declaration first then call of a function.

```
hello() //call
function hello(){ //declaration
  console.log("Hello world")
}
```

2. Hoisting Var keyword:

```
var x; //declaration
console.log(x) //call //output will be undefined because if value is not assigned before call then
always assign a "undefined" value. whereas it doesn't provide
undefined for const and let keywords
x=7; //assignment
```

```
-----
var x=7; //declaration and assignment
```

```
console.log(x) //call //7
```

```
-----
```

```
x=7; //assignment
```

```
console.log(x) //call //7
```

```
var x; //declaration
```

```
-----
```

```
console.log(x) //call //undefined
```

```
x=7; //assignment
```

```
var x; //declaration
```

```
-----
```

```
console.log(x) //call //undefined
```

```
var x=7; //declaration
```

3. Let/const Hoisting:

```
let x=7; //declaration and assignment
```

```
console.log(x) //call //7
```

```
-----
```

```
const x; //declaration
```

```
console.log(x) //call //Missing initializer in const declaration
```

```
x=7; //assignment
```

```
-----
```

Hoisting:

```
var x = 1;
```

```
function greeting(){
```

```
  console.log("Hi greeting");
```

```
}
```

```
var greeting1 = function (){
```

```
  console.log("Hey greeting1");
```

```
}
```

```
var greeting2 = () =>{
```

```
  console.log("Hello greeting2");
```

```
}
```

```
console.log(x); //1
```

```
greeting(); //Hi greeting
```

```
greeting1(); //Hey greeting1
```

```
greeting2(); //Hello greeting2
```

```
-----
```

```
console.log(x); //undefined //1
```


greeting(); //function body will assign before executing //Hi greeting- as output post execution
//greeting1(); //Uncaught TypeError: greeting1 is not a function because its treating like a
variable not a function, so rather than storing a function body it's storing undefined by default.
//greeting2(); ///Uncaught TypeError: greeting1 is not a function because its treating like a
variable not a function, so rather than storing a function body it's storing undefined by default.

```
var x = 1;
```

```
function greeting(){  
  console.log("Hi greeting");  
}
```

```
var greeting1 = function (){  
  console.log("Hey greeting1");  
}
```

```
var greeting2 = () =>{  
  console.log("Hello greeting2");  
}
```

```
=====
```

Window and This:

Window:

1. Javascript engine create global context execution and allocate some memory space. It is a big object with lot of methods and functions which is created in global space by JS engine.
2. Window is the main JavaScript object root, aka the global object in a browser, and it can also be treated as the root of the document object model. You can access it as window.
3. window.document or just document is the main object of the potentially visible (or better yet: rendered) document object model/DOM.
4. window is the global object, you can reference any properties of it with just the property name - so you do not have to write down window. - it will be figured out by the runtime.
5. window.document.getElementById("header") is the same as document.getElementById("header").

This:

1. At global level THIS points to the window.
2. With Window object THIS variable is created by default.
3. This === window //true

Example:

```

var a=10;
function b(){
  var x=10;
}
console.log(window.a); //10
console.log(a); //10
console.log(this.a); //10

```

```

=====
=====
=====

```

Even Propagation an STOP Propagation: Bydefault event capturing happen first and then even bubbling happen.

Event Bubbling:

1. When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors. With bubbling, the event is first captured and handled by the innermost element and then propagated to outer elements.
2. Bydefault event bubbling execute. To run event bubbling required to mention 3rd argument as FALSE or nothing.
3. "child clicked"
"parent clicked"
"grandparent clicked"
4. Drawback: Bubbling not occur at blur, focus, resizing of window etc.

Event Capturing or Event Trickling:

1. With capturing, the event is first captured by the outermost element and propagated to the inner elements.
2. To run event capturing required to mention 3rd argument as TRUE.
3. "grandparent clicked"
"parent clicked"
"child clicked"

Example:

html-

```

<div id="grandparent">
  <div id="parent">
    <div id="child">
      </div>
    </div>
  </div>
</div>

```

css-

```

div{

```

```
min-width: 10px;
min-height: 10px;
border: 1px solid red;
padding: 30px;
}
```

js-

for bubbling:

```
document.querySelector("#grandparent").addEventListener("click",()=>
{console.log("grandparent clicked")}, false); or
document.querySelector("#grandparent").addEventListener("click",()=>
{console.log("grandparent clicked")});
document.querySelector("#parent").addEventListener("click",()=> {console.log("parent
clicked")}, false); or document.querySelector("#parent").addEventListener("click",()=>
{console.log("parent clicked")});
document.querySelector("#child").addEventListener("click",()=> {console.log("child clicked")},
false); or document.querySelector("#child").addEventListener("click",()=> {console.log("child
clicked")})
```

for capturing:

```
document.querySelector("#grandparent").addEventListener("click",()=>
{console.log("grandparent clicked")}, true);
document.querySelector("#parent").addEventListener("click",()=> {console.log("parent
clicked")},true);
document.querySelector("#child").addEventListener("click",()=> {console.log("child
clicked")},true);
```

stopPropagation:

```
document.querySelector("#grandparent").addEventListener("click",()=>
{console.log("grandparent clicked")}, false);
```

```
document.querySelector("#parent").addEventListener("click",(e)=> {console.log("parent
clicked"); e.stopPropagation();},false);
```

```
document.querySelector("#child").addEventListener("click",()=> {console.log("child
clicked")},false);
```

```
=====
=====
=====
```

Event Delegation:

1. Event delegation makes use of one of the Event Propagation techniques called Event Bubbling
2. if we have a lot of elements handled in a similar way, then instead of assigning a handler to

each of them – we put a single handler on their common ancestor.

3. In the handler we get event.target to see where the event actually happened and handle it.

4. Less memory usage, better performance.

5. Less time required to set up event handlers on the page.

6. Event delegation is a pattern to handle events efficiently in JavaScript. The main idea is to reduce the number of event handlers on a webpage and thus improving the performance of the website.

7. When there are multiple DOM elements present, instead of adding event handlers on each one of them, you can just add one event handler

(on the parent/common ancestor element) which can do the exact same work which all those multiple event handlers were supposed to do.

Example:

html-

Counter: <input type="button" value="1" data-counter>

One more counter: <input type="button" value="2" data-counter>

<script>

```
document.addEventListener('click', function(event) {
```

```
    if (event.target.dataset.counter !== undefined) {
```

```
        event.target.value++;
```

```
        console.log(event.target.value)
```

```
    }
```

```
});
```

</script>

```
=====
=====
=====
```

Polyfill:

1. With the help of polyfill can write own implementation of BIND function.

2. Polyfills is a way to use modern features (usually JS) on browsers where it is currently unsupported. We do this by mimicking the functionality using supported methods along with our own logic.

3. A polyfill is a piece of code (usually JavaScript on the Web) used to provide modern functionality on older browsers that do not natively support it.

Polyfill: Sometimes array push, pop or filter methods and some window's functions like window.localStorage and window.sessionStorage these functions are not supported by browser, so in this case we can provide our own fallback support or own code that it replace the native functions

Example for bind:

With Bind-

```
let name = {
  first: "Priya",
  last: "Bagde"
}
let printName = function() {
  console.log(this.first+this.last)
}
let printNameFinal = printName.bind(name)
printNameFinal()
```

Without Bind-

```
let name = { first: "Priya", last: "Bagde" }
let printName = function(town, state) { console.log(this.first+" "+this.last+" "+town+" "+state) }
Function.prototype.mybind = function(...args) { //printName arguments
  let obj = this; //printName
  params = args.slice(1)
  return function(...args2) { //printNameFinal arguments
    obj.apply(args[0], [...params, ...args2])
  }
}
let printNameFinal = printName.mybind(name, "Chhindwara")
printNameFinal("Madhya Pradesh")
```

```
=====
=====
=====
```

Promises:-

👉 Why do you need a promise?

👉 Promises are used to handle asynchronous operations. They provide an alternative approach for callbacks by reducing the callback hell and writing the cleaner code.

👉 What are the three states of promise ?

👉 Promises have three states:

Pending: This is an initial state of the Promise before an operation begins

Fulfilled: This state indicates that the specified operation was completed.

Rejected: This state indicates that the operation did not complete. In this case an error value will be thrown.

👉 What is promise chaining ?

👉 The process of executing a sequence of asynchronous tasks one after another using promises is known as Promise chaining.

👉 What is promise.all ?

👉 Promise.all is a promise that takes an array of promises as an input (an iterable), and it gets resolved when all the promises get resolved or any one of them gets rejected. For example, the syntax of promise.all method is below,

```
Promise.all([Promise1, Promise2, Promise3])  
.then(result) => { console.log(result) }  
.catch(error => console.log(`Error in promises ${error}`))
```

👉 What are the pros and cons of promises over callbacks ?

👉 Pros:

It avoids callback hell which is unreadable

Easy to write sequential asynchronous code with .then()

Easy to write parallel asynchronous code with Promise.all()

Solves some of the common problems of callbacks(call the callback too late, too early, many times and swallow errors/exceptions)

Cons:

It makes little complex code

You need to load a polyfill if ES6 is not supported

👉 How to cancel a fetch request ?

👉 One shortcoming of native promises is no direct way to cancel a fetch request. But the new AbortController from js specification allows you to use a signal to abort one or multiple fetch calls.

```
=====
```

👉 What are default parameters?

👉 In E5, we need to depend on logical OR operators to handle default values of function parameters. Whereas in ES6, Default function parameters feature allows parameters to be initialized with default values if no value or undefined is passed. Let's compare the behavior with an examples,

//ES5

```
var calculateArea = function(height, width) {  
  height = height || 50;  
  width = width || 60;  
  return width * height;  
}
```

```
console.log(calculateArea()); //300
```

//ES6

```
var calculateArea = function(height = 50, width = 60) {  
  return width * height;  
}
```

```
console.log(calculateArea()); //300
```

👉 After default parameters you should not have parameters without default value-

```
function printValue(a=1, b) {  
    console.log("a = " + a + " and b = " + b);  
}  
printValue(); // Logs: a = 1 and b = undefined  
printValue(7); // Logs: a = 7 and b = undefined  
printValue(7, 3); // Logs: a = 7 and b = 3
```

👉 Default values for parameters and calling it without arguments-

```
function add(a=10, b=20)  
{  
    return a+b;  
}  
console.log(" Sum is : " + add()); // No argument //30  
console.log(" Sum is : " + add(1)); // with one argument //21  
console.log(" Sum is : " + add(5,6)); // with both argument //11
```

👉 JavaScript Default Parameters with null or empty Argument-

```
function test(a = 1)  
{  
    console.log(typeof a);  
    console.log("Value of a: " + a);  
}  
test(); // number, Value of a: 1  
test(undefined); // number, Value of a: 1  
test(""); // string, Value of a:  
test(null); // object, Value of a: null
```

👉 Default Parameters are evaluated at Call time-

```
function append(value, array = []) {  
    array.push(value)  
    return array  
}  
append(1) // [1]  
append(2) // [2], not [1, 2]
```

```
=====
```

JWT Token:

1. JSON Web Token (JWT) is an open standard (RFC 7519) that defines a compact and self-contained way for securely transmitting information between parties as a JSON object.

2. This information can be verified and trusted because it is digitally signed. JWTs can be signed using a public/private key pair using RSA.

```
=====
=====
=====
```

Polyfill of Bind:

```
const name = {
  firstName: "Priya",
  lastName: "Bagde"
}
```

```
let printName = function(hometown, state, country){
  console.log(this.firstName + " " + this.lastName + " " + state + " " + country)
}
let printMyName = printName.bind(name, "Mumbai", "MH");
printMyName("India"); //with bind method
```

//polyfill

```
Function.prototype.mybind = function(...args){
  let obj = this,
      params = args.slice(1);
  return function (...args2){
    obj.apply(args[0], [...params, ...args2])
  }
}
```

```
let printMyName2 = printName.mybind(name, "Mumbai", "MH")
printMyName2("India"); //with polyfill of bind
```

```
=====
=====
=====
```

Polyfill of Call:

```
const myName = {
  firstName: "Priya",
  lastName : "Bagde"
}
```

```
function printName(city, country){
  console.log( `${this.firstName} ${this.lastName}, ${city}, ${country}` );
}
```

```
Function.prototype.myCall= function(context, ...args){
  let currentContext = context || globalThis;
  let randomProp = Math.random();
```



```

while(currentContext.randomProp !== undefined){
  randomProp = Math.random();
}
currentContext.randomProp = this;
let result = currentContext.randomProp(...args);
delete currentContext.randomProp;
return result;
}

```

```

printName.myCall(myName, "MH", "India")
=====
=====
=====

```

Polyfill of Apply:

```

const myName = {
  firstName: "Priya",
  lastName : "Bagde"
}

```

```

function printName(city, country){
  console.log( `${this.firstName} ${this.lastName}, ${city}, ${country}` );
}

```

```

Function.prototype.myCall= function(context, args){
  let currentContext = context || globalThis;
  let randomProp = Math.random();
  while(currentContext.randomProp !== undefined){
    randomProp = Math.random();
  }
  currentContext.randomProp = this;
  let result = currentContext.randomProp(...args);
  delete currentContext.randomProp;
  return result;
}

```

```

printName.myCall(myName, ["MH", "India"])
=====
=====
=====

```

Polyfill of forEach:

```

Array.prototype.ourForEach = function (callback) {
  for (let i = 0; i < this.length; i++) {
    callback(this[i]);
  }
}

```

```

    }
  };
  const names = ["ali", "hamza", "jack"];
  names.forEach((x)=> console.log(x))
  =====
  =====
  =====

```

Polyfill of Map:

```

// Doing this will allow us to use arr.myMap() syntax
Array.prototype.myMap =function(cb) {
  // results results array that gets returned at the end
  const results = [];
  for (let i = 0; i < this.length; i++) {
    // returned values of our cb are pushed in the results[]
    // 'this' refers to the passed array
    results.push(cb(this[i], i, this));
  }

  return results;
}

```

```

const arr = [1, 2, 3, 4, 5, 6];
const myMapResult = arr.myMap((el, _idx, _arr) => {
  return el * 2;
});

```

```

console.log(myMapResult); //[2, 4, 6, 8, 10, 12];
=====
=====
=====

```

Polyfill of Filter:

```

// Doing this will allow us to use arr.myFilter() syntax
Array.prototype.myFilter = function myFilter(cb) {
  const results = [];
  for (let i = 0; i < this.length; i++) {
    if (cb(this[i])) {
      results.push(this[i]);
    }
  }
  return results;
}

```

```

const arr = [1, 2, 3, 4, 5, 6];

```

```
const foo = [
  { name: "S", age: 2 },
  { name: "V", age: 3 },
];
```

```
const myFilterResult = foo.myFilter((el, _idx, _arr) => {
  return el.name !== "S";
});
console.log(myFilterResult); // [{ name: "V", age: 3 }]
```

```
=====
=====
=====
```

Polyfill of Reduce:

// Doing this will allow us to use arr.myReduce() syntax

```
Array.prototype.myReduce =function (cb, initialValue) {
```

```
  let acc;let curr;
```

```
  if (!this.length && !initialValue)
```

```
    throw new Error("Can't reduce on empty array, provide initial value");
```

```
  else {
```

```
    // If initialValue is given then acc is that or acc = is the 0th index of this
```

```
    acc = initialValue ? initialValue : this[0];
```

```
    for (let i = 1; i < this.length; i++) {
```

```
      // current value of the array
```

```
      curr = this[i];
```

```
      // the returned cb value is assigned to acc
```

```
      acc = cb(acc, curr, i, this);
```

```
    }
```

```
  }
```

```
  return acc;
```

```
}
```

```
const arr = [1, 2, 3, 4];
```

```
const myReduceResult = arr.myReduce((acc, curr, _idx, _arr) => {
```

```
  acc += curr;
```

```
  return acc;
```

```
});
```

```
console.log(myReduceResult); // 10
```

```
=====
=====
=====
```

Polyfill of promise.all:

//let myPromiseAll

```
Promise.all = (promises) => {
```

```
  let responses = [];
```

```

let errorResp = [];
return new Promise((resolve, reject) => {
  /** Loop over promises array */
  promises.forEach(async (singlePromise, i) => {
    try {
      /** wait for resolving 1 promise */
      let res = await singlePromise;
      responses.push(res);
      if (i == promises.length - 1) {
        if (errorResp.length > 0) {
          reject(errorResp);
        } else {
          // resolve(responses)
          // To know our custom promise function returning result
          resolve("custom promise ::" + responses);
        }
      }
    } catch (err) {
      errorResp.push(err);
      reject(err);
    }
  });
});
};

```

```

let p1 = Promise.resolve("Promise1 resolved");

```

```

let p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Promise 2 resolved after 2 seconds");
  }, 1000);
});

```

```

Promise.all([p1, p2]).then(
  (res) => {
    console.log("Response => ", res);
    document.write("<b>Response => </b>" + res);
  },
  (err) => {
    console.log("error ==>", err);
  }
);

```

```

=====

```

=====

=====

Polyfill of //let myPromiseAll

```
Promise.all = (promises) => {
  let responses = [];
  let errorResp = [];
  return new Promise((resolve, reject) => {
    /** Loop over promises array */
    promises.forEach(async (singlePromise, i) => {
      try {
        /** wait for resolving 1 promise */
        let res = await singlePromise;
        responses.push(res);
        if (i == promises.length - 1) {
          if (errorResp.length > 0) {
            reject(errorResp);
          } else {
            // resolve(esponses)
            // To know our custom promise function returning result
            resolve("custom promise ::" + responses);
          }
        }
      } catch (err) {
        errorResp.push(err);
        reject(err);
      }
    });
  });
};
```

```
let p1 = Promise.resolve("Promise1 resolved");
let p2 = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Promise 2 resolved after 2 seconds");
  }, 1000);
});
let p3 = Promise.reject("Promise Rejected")
Promise.all([p1, p2, p3]).then(
  (res) => {
    console.log("Response => ", res);
    document.write("<b>Response => </b>" + res);
  },
  (err) => {
```

```

    console.log("error =>", err);
  }
);
=====
=====
=====
Polyfill of all.settled:-
Promise.allSettled = function(promises) {
  let mappedPromises = promises.map((p) => {
    return p
      .then((value) => {
        return {
          status: 'fulfilled',
          value
        };
      })
      .catch((reason) => {
        return {
          status: 'rejected',
          reason
        };
      });
  });
  return Promise.all(mappedPromises);
};

let promises = [
  Promise.resolve(2),
  Promise.reject('This is rejected'),
  new Promise((resolve, reject) => setTimeout(resolve, 400, 67)),
];

Promise.allSettled(promises).then((result) => console.log(result));
/*[{
  status: "fulfilled",
  value: 2
}, {
  reason: "This is rejected",
  status: "rejected"
}, {
  status: "fulfilled",
  value: 67
}]*/

```

```
=====
=====
=====
```

Polyfill of promises.allsettled:

```
Promise.allSettled = function (arrayOfPromises = []) {
  return new Promise(function promiseltor(resolve, reject) {
    let result = [];
    arrayOfPromises.forEach((item) => {
      item
      .then((value) => {
        result.push({ status: "fulfilled", value: value });
        if (arrayOfPromises.length === result.length) resolve(result);
      })
      .catch((err) => {
        result.push({ status: "rejected", reason: `${err}` });
        if (arrayOfPromises.length === result.length) resolve(result);
      });
    });
  });
};
```

```
let promises = [
  Promise.resolve(2),
  Promise.reject("This is rejected"),
  new Promise((resolve, reject) => setTimeout(resolve, 400, 67)),
];
```

```
Promise.allSettled(promises).then((result) => console.log(result));
```

```
=====
=====
=====
```

Polyfill of race:

```
Promise.letsBuildARace = function(arrayOfPromises){
  return new Promise((resolve, reject) => {
    arrayOfPromises.forEach((promise) => {
      Promise.resolve(promise).then(resolve, reject);
    })
  })
}
```

For Example:

```
var a = new Promise((resolve) => setTimeout(()=>{resolve(3)},200));
var b = new Promise((resolve,reject)=>reject(9));
var c= new Promise((resolve) => resolve(5));
```

```
var d= Promise.letsBuildARace([a,b,c]);
d.then(result=>console.log(result))
```

```
=====
=====
=====
```

Polyfill of Any:

```
Promise.letsBuildAnAny = function(arrayOfPromises){
let errors = [];
return new Promise((resolve, reject) => {
arrayOfPromises.forEach((promise, index)=>{
Promise.resolve(promise)
.then(resolve)
.catch((error)=>{
errors.push(error);
if(errors.length == arrayOfPromises.length)
reject(errors);
})
})
})
}
```

For Example:

```
var a = new Promise((resolve) => setTimeout(()=>{resolve(3)},200));
var b = new Promise((resolve,reject)=>reject(9));
var c= new Promise((resolve) => resolve(5));
```

```
var d= Promise.letsBuildAnAny([a,b,c]);
d.then(result=>console.log(result))
```

```
=====
=====
=====
```

Rest Parameter in Array :

function addSum(a,b,c, ...rest){ // ...rest indicating combination of those arguments which are left(rest).

```
  console.log(...rest) //6,7
  console.log(rest) //[6,7]
  console.log(arguments) //ES5 //{ "0":2, "1":3, "2":4, "3":6, "4":7}
  return a+b+c+rest[0]+rest[1];
}
```

```
console.log(addSum(2,3,4,6,7)) //22
```

```
=====
=====
=====
```


//Spread Operator in Array:

```
function getNames(name1, name2, name3){  
  console.log(name1,name2, name3);  
}
```

```
var names =["priya", "riya", "supriya"]
```

```
getNames(names[0], names[1], names[2]); //"priya" "riya" "supriya"
```

getNames(...names) //spread operator here used to spread the individual arguments //best approach because here we are passing all the arguments but we can use some of the arguments inside function without an error. Other approaches will gives an error to pass those arguments which are used in function.

```
getNames(names)
```

```
=====
```

Rest Spread in object:

```
var student = {  
  name: "priya",  
  age : 100,  
  hobbies : ["cooking", "dancing"]  
}
```

```
//const age = student.age; //earlier we like this
```

```
//console.log(age) //10
```

```
//const {age, ...rest} = student; //using destructuring
```

```
//console.log(age, rest) //100 {"name": "priya", "hobbies":["cooking", "dancing"]}
```

```
const {...rest} = student;
```

```
console.log(rest) //{"name": "priya", "age": 100, "hobbies":["cooking", "dancing"]}
```

```
=====
```

//spread operator in object: (Change the value of age)

```
var student = {  
  name: "priya",  
  age : 100,  
  hobbies : ["cooking", "dancing"]  
}
```

```
var newStudent = {  
  ...student, //copying one object to another object  
  age : 101  
}
```

```
console.log(newStudent)
```

```
=====
=====
=====
```

Callback, Promise and Async/await :-

```
const data = [ //array of object
```

```
  {name: "priya", role: "software developer"},
```

```
  {name: "riya", role: "software developer"},
```

```
  {name: "supriya", role: "software developer"}]
```

```
];
```

```
function getData(){
```

```
  setTimeout(()=>{
```

```
    let output ="";
```

```
    data.forEach((item)=>{
```

```
      output +=`<li> ${item.name}</li>`
```

```
    })
```

```
    document.body.innerHTML = output;
```

```
  }, 1000) //use 5000 instead of 1000 then we get a name of "dhanupriya".
```

```
}
```

```
function createData(dataInput){
```

```
  setTimeout(()=>{data.push(dataInput)}, 2000)
```

```
}
```

```
createData({name: "dhanupriya", role: "software developer"})
```

```
getData()
```

```
=====
```

Callback:

```
const data = [ //array of object
```

```
  {name: "priya", role: "software developer"},
```

```
  {name: "riya", role: "software developer"},
```

```
  {name: "supriya", role: "software developer"}]
```

```
];
```

```
function getData(){
```

```
  setTimeout(()=>{
```

```
    let output ="";
```

```
    data.forEach((item)=>{
```

```
      output +=`<li> ${item.name}</li>`
```

```
    })
```

```
    document.body.innerHTML = output;
```

```
  }, 1000)
```

```
}
```

```
function createData(dataInput, callback){
```

```

    setTimeout(()=>{
        data.push(dataInput)
        callback(); //getData function will get call once we push the new dataInput
    },2000)
}
createData({name: "dhanupriya", role: "software developer"}, getData) //we are passing getData
function as a callback
=====

```

Promises:

```

const data = [ //array of object
    {name: "priya", role: "software developer"},
    {name: "riya", role: "software developer"},
    {name: "supriya", role: "software developer"}
];

function getData(){
    setTimeout(()=>{
        let output ="";
        data.forEach((item)=>{
            output +=`<li> ${item.name}</li>`
        })
        document.body.innerHTML = output;
    }, 1000) //use 5000 instead of 1000 then we get a name of "dhanupriya".
}

function createData(dataInput){
    return new Promise((resolve, reject)=>{
        setTimeout(()=>{
            data.push(dataInput);
            let error = false; //if true then went to catch block
            if(!error) resolve();
            else reject("Error!!!!");
        },2000)
    })
}

createData({name: "dhanupriya", role: "software developer"})
.then(getData)
.catch(err => console.log("Errors !"))
=====

```

Async/Await:

```

const data = [ //array of object
    {name: "priya", role: "software developer"},
    {name: "riya", role: "software developer"},

```

```
{name: "supriya", role: "software developer"}  
];
```

```
function getData(){  
  setTimeout(()=>{  
    let output ="";  
    data.forEach((item)=>{  
      output +=`<li> ${item.name}</li>`  
    })  
    document.body.innerHTML = output;  
  }, 1000) //use 5000 instead of 1000 then we get a name of "dhanupriya".  
}  
function createData(dataInput){  
  return new Promise((resolve, reject)=>{  
    setTimeout(()=>{  
      data.push(dataInput);  
      let error = false; //if true then went to catch block  
      if(!error) resolve();  
      else reject("Error!!!!");  
    },2000)  
  })  
}
```

```
async function start(){ //using sync and await  
  await createData({name: "dhanupriya", role: "software developer"});  
  getData();  
}  
start();
```

```
=====
```

Callback:

```
var sum = function(a,b,c){  
  return{  
    sumOfTwo: function(){return a+b},  
    sumOfThree:function(){return a+b+c}  
  }  
}
```

```
var store = sum(2,3,4);  
console.log(store)  
console.log(store.sumOfTwo())  
console.log(store.sumOfThree())
```

```
-----  
var sum = function(a){  
    var c=4;  
    return function(b){  
        return a+b+c;  
    }  
}
```

```
var store = sum(2);  
console.log(store)  
console.log(store(5))
```

```
=====
```

Call, apply and bind:

```
var userDetails = {  
    name: "Priya",  
    age: 100,  
    role: "Software Developer",  
    printDetail : function(){  
        console.log(this)// this representing userDetails whole object  
    }  
}  
userDetails.printDetail();
```

```
var userDetails2 = {  
    name: "Riya",  
    age: 101,  
    role: "Software Developer"  
}
```

userDetails.printDetail.call(userDetails2); //if i want line 6th "this" should point to userDetails2 then use call function.

```
-----  
var userDetails = {  
    name: "Priya",  
    age: 100,  
    role: "Software Developer"  
}
```

```
let printDetail = function(){  
    console.log(this.name)  
}
```

printDetail.call(userDetails); // if function is independent of object.

```
var userDetails2 = {
```

```
    name: "Riya",
    age: 101,
    role: "Software Developer"
  }
  printDetail.call(userDetails2);
```

```
-----
var userDetails = {
  name: "Priya",
  age: 100,
  role: "Software Developer"
}
let printDetail = function(country){
  console.log(this.name+" "+country)
}
printDetail.call(userDetails, "India"); // if function is independent of object.
```

```
var userDetails2 = {
  name: "Riya",
  age: 101,
  role: "Software Developer"
}
```

```
printDetail.call(userDetails2, "India");
printDetail.apply(userDetails2, ["India"]);
let bindfunc= printDetail.bind(userDetails2, "India"); //creating a copy and invoke whenever
require //don't call the directly here like call function
bindfunc();
```

```
=====
=====
=====
```

Foreach:

```
const Data=[
{name:"Google", category:"Product Based"},
{name:"Accenture", category:"service Based"},
{name:"Amazon", category:"Product Based"}
];
```

```
for(let i=0; i<Data.length; i++){
  console.log(Data[i])
}
```

```
//foreach is HOF which take callback
Data.forEach((item)=>{ console.log(item) })
```

```
=====
=====
=====
Filter:
```

```
const Age=[1,2,3,4,5,6,7,8,9,0]
for(var i=0;i<Age.length;i++){
  if(Age[i]>5){
    console.log(Age[i])
  }
}
```

```
const UpdatedAge= Age.filter(function(item){
  if(item>5){
    return true;
  }
})
console.log(UpdatedAge)
```

```
const UpdatedAge= Age.filter((item)=>(item>5))
console.log(UpdatedAge)
```

```
=====
=====
=====
sort:-
```

```
const Data =[11,44,2,66,3,7,22,3]
const sorted= Data.sort((a,b)=>{
  if(a>b) return 1;
  else return -1;
})
console.log(sorted) //prefer for object
//a<b //[66,44,22,11,7,3,3,2]
//a>b //[2,3,3,7,11,22,44,66]
```

```
const sorted1= Data.sort((a,b)=>b-a)
console.log(sorted1) //prefer for arrays
//b-a //[66,44,22,11,7,3,3,2]
//a-b //[2,3,3,7,11,22,44,66]
```

```
=====
=====
=====
Reduce:
```

```
const Data =[11,44,2,66,3,7,22,3]
```

```

var updatedData=0;
for(var i=0; i<Data.length; i++){
  updatedData=updatedData+Data[i]
}
console.log(updatedData)
-----
const updated= Data.reduce((acc, item)=>{
  return acc+item
},0)

```

```

console.log(updated)
=====
=====
=====

```

ProtoType: If we can inherit the property of the one class to other class called as Inheritance.

```

const objName = {
  name : "priya",
  getName : function(){
    return this.name; //wanted to access above name inside block so we used "this"
  }
}
console.log("1",objName.getName());

```

```

const objState = {
  //name: "riya",
  state : "pune",
  __proto__ : objName
}
console.log("2",objState.getName()); //if will find in current object, if not available then it move to
upper object continuously. If still not found then it will give undefined.

```

```

const objCountry = {
  name: "supriya",
  country : "india",
  __proto__ : objState
}
console.log("3", objCountry.getName() ,objCountry.state)
-----

```

```

Array.prototype.convertToObject = function(){
  let newObj ={}
  this.forEach( ele =>{

```



```

        newObj[ele]= ele;
    }
)
return newObj;
}
const arr =["priya"];
console.log(arr.convertToObject()); //array to object conversion with prototype

```

```

function myProtoType(name){
    return this.name=name;
}
myProtoType.prototype=objName;
const myproto = myProtoType("priya")
console.log(myproto);
console.log(myproto.getName());

```

```

=====
=====
=====
Set: It contains only unique values. Can iterate.
let arr =[1,2,3]
let obj = new Set(arr) //set takesarray, string, object etc
obj.add(4)
obj.delete(3)
console.log(obj)

```

```

var obj1={name:"priya"}
obj.add(obj1)
console.log(obj)

```

```

=====
=====
=====
Map: store in key value pair. Can iterate.
let myMap = new Map([["a1", "priya"],["a2","riya"]])
myMap.set("a2","supriya")
console.log(myMap.get("a2"))// will get last value which we will assign
for(let [key,value] of myMap){
    console.log(`keys ${key}, value ${value}`)
}

```

```

=====
=====
=====
WeakSet: only store object, can't iterate with for/foreach
let ws = new WeakSet()

```

```
const obj={"name":"priya"}
ws.add(obj)
console.log(ws.has(obj));
```

WeakMap: It's similar to WeakSet where can't able to iterate and only stores an object

```
=====
=====
=====
```

Hoisting : Its related to memory management.

Global execution context having 2 component. In memory componenet variable and functions will store. Variable store with undefined and function store with prototype of function. Var or functions will push to Callstack and then pop.

```
=====
=====
=====
```

2 ways of Curryng function with closures:

*****Using Normal Function*****

```
function add(a){
  return function(b){
    if(b) return add(a+b);
    return a;
  }
}
console.log('Sum :', add(1)(2)(3)(4)(5)(6)(7)(8)(9)) //"Sum :", 36
```

*****Using Arrow Function*****

```
function sum(num1) {
  return (num2) => {
    if(!num2) {
      return num1;
    }
    return sum(num1 + num2);
  }
}
console.log('Sum :', sum(1)(2)(3)(4)(5)(6)(7)(8)(9)) //"Sum :", 36
```

*****Using Arrow ShortHand Function*****

```
const sum = (a) => (b) => (c) => (d) => a+b+c+d
console.log('Sum :', sum(1)(2)(3)(4))
```

*****Using Partial Function*****

```
function sum(...a){
  return function(...b){
    if(b?.length){
      return sum([...a,...b].reduce((acc, curr)=>acc+curr,0))
    }
    return a;
  }
}
console.log('Sum :', sum(1,2,3)(4,5)(6,7,8,9)()) //"Sum :", 36
```

=====

=====

=====

How to Remove All Falsy Values from an Array in JavaScript:

*****Using Filter Auto Coercion*****

```
[1, 2, 'b', 0, {}, "", NaN, 3, undefined, null, 5].filter(a => a);
```

*****Using !!*****

```
[1, 2, 'b', 0, {}, "", NaN, 3, undefined, null, 5].filter(a => !!a);
```

*****Using Boolean*****

```
[1, 2, 'b', 0, {}, "", NaN, 3, undefined, null, 5].filter(a => Boolean(a));
```

*****Using Boolean directly*****

```
[1, 2, 'b', 0, {}, "", NaN, 3, undefined, null, 5].filter(Boolean);
```

//This works because Boolean itself is a function, and the arguments filter supplies are passed directly to it.

//Boolean() is also a function that returns truthy when true and falsy when false!

=====

=====

=====

Memo Function:

*****Memoizing functions*****

```
const add = (n) => (n + 10);
add(9);
```

```

// a simple memoized function to add something
const memoizedAdd = () => {
  let cache = {};
  return (n) => {
    if (n in cache) {
      console.log('Fetching from cache');
      return cache[n];
    }
    else {
      console.log('Calculating result');
      let result = n + 10;
      cache[n] = result;
      return result;
    }
  }
}

// returned function from memoizedAdd
const newAdd = memoizedAdd();
console.time(); console.log(newAdd(9)); console.timeEnd();// calculated
console.time(); console.log(newAdd(9)); console.timeEnd();      // cached
console.time(); console.log(newAdd(9)); console.timeEnd();      // cached
//output:
//Calculating result   19   default: 0.38427734375 ms
//Fetching from cache  19   default: 0.30810546875 ms
//Fetching from cache  19   default: 0.27612304687 ms

```

*****Memoizing recursive functions*****

```

// a simple pure function to get a value adding 10
const add = (n) => (n + 10);
// a simple memoize function that takes in a function and returns a memoized function
const memoize = (fn) => {
  let cache = {};
  return (...args) => {
    let n = args[0]; // just taking one argument here
    if (n in cache) {
      console.log('Fetching from cache');
      return cache[n];
    }
    else {
      console.log('Calculating result');
      let result = fn(n);
      cache[n] = result;
    }
  }
}

```

```

    return result;
  }
}
}
// creating a memoized function for the 'add' pure function
const memoizedAdd = memoize(add);
console.time(); console.log(memoizedAdd(3)); console.timeEnd();// calculated
console.time(); console.log(memoizedAdd(3)); console.timeEnd(); // cached
console.time(); console.log(memoizedAdd(3)); console.timeEnd(); // cached
//output:
//Calculating result    13    default: 0.45068359375 ms
//Fetching from cache   13    default: 0.16674804687 ms
//Fetching from cache   13    default: 0.161865234375 ms
=====
=====
=====
*****Using Custom function with Recursion*****
const age=[1,2,
  [12,23,[75,34,[2,34]],[32,45]],
  [3,456],
  [56,[5]]]
var result=[]
function flattenArray(inputArray)
{
  inputArray.map((value)=>{
    if(Array.isArray(value))
    {
      flattenArray(value)
    }
    else{
      result.push(value)
    }
  })
}
flattenArray(age)
console.log(result) //[1,2,12,23,75,34,2,34,32,45,3,456,56,5]

*****Using One Line Arrow function*****
var result=[]
const flattenArray = (inputArray) => {
  inputArray.map((value)=> Array.isArray(value) ? flattenArray(value) : result.push(value))
}
flattenArray(age)
console.log(result) //[1,2,12,23,75,34,2,34,32,45,3,456,56,5]

```

*****Using Flat*****

```
const age=[1,2,
  [12,23,[75,34,[2,34]],,[32,45]],
  [3,456],
  [56,[5]]]
var updatedArr= age.flat(Infinity);
console.log(updatedArr) //[1,2,12,23,75,34,2,34,32,45,3,456,56,5]
```

*****Using Reducer*****

```
function flatDeep(arr, d = 1) {
  return d > 0 ? arr.reduce((acc, val) =>
    acc.concat(Array.isArray(val) ? flatDeep(val, d - 1) : val), [])
    : arr.slice();
};
console.log(flatDeep(arr, Infinity)) //[1,2,12,23,75,34,2,34,32,45,3,456,56,5]
```

=====

=====

=====

Regular Expression(Regex):

A regular expression is a group of characters or symbols which is used to find a specific pattern in a text.

Regular expressions are normally case-sensitive so the regular expression The would not match the string the

=====

=====

=====

Multiple functions:

```
var arr = [];
const checkSort =(a)=>{
  if(!arr.includes(a) && Number.isInteger(a)){
    arr.push(a)
  }
  return console.log(arr.sort((a,b)=>a-b));
}
checkSort(5);
checkSort(4);
checkSort(2);
checkSort("abc");
checkSort("44");
checkSort(-44);
```

```
//output: [-44,2,4,5]
```

```
=====
=====
=====
```

```
[..."priya"]
// ['p', 'r', 'i', 'y', 'a']
"priya".split("")
// ['p', 'r', 'i', 'y', 'a']
```

```
=====
=====
=====
```

Deep Copy and Shallow Copy:

A deep copy means that "all of the values of the new variable" are copied and disconnected from the original variable. It makes a copy of all the members of X, allocates different memory location for Y and then assigns the copied members to Y to achieve deep copy. In this way, if X vanishes Y is still valid in the memory.

A shallow copy means that certain (sub-)values are still connected to the original variable. It is primarily utilized for copying One Dimensional array elements, where it only copies the elements present at the first level.

A deep copying means that value of the new variable is disconnected from the original variable while a shallow copy means that some values are still connected to the original variable.

addresses of X and Y will be the same i.e. they will be pointing to the same memory location.

To copy an object in JavaScript, you have three options. Both spread (...) and Object.assign() perform a shallow copy while the JSON methods carry a deep copy.

--->Assignment operator "="

--->Use the spread (...) syntax

--->Use the Object.assign() method

--->Use the JSON.stringify() and JSON.parse() methods --where the stringify() method converts a particular JavaScript object to a string, and then the parse() method performs the parsing operation and returns an object.

For a primitive value, you just simply use a simple assignment: The important takeaway here is that you can quickly copy a primitive data type's exact value in a separate memory space by creating and assigning another variable to the variable being copied. Take note of how it is instantiated — const will not allow later changes.

```
let counter = 1;
```

```
let copiedCounter = counter;
```

```
copiedCounter = 2; //And when you change the value of the copied variable, the value of the
```

original remains the same.

```
console.log(counter);
```


Output: 1


However, if you use the assignment operator for a reference value, it will not copy the value. Instead, both variables will reference the same object in the memory:

JS objects (as non-primitive data types) differ because they have reference values and those values are mutable. This means they share a memory address when shallow copied.

 What is shallow vs Deep copy in javascript?

As we know in javascript there are two data types, one is primitive and the other is non-primitive. The primitive data type is immutable while non-primitive is mutable.

 In Shallow copy, when we assign a variable to another variable using the assignment operator, We are actually assigning its reference to another variable, which means both variables point to the same memory location. If we change any of them, it will be reflected in both. All non-primitive data types are shallow copied.

 In Deep copy, when we assign a variable to another variable using the assignment operator, We are actually assigning only value and both of these variables are pointing to different locations which means if I change one, then it will not get reflected in both. All primitive data types are deep copied.

```
=====
=====
=====
function countOccurance(arr){
  let map = {}, sortedArr = arr.sort((a,b)=>a-b);
  for(var item of sortedArr){
    if(map[item]){
      map[item]++
    }else{
      map[item] = 1;
    }
  }
  return map;
}
```



```
console.log(countOccurance([1, 4, 91, 21, 91, 0, 32, 4, 11, 3, 54, 34, 21]))
```

```
-----  
function countOccurance(arr){  
  let map = {}, sortedArr = arr.sort((a,b)=>a-b);  
  for(var item of sortedArr){  
    if(!(item in map)){ map[item] = 0}  
    map[item]++;  
  }  
  return map;  
}
```

```
console.log(countOccurance([1, 4, 91, 21, 91, 0, 32, 4, 11, 3, 54, 34, 21]))
```

```
-----  
function countOccurance(arr){  
  let map = {}, sortedArr = arr.sort((a,b)=>a-b);  
  for(var item of sortedArr){  
    if(map[item] == null){ map[item] = 0}  
    map[item]++;  
  }  
  return map;  
}
```

```
console.log(countOccurance([1, 4, 91, 21, 91, 0, 32, 4, 11, 3, 54, 34, 21]))
```

```
-----  
function countOccurance(arr){  
  let map = {}, sortedArr = arr.sort((a,b)=>a-b);  
  for(var item of sortedArr){  
    if(map.hasOwnProperty(item)){  
      map[item] = map[item] + 1  
    }else{  
      map[item] = 1;  
    }  
  }  
  return map;  
}
```

```
console.log(countOccurance([1, 4, 91, 21, 91, 0, 32, 4, 11, 3, 54, 34, 21]))
```

```
-----  
function countOccurance(arr){  
  let map = {}, sortedArr = arr.sort((a,b)=>a-b);  
  for(var item of sortedArr){  
    map[item] = map[item]+1 || 1  
  }  
  return map  
}
```

```
console.log(countOccurance([1, 4, 91, 21, 91, 0, 32, 4, 11, 3, 54, 34, 21]))
```

```
-----  
function countOccurance(arr){  
  let map = {}, sortedArr = arr.sort((a,b)=>a-b); //[0,1,3,4,4,11,21,21,32,34,54,91,91]  
  for(var item of sortedArr){  
    map[item] = (map[item] || 0) + 1  
  }  
  return map  
}  
console.log(countOccurance([1, 4, 91, 21, 91, 0, 32, 4, 11, 3, 54, 34, 21]))
```

```
-----  
function countOccurance(arr){  
  let map = {}, sortedArr = arr.sort((a,b)=>a-b);  
  for(var item of sortedArr){  
    if(map[item]){  
      map[item] = map[item] + 1  
    }else{  
      map[item] = 1;  
    }  
  }  
  return map;  
}  
console.log(countOccurance([1, 4, 91, 21, 91, 0, 32, 4, 11, 3, 54, 34, 21]))
```

```
-----  
function countOccurance(arr){  
  let map = {}, sortedArr = arr.sort((a,b)=>a-b);  
  for (var i=0; i<sortedArr.length; i++)  
  {  
    map[sortedArr[i]] = map[sortedArr[i]] ? map[sortedArr[i]]+1 : 1;  
  }  
  return map  
}  
console.log(countOccurance([1, 4, 91, 21, 91, 0, 32, 4, 11, 3, 54, 34, 21]))
```

```
-----  
var frequencySort = function(nums) {  
  const hash={}  
  for(let item of nums){  
    hash[item]  
      ?hash[item]++  
      :hash[item]=1  
  }  
  return hash  
};  
console.log(frequencySort([1,1,2,2,2,3]))
```

```

-----
var frequencySort = function(nums) {
  let map = new Map();
  for (const i of nums) {
    if (map.has(i)) {
      map.set(i, map.get(i) + 1);
    } else {
      map.set(i, 1);
    }
  }
  console.log(map);
  return map;
};

```

```

console.log(frequencySort([1,1,2,2,2,3]))

```

```

=====
=====
=====

```

Anagram: O(N)

```

let NO_OF_CHARS = 256;

```

```

function areAnagram(str1, str2)

```

```

{
    // Create a count array and initialize all values as 0
    let count = new Array(NO_OF_CHARS).fill(0);
    let i;

```

```

    if (str1.length != str2.length) return false;

```

```

    for(i = 0; i < str1.length; i++)

```

```

    {

```

```

        count[str1[i].charCodeAt(0)]++;

```

```

        count[str2[i].charCodeAt(0)]--;

```

```

    }

```

```

    // See if there is any non-zero value in count array

```

```

    for(i = 0; i < NO_OF_CHARS; i++)

```

```

    if (count[i] != 0)

```

```

    {

```

```

        return false;

```

```

    }

```

```

    return true;

```

```

}

```

```

// Driver code

```

```

let str1 =
"priya".split("");
let str2 =
"riyap".split("");

// Function call
if (areAnagram(str1, str2))
    document.write(
        "The two strings are " +
        "anagram of each other");
else
    document.write(
        "The two strings are " +
        "not anagram of each other");

```

```

=====
=====
=====

```

```

// How can you make a linkedlist nodes in javascript and
// hpw to check linkedlist is cyclic or not.

```

```

class LinkedList {
    constructor(data){
        this.data = data;
        this.next = null;
    }
}

```

```

let head = new LinkedList(10);
let A = new LinkedList(20);
let B = new LinkedList(30);
let C = new LinkedList(40);
let D = new LinkedList(50);

```

```

head.next = A;
A.next = B;
B.next = C;
C.next = D;
D.next = A; //it will be circluar linked list
//D.next = null; //it will be single linked list

```

```

//output : 10 -> 20 -> 30 -> 40 -> Null

```

```
function floyedCycleDetection(){
  let fast = head, slow = head;
  while(fast && fast.next){
    slow = slow.next;
    fast = fast.next.next;
    if(slow == fast) return true;
  }
  return false
}
```

```
=====
=====
=====
```

This: This is a keyword from where it gets belongs to. This is a current object. In javascript, this behave in diff manner, it depends in the env where it's getting invoked. (<https://www.youtube.com/watch?v=wp-NEcAck1k&t=427s>)

In IIFE : this is pointing to Window

In object : if normal function then pointing to block scope. if arrow function pointing to window.

In function constructor : pointing to function

If we didn't invoked properly(without paranthesis) then it will give undefined.

```
=====
=====
=====
```

```
function curryingWithRecursion(a){
  return function(b) {
    return b ? curryingWithRecursion(a+b) : a
  }
}
```

console.log(curryingWithRecursion(1)(2)(3)(4)(5)()) //always use at last empty parenthesis

```
=====
=====
=====
```

How can you optimise the multiple of times API calls - using debaouncing, we can restrict the function calls ..

Check code implementation:

```
function debounce(func, a){
  let timer;
  return ()=>{
    clearTimeout(timer);
    timer = setTimeout(()=>{
      func();
    }, a)
  }
}
```

```
}  
const printName = () => {  
  console.log("Priya");  
}
```

```
debounce(printName(), 8000)()
```

```
=====
```

```
function maxSumSubArray(arr){  
  let max = 0;  
  for(let i=0; i<arr.length; i++){  
    let currMax = 0;  
    for(let j=i; j<arr.length; j++){  
      currMax += arr[j]  
      if(currMax > max){ //or max = Math.max(max, currMax)  
        max = currMax  
      }  
    }  
  }  
  return max  
}
```

```
console.log(maxSumSubArray([1,2,-1,3,-2]))
```

```
=====
```

```
function hello(){  
  for(var i=0; i<=3; i++){  
    setTimeout(()=>{  
      console.log(i)  
    }, 1000)  
  }  
}  
hello() //4 4 4 4
```

```
function hello(){  
  for(let i=0; i<=3; i++){ //using let getting new references  
    setTimeout(()=>{  
      console.log(i)  
    }, 1000)  
  }  
}
```

```
}  
hello() //0 1 2 3
```

```
function hello(){  
  for(let i=0; i<=3; i++){ //alternative of let i.e closures and passing indexes  
    function hi(i){  
      setTimeout(()=>{  
        console.log(i)  
      }, 1000)  
    }  
    hi(i)  
  }  
}  
hello() //0 1 2 3
```

```
=====
```

```
=====
```

```
=====
```

```
//move zeroes
```

```
function moveZeroes(arr){  
  for(let i=0; i<arr.length; i++){  
    if(arr[i] === 0){  
      arr.push(0);  
      arr.splice(i,1)  
    }  
  }  
  return arr  
}  
console.log(moveZeroes([0,2,3,0,50,0,3,0,8,2,0]))
```

```
=====
```

```
=====
```

```
=====
```

```
function sumofTwopair(arr, target){  
  for(let i=0; i<arr.length; i++){  
    for(let j=i+1; j<arr.length; j++){  
      if(arr[i]+arr[j] === target) return true  
    }  
  }  
  return false  
}  
console.log(sumofTwopair([0,22,3,0,50,0,3,0,8,2,0],5))
```

```
-----
```

```
function sumofTwopair(arr, target){
```

```

let s = new Set();
for(let item of arr){
  if(s.has(target-item)){
    return true
  }else{
    s.add(item)
  }
}
return false
}
console.log(sumofTwopair([0,2,3,0,50,0,3,0,8,2,0],5))

```

```

=====
=====
=====

```

Polyfill of Flat:

```

function flatArray(arr){
  let result = arr.reduce((accum, item)=>{
    Array.isArray(item)
    ? accum.push(...flatArray(item))
    : accum.push(item);
    return accum
  },[])
  return result;
}
console.log(flatArray([1,2,3,[4,[5,[6,7,[8]]]]]))

```

```

=====
=====
=====

```

Merge 2 sorted array:

```

function mergedTwoArrays(arr1,arr2){
  let mergedArray= [];
  let i = 0, j=0;
  while((i < arr1.length)&&(j< arr2.length)){
    if(arr1[i]<arr2[j]){
      mergedArray.push(arr1[i]);
      i++;
    }
    else{
      mergedArray.push(arr2[j]);
      j++;
    }
  }
  if(i<=(arr1.length-1)){

```



```

    arr1.splice(0,i);
    mergedArray=mergedArray.concat(arr1);
  }
  else if(j<=(arr2.length-1)){
    arr2.splice(0,j);
    mergedArray=mergedArray.concat(arr2);
  }
  return mergedArray;
}
console.log(mergedTwoArrays([1,2,3,4], [2,5,6]))

```

```

function mergedTwoArrays(arr1,arr2){
  let result= [], i=0, j=0;
  while(i < arr1.length && j< arr2.length){
    if(arr1[i]<arr2[j]){
      result.push(arr1[i]); i++;
    }
    else{
      result.push(arr2[j]); j++;
    }
  }
  while(i < arr1.length){
    result.push(arr1[i]); i++;
  }
  while(j< arr2.length){
    result.push(arr2[j]); j++;
  }
  return result;
}
console.log(mergedTwoArrays([1,2,3,4], [2,5,6]))

```

```

=====
=====
=====

=====
=====
=====

```

*****Implementation of Currying*****

```

//fn is pointing to sum function, i.e (a,b,c,d) => a+b+c+d.
function currying(fn){
  //args is pointing to (1)(2)(3)(4)(5)
  return function curry(...args){ //1 //2 //3 //4

```

```

//when length get matched, then add all the values
if(args.length >= fn.length) return fn(...args);
else{
  return function(...next){
    return curry(...args, ...next)
  }
}
}

```

```

const sum = (a,b,c,d) => a+b+c+d;
const totalSum = currying(sum)
console.log(totalSum(1)(2)(3)(4))

```

```

=====
=====
=====

```

Destructuring in nested object:

```

const obj = {
  name : "John",
  address : {
    country : "India",
    add : {
      city : "Delhi"
    }
  }
}

```

```

const {address} = obj;
console.log(address) //{country : "India", "add": {"city" : "India"}}

```

```

const {address:{country}} = obj;
console.log(country) //India

```

```

const {address:{add:{city}}} = obj;
console.log(city) //Delhi

```

```

=====
=====
=====

```

```

=====
=====
=====

```

```
=====
=====
=====
```

```
-----
-----
-----
-----
-----
-----
```

Dassault Systems:

```
const inputArray = ["INDIA:[Delhi,Mumbai,Chennai]"]; //1
const expectedOutputArray = ["INDIA:[Delhi]", "OR", "INDIA:[Mumbai]", "OR",
"INDIA:[Chennai]"];
```

//GENERIC JS FUNCTION --> String with comma will split as separate items in new ARRAY and separated with OR between each

```
function convertToORBasedExpression (arr){
  var outputArray =[];
  var temp = arr[0].split(":")[1]; //["Delhi,Mumbai,Chennai"]
  var tmp = temp.slice(1,21).split(",") //["Delhi","Mumbai","Chennai"]
  for (let item of tmp){
    //console.log("INDIA:"+item)
    outputArray.push("INDIA:"+item)
  }
  // console.log(temp.slice(1,21).split(","))

  return outputArray; //5
}
convertToORBasedExpression(inputArray)
```

```
=====
=====
```

30 interview questions and answers related to the polyfill of the `.bind()` method in JavaScript:

1. What is the purpose of the `.bind()` method in JavaScript?
- Answer: The `.bind()` method is used to create a new function with a specified `this` value and initial parameters.`
2. How does the polyfill save a reference to the original function?

- Answer: It uses ``var originalFunction = this;`` to store a reference to the original function.
3. How does the ``bind()`` method differ from other function-invoking methods?
 - Answer: Unlike other methods that immediately invoke the function, ``bind()`` returns a new function with the specified ``this`` value and parameters.
 4. Explain the role of the ``bound`` function in the polyfill.
 - Answer: The ``bound`` function is the actual function returned by the polyfill, and it combines the bound arguments with new arguments when invoked.
 5. Explain the purpose of the ``printName`` function in the code.
 - Answer: The ``printName`` function is a template function that prints the full name, state, and country, using the ``this`` context and additional parameters.
 6. What does the ``bind()`` method do in the line ``let printMyName = printName.bind(name, "Mumbai", "MH")``?
 - Answer: It binds the ``printName`` function to the ``name`` object, setting the ``this`` value, and pre-filling the parameters with "Mumbai" and "MH".
 7. Describe the output of ``printMyName("India")`` in the code.
 - Answer: The output will be "Priya Bagde MH India" because the ``printMyName`` function is bound to the ``name`` object with additional parameters.
 8. What is a polyfill in JavaScript?
 - Answer: A polyfill is a piece of code that provides functionality that is not natively supported by a web browser.
 9. Why would you need a polyfill for the ``bind()`` method?
 - Answer: Some older browsers may not support the ``bind()`` method, so a polyfill ensures consistent behavior across different environments.
 10. Explain the purpose of the ``mybind`` method in the polyfill.
 - Answer: The ``mybind`` method is a custom implementation of the ``bind()`` method, allowing users to set the ``this`` value and pre-fill parameters.
 11. How does the ``mybind`` method achieve its goal in the polyfill?
 - Answer: It returns a new function that calls the original function with the specified ``this`` value and a combination of pre-filled and provided parameters.
 12. What does ``Function.prototype`` refer to in JavaScript?
 - Answer: It refers to the prototype object for all function instances, allowing you to add properties and methods that are shared across all functions.

13. Explain the line ``let obj = this`` in the polyfill.

- Answer: It stores a reference to the original function (the object to be bound) in the variable ``obj`` for later use.

14. What does ``params = args.slice(1)`` do in the polyfill?

- Answer: It extracts the parameters (excluding the ``this`` value) provided to the ``mybind`` method and stores them in the ``params`` array.

15. How does the polyfill handle multiple invocations of the bound function?

- Answer: The polyfill creates a new function each time ``mybind`` is called, ensuring that each bound function has its own set of parameters and ``this`` value.

16. Explain the line ``obj.apply(args1[0], [...params, ...args2])`` in the polyfill.

- Answer: It invokes the original function (``obj``) with a specified ``this`` value (``args1[0]``) and a combination of pre-filled parameters, followed by any additional parameters passed during the invocation.

17. What happens if you call ``mybind()`` without any arguments?

- Answer: It would result in an error because ``args[0]`` would be undefined, and the polyfill expects a valid ``this`` value.

18. How would you modify the polyfill to handle missing ``this`` value gracefully?

- Answer: You can add a check for the existence of ``args[0]`` and provide a default value if it is undefined.

19. How does the polyfill handle arguments when the bound function is called?

- Answer: It concatenates the bound arguments with the new arguments provided during the function call.

20. What are the advantages of using the ``bind()`` method over the polyfill?

- Answer: The native ``bind()`` method is widely supported, efficient, and easier to use compared to a custom polyfill. It also provides additional features like partial application and function composition.

21. Can you use the polyfill with functions that have a variable number of arguments?

- Answer: Yes, the polyfill can be used with functions that have a variable number of arguments, as it dynamically combines pre-filled and provided parameters.

22. How would you modify the polyfill to support a variable number of arguments?

- Answer: You can use the ``arguments`` object within the returned function to capture all arguments dynamically.

23. What happens if you call ``bind()`` on a function that is already bound?

- Answer: The `.bind()` method can be called on a function that is already bound, but it will create a new function with the specified `this` value and parameters.

24. Explain the use of `...args1` in the polyfill.

- Answer: It is the spread syntax, used to pass an array of arguments to the `apply` method, allowing for a dynamic number of parameters.

25. What is the purpose of the line `let printMyName2 = printName.mybind(name, "Mumbai", "MH")` in the code?

- Answer: It demonstrates the usage of the custom polyfill (`mybind`) with the `printName` function, binding it to the `name` object and pre-filling parameters.

26. How would you modify the polyfill to support a different context for `this` dynamically?

- Answer: You can make the polyfill accept the desired `this` value as an argument during the invocation instead of hardcoding it.

27. Can you use the `.bind()` method with arrow functions?

- Answer: No, arrow functions do not have their own `this` value, so the `.bind()` method is not applicable to them.

28. Explain the potential memory implications of using the polyfill multiple times.

- Answer: Each call to the polyfill creates a new function, potentially leading to increased memory usage if used excessively. It's important to manage function instances appropriately.

29. What are some alternative ways to achieve function binding in JavaScript?

- Answer: Other methods include using the `call()` and `apply()` methods or creating wrapper functions that explicitly set the `this` value before invoking the original function.

30. Is the polyfill compatible with ES6 classes and methods?

- Answer: Yes, the polyfill can be used with ES6 classes and methods.

31. What would happen if you didn't use `apply` in the polyfill and just called `originalFunction(context, newArguments)`?

- Answer: Without `apply`, the arguments would not be passed correctly, and it might not behave as expected.

32. What is the significance of `Function.prototype.bind` in the polyfill?

- Answer: It extends the prototype of the `Function` object, making the `bind` method available for all functions.

33. How would you apply the polyfill to a specific function in your code?

- Answer: By checking if `Function.prototype.bind` is not already defined and applying the polyfill if needed.

34. Is it possible to change the `this` value of an arrow function using the polyfill?
- Answer: No, arrow functions have a fixed `this` value and cannot be changed using `bind`.
35. Can you use the polyfill in an asynchronous context?
- Answer: Yes, the polyfill works in both synchronous and asynchronous contexts.
36. Can you use the polyfill for functions with the `new` keyword?
- Answer: The polyfill is not suitable for functions used as constructors with the `new` keyword.
37. What is the impact of the polyfill on the prototype chain of functions?
- Answer: The polyfill does not affect the prototype chain of functions.

Part 1:

https://www.linkedin.com/posts/priya-bagde_part1-frontend-javascript-activity-7153331959824330753-8oUH?utm_source=share&utm_medium=member_desktop

Part 2:

https://www.linkedin.com/posts/priya-bagde_part2-frontend-javascript-activity-7153332995796434945-uPvW?utm_source=share&utm_medium=member_desktop

Part 3:

https://www.linkedin.com/posts/priya-bagde_part3-frontend-javascript-activity-7153333414849347584-rIBI?utm_source=share&utm_medium=member_desktop

Part 4:

https://www.linkedin.com/posts/priya-bagde_part4-frontend-javascript-activity-7153333978718973953-db63?utm_source=share&utm_medium=member_desktop

=====

"What's the differences between Promise.any, Promise.race, Promise.all, and Promise.allSettled with real-time examples."

In short, use

- 👉 **Promise.any** for quick responses,
- 👉 **Promise.race** for speed-centric scenarios,
- 👉 **Promise.all** when all promises must succeed, and
- 👉 **Promise.allSettled** when you want to capture the outcome of all promises, regardless of success or failure.

Taking a moment to collect my thoughts, I explained,

"Absolutely, let's dive into the Promise methods. Firstly, Promise.any resolves as soon as any of the promises it receives resolves, rejecting only if all promises are rejected. This is particularly handy when dealing with multiple APIs, and we want to proceed as soon as we get the first

successful response.

On the other hand, `Promise.race` resolves or rejects as soon as any of the promises resolves or rejects. This is useful when speed is crucial, like when fetching data from multiple sources and taking the result from the fastest one.

Now, for scenarios where we need all promises to succeed before proceeding, we have `Promise.all`. It resolves only if all promises resolve, and it rejects if any of them fail. This is valuable when aggregating data from various sources and ensuring a comprehensive success criterion.

Lastly, `Promise.allSettled` is beneficial when we want to wait for all promises to settle, regardless of whether they resolve or reject. It returns an array of result objects, each indicating the status and value. This method is particularly useful for scenarios where we need to process all outcomes, successful or not, such as logging API responses for auditing purposes."

=====

Memoisation:

Write a memoization for computing the Fibonacci series".

Taking a moment to collect my thoughts, I explained the below points, but did i missed any other point here.

📌 Memoization is a performance optimization technique where the results of expensive function calls are **cached and reused** when the same inputs occur again.

📌 This helps to avoid **redundant computations**, improving the overall efficiency of the program.

📌 The key is to store the results of function calls based on their inputs and check the cache before performing a potentially expensive computation.

📌 There are different ways to implement memoization, including manual **caching with objects**, **using higher-order functions**, **using closures**, or **leveraging built-in features like the Map object**.

📌 For example, consider a function that computes the **nth Fibonacci number**. Without memoization, the function might repeatedly calculate the same Fibonacci numbers, leading to unnecessary overhead. By implementing memoization, we can store previously computed results in a cache, preventing redundant calculations.

```
// Function with memoization for Fibonacci
function fibonacciWithMemoization(n, memo = {}) {
  // Check if the result is already memoized
  if (memo[n] !== undefined) {
    return memo[n];
  }
}
```



```
// Base cases
if (n <= 1) {
  return n;
}

// Recursive calls with memoization
memo[n] = fibonacciWithMemoization(n - 1, memo) + fibonacciWithMemoization(n - 2, memo);

return memo[n];
}

// Example usage
const result = fibonacciWithMemoization(5);
console.log(result); // Output: 5
```