- Write code to add class to div ?

```
// Select the div element
var element = document.getElementById("myDiv");

// Add a class
element.classList.add("newClass");
```

---

67. Is javascript synchronous or asynchronous and single threaded or multithreaded ?

- Javascript is a **synchronous single threaded language**. This means that it executes line by line in order and each line must finish executing before the next line starts.
- However, javascript has can handle asynchronous operations using mechanisms like callbacks, promises and async/await. These mechanisms allows javascript to perform tasks such as network requests, file reading, setTimeout/setInterval without blocking the main thread.
- These mechanisms allow JavaScript to delegate tasks to the browser and then continue executing other code while waiting for those tasks to complete. This asynchronous behavior gives the illusion of concurrency, even though JavaScript itself remains single-threaded.

---

36 Output based questions:

1. What is the output of 3+2+"7" ?

- "57"

Explaination:

- In javascript, the addition associativity is from left to right. Due to this initially 3+2 will be evaluated and it will become 5. Now expression becomes 5+"7".
- In javascript when ever we try to perform addition between a numeric value and a string, javascript will perform type coercion and convert numeric value in to a string.
- Now expression becomes "5"+ "7" this performs string concatenation and results in "57".

---

2. What is the output of below logic ?

```
const a = 1<2<3;
const b = 1>2>3;

console.log(a,b) //true,false
```

Output:

- true, false
  - In JavaScript, the comparison operators `<` and `>` have left-to-right associativity. So, `1 < 2 < 3` is evaluated as `(1 < 2) < 3`, which becomes `true < 3`. When comparing a boolean value (`true`) with a number (`3`), JavaScript coerces the boolean to a number, which is `1`. So, `true < 3` evaluates to `1 < 3`, which is `true`.
  - Similarly, `1 > 2 > 3` is evaluated as `(1 > 2) > 3`, which becomes `false > 3`. When comparing a boolean value (`false`) with a number (`3`), JavaScript coerces the boolean to a number, which is `0`. So, `false > 3` evaluates to `0 > 3`, which is `false`.
  - That's why `console.log(a, b)` prints `true false`.

---

3. Guess the ouput ?

```
const p = { k: 1, l: 2 };
 const q = { k: 1, l: 2 };
 let isEqual = p==q;
 let isStartEqual = p=== q;

console.log(isEqual, isStartEqual)
```

- **OutPut:**
  - False,False

  In JavaScript, when you compare objects using `==` or `===`, you're comparing their references in memory, not their actual contents. Even if two objects have the same properties and values, they are considered unequal unless they reference the exact same object in memory.

  In your code:

  - `isEqual` will be `false` because `p` and `q` are two different objects in memory, even though they have the same properties and values.
  - `isStartEqual` will also be `false` for the same reason. The `===` operator checks for strict equality, meaning it not only compares values but also ensures that the objects being compared reference the exact same memory location.

  So, `console.log(isEqual, isStartEqual)` will output `false false`.

---

4. Guess the output ?

```
a) 2+2 = ?
b) "2"+"2" = ?
c) 2+2-2 = ?
d) "2"+"2"-"2" = ? (tricky remember this)
e) 4+"2"+2+4+"25"+2+2 ?
```

- **Output:**

```
a) 2+2 = ?
console.log(2 + 2); // Output: 4

b) "2"+"2" = ?
console.log("2" + "2");
// Output: "22" (string concatenation)

c) 2+2-2 = ?
console.log(2 + 2 - 2); // Output: 2

d) "2"+"2"-"2" = ?
console.log("2" + "2" - "2");
// Output: 20 (string "22" is converted
 to a number, then subtracted by the number 2)

e) 4+"2"+2+4+"25"+2+2
console.log(4+"2"+2+4+"25"+2+2);
// "42242522"
```

---

5. What is the output of below logic ?

```
let a = 'jscafe'
a[0] = 'c'

console.log(a)
```

- **Output:**
  - “jscafe”
  - Strings are immutable in javascript so we cannot change individual characters by index where as we can create a new string with desired

modification as below.

- a = “cscafe” // outputs “cscafe”

---

6. Output of below logic ?

```
var x=10;
function foo(){
var x = 5;
console.log(x)
}

foo();
console.log(x)
```

Output: 5 and 10

In JavaScript, this code demonstrates variable scoping. When you declare a variable inside a function using the `var` keyword, it creates a new variable scoped to that function, which may shadow a variable with the same name in an outer scope. Here's what happens step by step:

1. `var x = 10;`: Declares a global variable `x` and initializes it with the value `10`.
2. `function foo() { ... }`: Defines a function named `foo`.
3. `var x = 5;`: Inside the function `foo`, declares a local variable `x` and initializes it with the value `5`. This `x` is scoped to the function `foo` and is different from the global `x`.
4. `console.log(x);`: Logs the value of the local variable `x` (which is `5`) to the console from within the `foo` function.
5. `foo();`: Calls the `foo` function.
6. `console.log(x);`: Logs the value of the global variable `x` (which is still `10`) to the console outside the `foo` function.

7. Guess the output ?

```
console.log("Start");
setTimeout(() => {
 console.log("Timeout");
});
Promise.resolve().then(() => console.log("Promise"));
console.log("End");
```

Output:

- Start, End,Promise,Timeout.
  - "Start" is logged first because it's a synchronous operation.
  - Then, "End" is logged because it's another synchronous operation.
  - "Promise" is logged because `Promise.resolve().then()` is a microtask and will be executed before the next tick of the event loop.
  - Finally, "Timeout" is logged. Even though it's a setTimeout with a delay of 0 milliseconds, it's still a macrotask and will be executed in the next tick of the event loop after all microtasks have been executed.

8. This code prints 6 everytime. How to print 1,2,3,4,5,6 ? (Most asked)

```
function x(){

 for(var i=1;i<=5;i++){
```

```
    setTimeout(()⇒{
      console.log(i)
    },i*1000)
  }

}

x();
```

Solution: Either use let or closure

```
function x() {
  function closur(x) {
    // Set a timeout to log value of x after x seconds
    setTimeout(() ⇒ {
      console.log(x);
    }, x * 1000);
  };

  // Loop from 1 to 5
  for (var i = 1; i <= 5; i++) {
    // Call the closure function with current value of i
    closur(i);
  }
}

// Call the outer function x
x();
```

The function we have written defines an inner function `closur` which is supposed to log the value of `x` after `x` seconds. The outer function `x` calls this inner function for values from 1 to 5.

The code will log the values 1 to 5 after 1 to 5 seconds respectively. Here's an explanation of how it works:

1. The outer function `x` is called.
2. Inside `x`, a loop runs from `i=1` to `i=5`.
3. For each iteration of the loop, the inner function `closur` is called with the current value of `i`.
4. Inside `closur`, a `setTimeout` is set to log the value of `x` after `x` seconds.

Each call to `closur(i)` creates a new closure that captures the current value of `i` and sets a timeout to log that value after `i` seconds.

When you run this code, the output will be:

```
1 (after 1 second)
2 (after 2 seconds)
3 (after 3 seconds)
4 (after 4 seconds)
5 (after 5 seconds)
```

This happens because each iteration of the loop calls `closur` with a different value of `i`, and each `setTimeout` inside `closur` is set to log that value after `i` seconds.

---

9. What will be the output or below code ?

```
function x(){
  let a = 10;
  function d(){
    console.log(a);
  }
  a = 500;
  return d;
}
```

```
var z = x();
z();
```

Solution: 500 - Closures concept

In JavaScript, this code demonstrates lexical scoping and closure. Let's break it down:

1. `function x() { ... }`: Defines a function named `x`.
2. `let a = 10;`: Declares a variable `a` inside the function `x` and initializes it with the value `10`.
3. `function d() { ... }`: Defines a nested function named `d` inside the function `x`.
4. `console.log(a);`: Logs the value of the variable `a` to the console. Since `d` is defined within the scope of `x`, it has access to the variable `a` defined in `x`.
5. `a = 500;`: Changes the value of the variable `a` to `500`.
6. `return d;`: Returns the function `d` from the function `x`.
7. `var z = x();`: Calls the function `x` and assigns the returned function `d` to the variable `z`.
8. `z();`: Calls the function `d` through the variable `z`.

When you run this code, it will log the value of `a` at the time of executing `d`, which is `500`, because `d` retains access to the variable `a` even after `x` has finished executing. This behavior is possible due to closure, which allows inner functions to access variables from their outer scope even after the outer function has completed execution.

---

10. What's the output of below logic ?

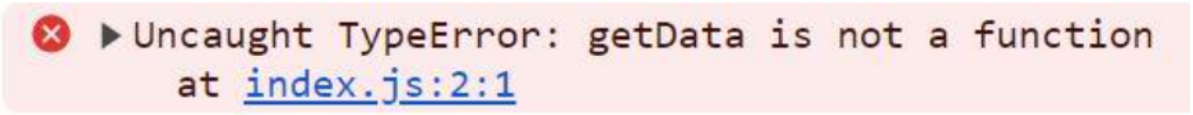
```
getData1()
getData();
```

```
function getData1(){
 console.log("getData11")
}

var getData = () => {
 console.log("Hello")
}

 /* Here declaring getData with let causes
 reference error.ie.,"ReferenceError: Cannot
 access 'getData' before initialization As we
 are declaring with var it throws type error
 as show below */
```

Output:



Explanation:

In JavaScript, function declarations are hoisted to the top of their scope, while variable declarations using `var` are also hoisted but initialized with `undefined`. Here's what happens in your code:

1. `getData1()` is a function declaration and `getData()` is a variable declaration with an arrow function expression assigned to it.
2. When the code runs:
    - `getData1()` is a function declaration, so it's hoisted to the top and can be called anywhere in the code. However, it's not called immediately.
    - `getData` is declared using `var`, so it's also hoisted to the top but initialized with `undefined`.
    - The arrow function assigned to `getData` is not hoisted because it's assigned to a variable.

3. When `getData()` is invoked:
    - It will throw an error because `getData` is `undefined`, and you cannot call `undefined` as a function.

Therefore, if you try to run the code as is, you'll encounter an error when attempting to call `getData()`.

If you want to avoid this error, you should either define `getData` before calling it or use a function declaration instead of a variable declaration for `getData`. Here's how you can do it:

Modification needed for code:

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

getData1(); // This will log "getData11"
getData();  // This will log "Hello"
```

---

11. Whats the output of below code ?

```
function func() {
  try {
    console.log(1)
    return
  } catch (e) {
    console.log(2)
  } finally {
    console.log(3)
  }
  console.log(4)
}
```

```
func()
```

Output: 1 & 3

1. The function `func()` is defined.
2. Inside the `try` block:
    - `console.log(1)` is executed, printing `1` to the console.
    - `return` is encountered, which immediately exits the function.
3. The `finally` block is executed:
    - `console.log(3)` is executed, printing `3` to the console.

Since `return` is encountered within the `try` block, the control exits the function immediately after `console.log(1)`. The `catch` block is skipped because there are no errors, and the code in the `finally` block is executed regardless of whether an error occurred or not.

So, when you run this code, it will only print `1` and `3` to the console.

---

12. What's the output of below code ?

```
const nums = [1,2,3,4,5,6,7];
nums.forEach((n) => {
   if(n%2 === 0) {
    break;
   }
   console.log(n);
});
```

Explanation:

Many of you might have thought the output to be *1,2,3,4,5,6,7.* But "break" statement works only loops like for, while, do...while and not for map(),

forEach(). They are essentially functions by nature which takes a callback and not loops.

```
Uncaught SyntaxError: Illegal break statement (at Script snippet #5:4
Script snippet #5:4:6)
```

---

13. Whats the output of below code ?

```
let a = true;
setTimeout(() => {
  a = false;
}, 2000)


while(a) {
  console.log(' -- inside whilee -- ');
}
```

Solution: https://medium.com/@iamyashkhandelwal/5-output-based-interview-questions-in-javascript-b64a707f34d2

This code snippet creates an infinite loop. Let's break it down:

1. `let a = true;`: This declares a variable `a` and initializes it to `true`.
2. `setTimeout(() => { a = false; }, 2000)`: This sets up a timer to execute a function after 2000 milliseconds (2 seconds). The function assigned to `setTimeout` will set the value of `a` to `false` after the timeout.
3. `while(a) { console.log(' -- inside whilee -- '); }`: This is a while loop that continues to execute as long as the condition `a` is `true`. Inside the loop, it prints `' -- inside whilee -- '`.

The issue here is that the while loop runs indefinitely because there's no opportunity for the JavaScript event loop to process the setTimeout callback

and update the value of `a`. So, even though `a` will eventually become `false` after 2 seconds, the while loop will not terminate because it doesn't yield control to allow other tasks, like the callback, to execute.

To fix this, you might consider using asynchronous programming techniques like Promises, async/await, or handling the setTimeout callback differently.

---

14. Whats the output of below code ?

```
setTimeout(() => console.log(1), 0);

console.log(2);

new Promise(res => {
  console.log(3)
  res();
}).then(() => console.log(4));

console.log(5);
```

This code demonstrates the event loop in JavaScript. Here's the breakdown of what happens:

1. `setTimeout(() => console.log(1), 0);`: This schedules a callback function to be executed after 0 milliseconds. However, due to JavaScript's asynchronous nature, it doesn't guarantee that it will execute immediately after the current synchronous code block.
2. `console.log(2);`: This immediately logs `2` to the console.
3. `new Promise(res => { console.log(3); res(); }).then(() => console.log(4));`: This creates a new Promise. The executor function inside the Promise logs `3` to the console and then resolves the Promise immediately with `res()`. The `then()` method is chained to the Promise, so once it's resolved, it logs `4` to the console.
4. `console.log(5);`: This logs `5` to the console.

When you run this code, the order of the output might seem a bit counterintuitive:

```
2
3
5
4
1
```

Here's why:

- `console.log(2);` is executed first because it's synchronous code.
- Then, the Promise executor is executed synchronously, so `console.log(3);` is logged.
- After that, `console.log(5);` is executed.
- Once the current synchronous execution is done, the event loop picks up the resolved Promise and executes its `then()` callback, logging `4`.
- Finally, the callback passed to `setTimeout` is executed, logging `1`. Although it was scheduled to run immediately with a delay of 0 milliseconds, it's still processed asynchronously and placed in the event queue, after the synchronous code has finished executing.

https://medium.com/@iamyashkhandelwal/5-output-based-interview-questions-in-javascript-b64a707f34d2

---

15. Output of below logic ?

```
async function foo() {
  console.log("A");
  await Promise.resolve();
  console.log("B");
  await new Promise(resolve => setTimeout(resolve, 0));
```

```
  console.log("C");
}

console.log("D");
foo();
console.log("E")
```

Output:

D, A, E, B, C

Explanation:

The main context logs “D” because it is synchronous and executed immediately.

The foo() function logs "A" to the console since it's synchronous and executed immediately. await Promise.resolve() : This line awaits the resolution of a Promise. The Promise.resolve() function returns a resolved Promise immediately. The control is temporarily returned to the caller function (foo()), allowing other synchronous operations to execute.

Back to the main context: console.log("E") : This line logs "E" to the console since it's a synchronous operation. The foo()

function is still not fully executed, and it's waiting for the resolution of the Promise inside it. Inside foo()

(resumed execution): console.log("B") : This line logs "B" to the console since it's a synchronous operation.

await new Promise(resolve ⇒ setTimeout(resolve, 0));

This line awaits the resolution of a Promise returned by the setTimeout function. Although the delay is set to 0 milliseconds, the setTimeout callback is pushed into the callback queue, allowing the synchronous code to continue.

Back to the main context: The control is still waiting for the foo() function to complete.

Inside foo() (resumed execution): The callback from the setTimeout is picked up from the callback queue, and the promise is resolved. This allows the execution of the next await .

console.log("C") : This line logs "C" to the console since it's a synchronous operation. foo() function completes.

---

16. Guess the output ?

```
let output = (function(x){
    delete x;
    return x;
})(3);
console.log(output);
```

Output: 3

Let me break it down for you:

1. The code defines an immediately invoked function expression (IIFE) that takes a parameter `x`.
2. Inside the function, `delete x;` is called. However, `delete` operator is used to delete properties from objects, not variables. When you try to delete a variable, it doesn't actually delete the variable itself, but it's syntactically incorrect and may not have any effect depending on the context (in strict mode, it throws an error). So, `delete x;` doesn't do anything in this case.
3. Finally, the function returns `x`. Since `x` was passed as `3` when calling the function `(function(x){ ... })(3)`, it returns `3`.
4. The returned value is assigned to the variable `output`.
5. `console.log(output);` then logs the value of `output`, which is `3`.

---

17. Guess the output of below code ?

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

Output: 3 3 3

This might seem counterintuitive at first glance, but it's due to how JavaScript handles closures and asynchronous execution.

Here's why:

1. The `for` loop initializes a variable `i` to `0`.
2. It sets up a timeout for `i` milliseconds plus the current value of `i`, which means the timeouts will be `1000`, `1001`, and `1002` milliseconds.
3. After setting up the timeouts, the loop increments `i`.
4. The loop checks if `i` is still less than `3`. Since it's now `3`, the loop exits.

When the timeouts execute after their respective intervals, they access the variable `i` from the outer scope. At the time of execution, `i` is `3` because the loop has already finished and incremented `i` to `3`. So, all three timeouts log `3`.

---

18. Guess the output ?

```
let output = (function(x){
    delete x;
    return x;
})(3);
console.log(output);
```

Output: 3

Let me break it down for you:

1. The code defines an immediately invoked function expression (IIFE) that takes a parameter `x`.
2. Inside the function, `delete x;` is called. However, `delete` operator is used to delete properties from objects, not variables. When you try to delete a variable, it doesn't actually delete the variable itself, but it's syntactically incorrect and may not have any effect depending on the context (in strict mode, it throws an error). So, `delete x;` doesn't do anything in this case.
3. Finally, the function returns `x`. Since `x` was passed as `3` when calling the function `(function(x){ ... })(3)`, it returns `3`.
4. The returned value is assigned to the variable `output`.
5. `console.log(output);` then logs the value of `output`, which is `3`.

---

19. Guess the output ?

```
let c=0;

let id = setInterval(() => {
   console.log(c++)
},10)

setTimeout(() => {
   clearInterval(id)
},2000)
```

This JavaScript code sets up an interval that increments the value of `c` every 200 milliseconds and logs its value to the console. After 2 seconds (2000 milliseconds), it clears the interval.

Here's what each part does:

- `let c = 0;`: Initializes a variable `c` and sets its initial value to 0.

- `let id = setInterval(() => { console.log(c++) }, 200)`: Sets up an interval that executes a function every 200 milliseconds. The function logs the current value of `c` to the console and then increments `c`.
- `setTimeout(() => { clearInterval(id) }, 2000)`: Sets a timeout function that executes after 2000 milliseconds (2 seconds). This function clears the interval identified by `id`, effectively stopping the logging of `c`.

This code essentially logs the values of `c` at 200 milliseconds intervals until 2 seconds have passed, at which point it stops logging.

---

20. What would be the output of following code ?

```
function getName1(){
   console.log(this.name);
}

Object.prototype.getName2 = () =>{
   console.log(this.name)
}

let personObj = {
   name:"Tony",
   print:getName1
}

personObj.print();
personObj.getName2();
```

Output: Tony undefined

Explaination: **getName1()** function works fine because it's being called from ***personObj***, so it has access to *this.name* property. But when while calling **getnName2** which is defined under *Object.prototype* doesn't have any

proprty named *this.name*. There should be *name* property under prototype. Following is the code:

```
function getName1(){
   console.log(this.name);
}

Object.prototype.getName2 = () =>{
 console.log(Object.getPrototypeOf(this).name);
}

let personObj = {
   name:"Tony",
   print:getName1
}

personObj.print();
Object.prototype.name="Steve";
personObj.getName2();
```

---

21. What would be the output of following code ?

```
function test() {
  console.log(a);
  console.log(foo());
  var a = 1;
  function foo() {
     return 2;
  }
}

test();
```

Output: undefined and 2

In JavaScript, this code will result in `undefined` being logged for `console.log(a)` and `2` being logged for `console.log(foo())`. This is due to variable hoisting and function declaration hoisting.

Here's what's happening step by step:

1. The `test` function is called.
2. Inside `test`:
    - `console.log(a)` is executed. Since `a` is declared later in the function, it's hoisted to the top of the function scope, but not initialized yet. So, `a` is `undefined` at this point.
    - `console.log(foo())` is executed. The `foo` function is declared and assigned before it's called, so it returns `2`.
    - `var a = 1;` declares and initializes `a` with the value `1`.

Therefore, when `console.log(a)` is executed, `a` is `undefined` due to hoisting, and when `console.log(foo())` is executed, it logs `2`, the return value of the `foo` function.

---

22. What is the output of below logic ?

```
function job(){
  return new Promise((resolve,reject)⇒{
    reject()
  })
}

let promise = job();

promise.then(()⇒{
  console.log("1111111111")
}).then(()⇒{
  console.log("22222222222")
```

```
}).catch(()=>{
  console.log("3333333333")
}).then(()=>{
  console.log("4444444444")
})
```

In this code, a Promise is created with the `job` function. Inside the `job` function, a Promise is constructed with the executor function that immediately rejects the Promise.

Then, the `job` function is called and assigned to the variable `promise`.

After that, a series of `then` and `catch` methods are chained to the `promise`:

1. The first `then` method is chained to the `promise`, but it is not executed because the Promise is rejected, so the execution jumps to the `catch` method.
2. The `catch` method catches the rejection of the Promise and executes its callback, logging "3333333333".
3. Another `then` method is chained after the `catch` method. Despite the previous rejection, this `then` method will still be executed because it's part of the Promise chain, regardless of previous rejections or resolutions. It logs "4444444444".

So, when you run this code, you'll see the following output:

```
3333333333
4444444444
```

---

23. Guess the output ?

```
var a = 1;

function data() {
```

```
 if(!a) {
   var a = 10;
 }
 console.log(a);
}

data();
console.log(a);
```

Explanation:

```
var a = 1;

function toTheMoon() {
 var a;
 /* var has function scope,Hence
  it's declaration will be hoisted */

 if(!a) {
   a = 10;
 }
 console.log(a);
 // 10 precendence will be given to local scoped variable.
}


toTheMoon();
console.log(a); // 1 refers to the `a` defined at the top.
```

---

24. Tests your array basics

```
function guessArray() {
  let a = [1, 2];
  let b = [1, 2];

  console.log(a == b);
  console.log(a === b);
}

guessArray();
```

In JavaScript, when you compare two arrays using the `==` or `===` operators, you're comparing their references, not their contents. So, even if two arrays have the same elements, they will not be considered equal unless they refer to the exact same object in memory.

In your `guessArray` function, `a` and `b` are two separate arrays with the same elements, but they are distinct objects in memory. Therefore, `a == b` and `a === b` will both return `false`, indicating that `a` and `b` are not the same object.

If you want to compare the contents of the arrays, you'll need to compare each element individually.

---

25. Test your basics on comparision ?

```
let a = 3;
let b = new Number(3);
let c = 3;

console.log(a == b);
console.log(a === b);
console.log(b === c);
```

`new Number()` is a built-in function constructor. Although it looks like a number, it's not really a number: it has a bunch of extra features and is an object.

When we use the `==` operator (Equality operator), it only checks whether it has the same *value*. They both have the value of `3`, so it returns `true`.

However, when we use the `===` operator (Strict equality operator), both value *and* type should be the same. It's not: `new Number()` is not a number, it's an **object**. Both return `false.`

---

26. Guess the output ?

```
var x = 23;
(function(){
  var x = 43;

  (function random(){
    x++;
    console.log(x);
    var x = 21;
  })();
})();
```

Solution:

The provided code snippet demonstrates the behavior of variable hoisting and function scope in JavaScript. Let's analyze the code step-by-step to understand the output:

```
var x = 23;
(function(){
  var x = 43;

  (function random(){
    x++;
```

```
    console.log(x);
    var x = 21;
  })();

})();
```

Breakdown

1. **Global Scope:**

   ```
   var x = 23;
   ```

   - A global variable `x` is declared and initialized with the value `23`.

2. **First IIFE (Immediately Invoked Function Expression):**

   ```
   (function(){
     var x = 43;
     // ...
   })();
   ```

   - A new function scope is created. Inside this function, a local variable `x` is declared and initialized with the value `43`. This `x` shadows the global `x`.

3. **Second IIFE (Nested function, named `random`):**

   ```
   (function random(){
     x++;
     console.log(x);
     var x = 21;
   })();
   ```

   - Another function scope is created inside the first IIFE. The function `random` is invoked immediately.

4. **Inside the `random` function:**

```
x++;
console.log(x);
var x = 21;
```

- Here, variable hoisting comes into play. The declaration `var x = 21;` is hoisted to the top of the function `random`, but not its initialization. Thus, the code is interpreted as:

  ```
  var x;   // x is hoisted, but not initialized
  x++;
  console.log(x);
  x = 21;
  ```

- Initially, `x` is `undefined` because the hoisted declaration of `x` does not include its initialization.
- `x++` attempts to increment `x` when it is still `undefined`. In JavaScript, `undefined++` results in `NaN` (Not a Number).
- Therefore, `console.log(x);` outputs `NaN`.
- After the `console.log` statement, `x` is assigned the value `21`, but this assignment happens after the `console.log` and thus does not affect the output.

Summary

When `random` function is executed, the following sequence occurs:

1. `var x;` (hoisting, `x` is `undefined` at this point)
2. `x++;` (`undefined++` results in `NaN`)
3. `console.log(x);` outputs `NaN`
4. `x = 21;` (assigns `21` to `x`, but this is after the `console.log`)

Output

Thus, the output of the code is:

```
NaN
```

---

27. Answer below queries on typeOf operator in javascript ?

```
typeof [1,2,3,4]     // Returns object
typeof null          // Returns object
typeof NaN           // Returns number
typeof 1234n         // Returns bigint
typeof 3.14          // Returns number
typeof Symbol()      // Returns symbol

typeof "John"        // Returns string
typeof 33            // Returns number
typeof true          // Returns boolean
typeof undefined     // Returns undefined
```

---

28. Can you find is there any security issue in the javascript code?

```
const data = await fetch("api");
const div = document.getElementById("todo")
div.innerHTML = data;
```

The provided JavaScript code seems straightforward, but there's a potential security issue related to how it handles data from the API response.

1. **Cross-Site Scripting (XSS)**:
   The code directly assigns the fetched data (`data`) to the `innerHTML` property of the `div` element. If the data fetched from the API contains untrusted or user-controlled content (such as user-generated content or content from a third-party API), it could potentially contain malicious scripts. Assigning such data directly to `innerHTML` can lead to XSS vulnerabilities, as it allows execution of arbitrary scripts in the context of the page.

To mitigate this security risk, you should properly sanitize or escape the data before assigning it to `innerHTML` , or consider using safer alternatives like `textContent` or creating DOM elements programmatically.

Here's an example of how you could sanitize the data using a library like DOMPurify:

```
const data = await fetch("api");
const div = document.getElementById("todo");
data.text().then(text => {
   div.innerHTML = DOMPurify.sanitize(text);
});
```

By using `DOMPurify.sanitize()` , you can ensure that any potentially harmful content is removed or escaped, reducing the risk of XSS attacks. Make sure to include the DOMPurify library in your project if you choose to use it.

Always remember to validate and sanitize any data that originates from external sources before inserting it into your DOM.

---

29. Can you guess the output for below code ?

```
console.log(typeof typeof 1)
```

In the above code, we are using typeof operator. So what is this typeof ?

- typeof is an operator in javascript which will return the data type of the operand which is passed to it
- In the above example, Initially typeof 1 will be executed which will return “number”.
- Now the expression will become typeof “number” which will result in “string”.

---

30. Guess the output of below code ?

```
x++;
console.log(x);
var x = 21;
```

Explaination:

- In this example, First we need to understand how variables declared with var are hoisted ?
  - In javascript variables declared with var are hoisted on to the top of their scope and initialized with the value of undefined. Due to this, in the first line, value of x is undefined.
- In the first line we are using postfix operator. so javascript will perform type cohercion and tries to convert this undefined into a number.
- As a result, value of x will become Nan and in the 2nd line x value ie., Nan is logged to the console.

31. Guess the output ?

```
const x = [1];
const y = [2];
console.log(x+y);
```

Explanation:

1. **Variable Declaration:**
   - `const x = [1];` declares a constant variable `x` and assigns it an array containing a single element: the number 1.
   - `const y = [2];` declares another constant variable `y` and assigns it an array containing the number 2.
2. **Array Concatenation (Attempt):**
   - `x+y` attempts to concatenate the arrays `x` and `y`.
   - **However, the `+` operator performs string concatenation when used with arrays in JavaScript.**
3. **Implicit Type Conversion:**
   - JavaScript implicitly converts the arrays `x` and `y` to strings.
   - The array `[1]` is converted to the string `"1"`.
   - The array `[2]` is converted to the string `"2"`.
4. **String Concatenation:**
   - The `+` operator then concatenates the strings `"1"` and `"2"`, resulting in the string `"12"`.
5. **Console Output:**
   - `console.log(x+y);` prints the resulting string `"12"` to the console.

Key Points:

- The `+` operator does not perform array concatenation in JavaScript.
- When used with arrays, the `+` operator implicitly converts the arrays to strings and then concatenates them.

---

32. Guess the output ?

```
const data = {
   name: "sai",
   name: "krishna"
}

console.log(data.name);
```

Output: “krishna”.

Explanation:

In JavaScript, if an object has multiple properties with the same name, the last one defined in the object will overwrite the previous ones. This behavior occurs because object properties in JavaScript are treated as key-value pairs, where keys must be unique.

---

33. What is the output of below code ?

```
let x = 10+2*3
console.log(x);
```

Output: 16

Video Explaination: https://www.instagram.com/p/DHqg2pXCXmw/

Explaination:

- The expression `10 + 2 * 3` is evaluated according to operator precedence rules in JavaScript. Multiplication (`*`) has a higher precedence than addition (`+`), so the multiplication is performed first:
  - First, compute `2 * 3` which equals `6`.
  - Then, add `10` to `6`, resulting in `16`.

---

34. Guess the output ?

```
const x = [1,2,3];
const y = [1,3,4];
console.log(x+y);
```

Output: “1,2,31,3,4”;

Video Explaination: https://www.instagram.com/p/DHoMoC0ii_x/

Explaination:

Here we are defining two arrays:

- `x` is `[1, 2, 3]`
- `y` is `[1, 3, 4]`

When you use the `+` operator with arrays, JavaScript converts each array to a string. Under the hood, this conversion happens by calling the `toString()` method on the arrays. For example:

- `x.toString()` becomes `"1,2,3"`
- `y.toString()` becomes `"1,3,4"`

Then, the `+` operator concatenates these two strings together. So the result of `x + y` is the string:

Finally, `console.log` prints "1,2,31,3,4" this string to the console.

35. Guess the output ?

```
console.log(+true);
console.log(!"sai");
```

Output: 1,false

Video Explaination:https://www.instagram.com/p/DHlWUMaChl9/

Explaination:

`console.log(+true);`

- The unary plus operator (`+`) converts its operand into a number.
- `true` is converted to `1`.
- **Result:** This prints `1` to the console.

`console.log(!"sai");`

- The logical NOT operator (`!`) converts its operand to a boolean and then negates it.
- The string `"sai"` is non-empty, which makes it a truthy value.
- Negating a truthy value results in `false`.
- **Result:** This prints `false` to the console.

In summary:

- `+true` **becomes** `1`.
- `!"sai"` **becomes** `false`.

36. What is the output of below code ?

```
console.log([]+[]);
console.log([1]+[]);
console.log([1]+"abc");
```

Output: “”,”1”,”1abc”

Video Explaination: https://www.instagram.com/p/DHixQfaiwKE/

Explaination:

When you use the `+` operator with arrays, JavaScript first converts the arrays into strings using their `toString()` method:

- `[] + []`

  An empty array (`[]`) converts to an empty string (`""`). So, `"" + ""` results in an empty string.

- `[1] + []`

  The array `[1]` converts to `"1"`, and `[]` converts to `""`. So, `"1" + ""` results in `"1"`.

- `[1] + "abc"`

  Again, `[1]` converts to `"1"`. Then `"1" + "abc"` concatenates to give `"1abc"`.

In summary, the arrays are turned into strings before they are concatenated.

---

37. Guess the output ?

```
function getAge(...args){
  console.log(typeof args)
}

getAge(21)
```

Output: “object”

Video Explaination: https://www.instagram.com/p/DHdo1CLi4ud/

Explaination:

The output of the code will be **"object"**. This is because :

1. **Rest Parameters (`...args`):**

   The `...args` in the function definition is JavaScript's **rest parameter syntax**. It collects all passed arguments into a single array. So even though you pass `21`, `args` becomes `[21]` (an array).

2. `typeof` **Behavior:**

   In JavaScript, arrays are technically a type of **object**. When you use `typeof` on an array (like `args`), it returns `"object"`, not `"array"`.

---

38. Guess the output ?

```
const obj = {
  a: "one",
  b: "two",
  a: "three"
}

console.log(obj);
```

Output:

{

a: "three",

b: "two"

}

Video Explaination: https://www.instagram.com/p/DHDwosvCVdx/

Explaination:

This is because.,

1. **Duplicate Keys in Objects:**

   In JavaScript, if an object has **duplicate keys**, the **last occurrence** of the key will overwrite the previous value. This happens silently (no error is thrown).

2. **What Happens Here:**

   - The key `a` is first assigned `"one"`.
   - Then, `a` is redefined with `"three"` (overwriting the first `a`).
   - The key `b` stays `"two"`.

```
const obj = {
  a: "one",    // overwritten by the next 'a'
  b: "two",
  a: "three"   // this is the final value of 'a'
};
```

---

39. Guess the output ?

```
var z = 1, y = z = typeof y;
console.log(y);
```

Output: `"undefined"`

Video Explaination: https://www.instagram.com/p/DGlAOTsimyX/

Explaination:

The output of the code will be `"undefined"`. Here's why:

Step-by-Step Breakdown:

1. **Variable Hoisting**:

   Variables `z` and `y` are declared (due to `var` hoisting) but not yet initialized. At this stage:

- `z` is `undefined` (temporarily).
- `y` is `undefined` (temporarily).

2. **Initial Assignments**:

```
va = 1; // z is now assigned `1`.
```

3. **The Tricky Line**:

```
y = z = typeof y; // Evaluated from right-to-left!
```

   - **Step 1**: `typeof y` is evaluated.

     Since `y` is declared (due to hoisting) but **not yet initialized**, `typeof y` returns `"undefined"`.

   - **Step 2**: Assign `"undefined"` to `z`.

     Now, `z = "undefined"` (overwriting its earlier value of `1`).

   - **Step 3**: Assign `z`'s value (`"undefined"`) to `y`.

     Now, `y = "undefined"`.

4. **Final Result**:

   `console.log(y);` → Outputs `"undefined"`.

Key Takeaways:

- **Hoisting**: Variables declared with `var` are hoisted and initialized to `undefined` before assignment.
- **Assignment Order**: `a = b = c` is evaluated right-to-left (`b = c` first, then `a = b`).
- **Overwriting**: The initial value of `z` (`1`) is overwritten by `typeof y`.

---

40. Guess the output ?

```
console.log(false || null || "Hello");
console.log(false && null && "Hello");
```

Video Explaination: https://www.instagram.com/p/DGYC5Kqiwm1/

Explaination:

The outputs will be:

`"Hello"` and `false`.

Explanation:

1. First Line: `false || null || "Hello"`

- **Logical OR (`||`)** returns the **first truthy value** it finds.
- Check each value left-to-right:
  - `false` → falsy → move to next.
  - `null` → falsy → move to next.
  - `"Hello"` → truthy (non-empty string) → **return "Hello"**.

2. Second Line: `false && null && "Hello"`

- **Logical AND (`&&`)** returns the **first falsy value** it finds.
- Check each value left-to-right:
  - `false` → falsy → **return** `false` **immediately**.

Key Rules:

- `||` **(OR)**: Stops at the first truthy value.
- `&&` **(AND)**: Stops at the first falsy value.
- If all values are checked (e.g., all truthy for `||` or all falsy for `&&`), the last value is returned.

---

41. Guess the output ?

```
const numbers = [1,2,3,4,5];
const [x,...y] = numbers;
console.log(x,y);
```

Video Explaination: https://www.instagram.com/p/DGQT6ioCreI/

Explaination:

The output will be:

`1 [2, 3, 4, 5]`

1. **Array Destructuring**:

   ```
   const [x, ...y] = [1, 2, 3, 4, 5];
   ```

   - `x` is assigned the **first element** of the array (`1`).
   - `...y` uses the **rest operator** (`...`) to collect the **remaining elements** into a new array `y`.

2. **Result**:
   - `x` → `1`
   - `y` → `[2, 3, 4, 5]`

Key Rules:

- The rest operator (`...`) **must always be the last element** in destructuring.
- It captures all remaining elements into an array.

---

42. Guess the output ?

```
const str = "abc"+ + "def";
console.log(str);
```

Video Explaination: https://www.instagram.com/p/DGNt9DOC5C3/

Explaination:

The output will be:

`"abcNaN"`

1. **Code Breakdown**:

```
const str = "abc" + + "def";
```

   - The `+ + "def"` part is key. Let's break it down:
     - The first `+` is a **string concatenation operator**.
     - The second `+` is a **unary plus operator** (attempting to convert `"def"` to a number).
2. **Unary Plus Operation**:
   - `+ "def"` tries to convert the string `"def"` to a number.
   - Since `"def"` is not a valid number, this results in `NaN` (Not-a-Number).
3. **Concatenation**:
   - Now the expression becomes: `"abc" + NaN`.
   - JavaScript converts `NaN` to the string `"NaN"` during concatenation.
   - Final result: `"abc" + "NaN"` → `"abcNaN"`.

Key Takeaway:

- The unary `+` operator converts values to numbers.
- Invalid conversions (like `"def"`) produce `NaN`.
- `NaN` becomes `"NaN"` when concatenated with a string.

---

43. Guess the output ?

```
let newlist = [1].push(2);
console.log(newlist.push(3));
```

Video Explaination: https://www.instagram.com/p/DGLKah0iB9G/

Explaination:

The code will throw an error:

`"TypeError: newlist.push is not a function"`

1. `[1].push(2)`:
    - The `push()` method adds `2` to the array `[1]`, making it `[1, 2]`.
    - **But** `push()` returns the **new length of the array** (not the array itself).
    - So `newlist` is assigned the value `2` (the new length of the array).

```
let newlist = [1].push(2); // newlist = 2 (a number)
```

2. `newlist.push(3)`:
    - `newlist` is `2` (a number), **not an array**.
    - Numbers do **not** have a `push()` method → this causes an error.

Why This Happens:

- `push()` modifies the original array but returns the **length**, not the array.
- We tried to use `push()` on a number (`2`), which is invalid.

Key Takeaway:

`push()` returns the array's **length**, not the array itself. Always work directly with the array variable.

---

44. Guess the output ?

```
console.log(0 || 1);
console.log(1 || 2);
```

```
console.log(0 && 1);
console.log(1 && 2);
```

Video Explaination: https://www.instagram.com/p/DGGCcEUisBu/

Explaination:

The outputs will be:

`1`, `1`, `0`, `2`

1. `console.log(0 || 1);`

- **Logical OR (`||`)** returns the **first truthy value**.
- `0` is falsy → check next value.
- `1` is truthy → **return** `1`.

  Output: `1`

---

2. `console.log(1 || 2);`

- **Logical OR (`||`)** stops at the first truthy value.
- `1` is truthy → **return** `1` **immediately**.

  Output: `1`

---

3. `console.log(0 && 1);`

- **Logical AND (`&&`)** returns the **first falsy value**.
- `0` is falsy → **return** `0` **immediately**.

  Output: `0`

---

4. `console.log(1 && 2);`

- **Logical AND (`&&`)** returns the **last value** if all are truthy.
- `1` is truthy → check next value.
- `2` is truthy → **return** `2`.

  Output: `2`

Key Rules:

- `||` **(OR)**:
  - Returns the first **truthy** value.
  - If all are falsy, returns the last value.
- `&&` **(AND)**:
  - Returns the first **falsy** value.
  - If all are truthy, returns the last value.

---

45. Guess the output ?

```
console.log(data());
var data = function(){
 return "1";
}
```

Video Explaination: https://www.instagram.com/p/DF7sDc8C7Zw/

Explaination:

The code will throw an error:

`TypeError: data is not a function`

Why This Happens:

1. **Variable Hoisting**:

   JavaScript hoists the declaration of `data` (using `var`) to the top of the scope, but **not its initialization**.

   At the time of `console.log(data())`:

   - `data` exists (due to hoisting), but its value is `undefined`.
   - `undefined` is not a function → error when trying to call `data()`.

2. **Execution Order**:

```
console.log(data()); // ❌ Called BEFORE `data` is assigned a function
var data = function() { // Function expression (not hoisted)
  return "1";
};
```

Key Fix:

Use a **function declaration** (which is fully hoisted):

```
console.log(data()); // Works (output: "1")
function data() {    // Function declaration (hoisted)
  return "1";
}
```

Comparison:

Scenario	Hoisting Behavior	Result
`var data = function()`	Only `data` is hoisted (as `undefined`)	Error (not a function)
`function data()`	Entire function is hoisted	Works

Key Takeaway:

Function expressions assigned to variables (`var x = function(){}`) are **not hoisted** as callable functions. Function declarations (`function x(){}`) are fully hoisted.

---

46. Guess the output ?

```
const arr = [1,2,3];
arr[5] = 6;
console.log(arr.length);
```

Video Explaination: https://www.instagram.com/p/DF5IdHRijnU/

Explaination:

The output will be `6`.

1. **Initial Array**:

   ```
   const arr = [1, 2, 3]; // arr.length is 3
   ```

   The array starts with indexes 0, 1, and 2.

2. **Assigning to Index 5**:

   ```
   arr[5] = 6;
   ```

   - JavaScript automatically **expands the array** to accommodate index 5.
   - Indexes 3 and 4 become "empty slots" (they are not initialized with any value).

   The array now looks like:

   ```
   [1, 2, 3, empty, empty, 6]
   ```

3. **Array Length**:

   - The `length` of an array is always **1 more than the highest index** assigned.
   - Here, the highest index is 5 → `length = 5 + 1 = 6`.

Final Output:

```
console.log(arr.length); // 6
```

Key Takeaways:

- Arrays in JavaScript are **dynamic** and expand when you assign values to non-existing indexes.
- Empty slots are not the same as `undefined` – they are gaps in the array.

- The `length` property reflects the highest assigned index + 1, even if there are gaps.

---

47. Guess the output ?

```
const obj = {
  a:1
}
obj.a = 2;
console.log(obj);
```

Video Explaination: https://www.instagram.com/p/DF2i864Ca1E/

Explaination:

The output will be:
`{ a: 2 }`

1. **Object Creation**:

   ```
   const obj = { a: 1 };
   ```

   - Creates an object with a property `a` initialized to `1`.

2. **Modifying the Property**:

   ```
   obj.a = 2;
   ```

   - `const` only prevents reassigning the **variable** `obj` to a new object.
   - **Properties of the object can still be modified**.
   - The value of `a` changes from `1` to `2`.

3. **Final Output**:

   ```
   console.log(obj); // { a: 2 }
   ```

Key Takeaways:

- `const` protects the **variable** (`obj`) from being reassigned, not its **contents**.
- Objects declared with `const` can have their properties updated.

---

48. Guess the output ?

```
let x = {
  a: undefined,
  b: null
}
console.log(JSON.stringify(x))
```

Video Explaination: https://www.instagram.com/p/DF0BHY1iDV7/

Explaination:

The output will be:

`{"b":null}`

1. **Object** `x`:

   ```
   let x = {
     a: undefined,
     b: null
   };
   ```

2. `JSON.stringify()` **Behavior**:
   - `undefined` → **Excluded** from JSON output.
   - `null` → **Included** as `null` in JSON.
3. **Result**:
   - Property `a` (with value `undefined`) is **removed**.
   - Property `b` (with value `null`) is **kept**.

Final Output:

```
console.log(JSON.stringify(x)); // {"b":null}
```

Key Takeaways:

- `JSON.stringify()` ignores properties with `undefined` values.
- `null` is a valid JSON value and is preserved.

---

49. Guess the output ?

```
console.log(true + 1);
console.log(true+"1");
```

Video Explaination: https://www.instagram.com/p/DFxaRLyCu6n/

Explaination:

The outputs will be:

`2` and `"true1"`

1. `console.log(true + 1);`

- **Boolean Conversion**: JavaScript converts `true` to its numeric equivalent:
  - `true` → `1`
- **Addition**:

  ```
  1 (true) + 1 = 2
  ```

  Output: `2`

2. `console.log(true + "1");`

- **String Concatenation**: When `+` is used with a string (`"1"`), JavaScript converts both values to strings.

- `true` → `"true"`
- `"1"` stays as `"1"`

- **Concatenation**:

```
"true" + "1" = "true1"
```

Output: `"true1"`

Key Rules:

- `+` **with numbers**: Booleans convert to numbers (`true` → `1`, `false` → `0`).
- `+` **with strings**: Converts all values to strings and concatenates them.

---

50. Guess the output ?

```
const str = "hello";
str.data = "val";
console.log(str.data);
```

Video Explaination: https://www.instagram.com/p/DFu24SVCn3X/

Explaination:

The output will be `undefined`.

1. **Primitive Strings vs. String Objects**:
   - `"hello"` is a **primitive string**, not an object.
   - Primitives **cannot have properties** added to them.
2. **What Happens When You Try**:
   - When you write `str.data = "val"`, JavaScript temporarily converts the primitive string to a **String object** to allow the assignment.
   - However, this temporary object is **discarded immediately**. The property `data` is not saved.

3. **Accessing** `str.data`:
    - When you read `str.data`, JavaScript creates a **new temporary String object** (which doesn't have the `data` property).
    - Thus, `console.log(str.data)` returns `undefined`.

Key Takeaway:

Primitive strings (like `"hello"`) **cannot hold custom properties**. Use an object (e.g., `{}`) if you need to add properties.

---

27 Problem solving questions:

1. Write a program to remove duplicates from an array ?

(Most Most Asked question)

```
const removeDuplicatesWay1 = (array) => {
  let uniqueArr = [];

  for (let i = 0; i <= array.length - 1; i++) {
    if (uniqueArr.indexOf(array[i]) == -1) {
      uniqueArr.push(array[i]);
    }
  }

  return uniqueArr;
};

removeDuplicatesWay1([1, 2, 1, 3, 4, 2, 2, 1, 5, 6]);
```

```
// ------------------------- (or)-----------------------------

function removeDuplicatesWay2(arr) {
  /* Use the Set object to remove duplicates. This works
   because Sets only store unique values */
  return Array.from(new Set(arr));
  // return [...new Set(arr)] ⇒ another way
}

removeDuplicates([1, 2, 1, 3, 4, 2, 2, 1, 5, 6]);
```

2. Write a JavaScript function that takes an array of numbers and returns a new array with only the even numbers.

```
function findEvenNumbers(arr) {
    const result = [];

    for (let i = 0; i < arr.length; i++) {
        if (arr[i] % 2 === 0) {
            result.push(arr[i]);
            // Add even numbers to the result array
        }
    }

    return result;
}

// Example usage:
const numbers = [1, 2, 3, 4, 5, 6, 7, 8,-8,19, 9, 10];
console.log("Even numbers:", findEvenNumbers(numbers));
```

Time complexity: O(N)

3. How to check whether a string is palindrome or not ?

```
const checkPallindrome = (str) => {
  const len = str.length;

  for (let i = 0; i < len/2; i++) {
    if (str[i] !== str[len - i - 1]) {
      return "Not pallindrome";
    }
  }
  return "pallindrome";
};

console.log(checkPallindrome("madam"));
```

4. Find the factorial of given number ?

```
const findFactorial = (num) => {
  if (num == 0 || num == 1) {
    return 1;
  } else {
    return num * findFactorial(num - 1);
  }
};

console.log(findFactorial(4));
```

---

5. Program to find longest word in a given sentence ?

```
const findLongestWord = (sentence) => {
  let wordsArray = sentence.split(" ");
  let longestWord = "";

  for (let i = 0; i < wordsArray.length; i++) {
    if (wordsArray[i].length > longestWord.length) {
      longestWord = wordsArray[i];
    }
  }

  console.log(longestWord);
};

findLongestWord("Hi Iam Saikrishna Iam a UI Developer");
```

---

6. Write a JavaScript program to find the maximum number in an array.

```
function findMax(arr) {
    if (arr.length === 0) {
        return undefined;
        // Handle empty array case
    }

    let max = arr[0];
    // Initialize max with the first element

    for (let i = 1; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
            // Update max if current element is greater
```

```
        }
    }

    return max;
}

// Example usage:
const numbers = [1, 6, -33, 9, 4, 8, 2];
console.log("Maximum number is:", findMax(numbers));
```

Time complexity: O(N)

7. Write a JavaScript function to check if a given number is prime.

```
function isPrime(number) {
    if (number <= 1) {
        return false;
        // 1 and numbers less than 1 are not prime
    }

    // Loop up to the square root of the number
    for (let i = 2; i <= Math.sqrt(number); i++) {
        if (number % i === 0) {
            return false;
            // If divisible by any number, not prime
        }
    }

    return true;
    // If not divisible by any number, it's prime
}

// Example usage:
```

```
console.log(isPrime(17)); // true
console.log(isPrime(19)); // false
```

Time complexity: O(N)

8. Program to find Reverse of a string without using built-in method ?

```
const findReverse = (sampleString) => {
  let reverse = "";

  for (let i = sampleString.length - 1; i >= 0; i--) {
    reverse += sampleString[i];
  }
  console.log(reverse);
};

findReverse("Hello Iam Saikrishna Ui Developer");
```

9. Find the smallest word in a given sentence ?

```
function findSmallestWord() {
  const sentence = "Find the smallest word";
  const words = sentence.split(' ');
  let smallestWord = words[0];

  for (let i = 1; i < words.length; i++) {
     if (words[i].length < smallestWord.length) {
        smallestWord = words[i];
     }
  }
  console.log(smallestWord);
}
```

```
findSmallestWord();
```

---

10. Write a function sumOfThirds(arr), which takes an array arr as an argument. This function should return a sum of every third number in the array, starting from the first one.

Directions:

If the input array is empty or contains less than 3 numbers then return 0.

The input array will contain only numbers.

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

---

11. Write a JavaScript function that returns the Fibonacci sequence up to a given number of terms.

```
function fibonacciSequence(numTerms) {
    if (numTerms <= 0) {
        return [];
    } else if (numTerms === 1) {
        return [0];
    }
```

```
    const sequence = [0, 1];

    for (let i = 2; i < numTerms; i++) {
        const nextFibonacci = sequence[i - 1] + sequence[i - 2];
        sequence.push(nextFibonacci);
    }

    return sequence;
}

// Example usage:
const numTerms = 10;
const fibonacciSeries = fibonacciSequence(numTerms);
console.log(fibonacciSeries);
// Output: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Time complexity: O(N)

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

```
const findConsecutive = (array) => {
  let maxCount = 0;
  let currentConsCount = 0;

  for (let i = 0; i <= array.length - 1; i++) {
    if (array[i] === 1) {
      currentConsCount += 1;
      maxCount = Math.max(currentConsCount, maxCount);
    } else {
      currentConsCount = 0;
    }
  }

  console.log(maxCount);
};
```

```
findConsecutive([1, 1, 9, 1, 9, 19, 7, 1, 1, 1, 2, 5, 1]);
// output: 3
```

---

13. Given 2 arrays that are sorted [0,3,4,31] and [4,6,30]. Merge them and sort [0,3,4,4,6,30,31] ?

```
const sortedData = (arr1,arr2) => {

let i = 1;
  let j=1;
  let array1 = arr1[0];
  let array2 = arr2[0];

  let mergedArray = [];

  while(array1 || array2){

    if(array2 === undefined || array1 < array2){
      mergedArray.push(array1);
      array1 = arr1[i];
      i++
    }else{
      mergedArray.push(array2);
      array2 = arr2[j];
      j++
    }

  }
  console.log(mergedArray)
```

```
}

sortedData([1,3,4,5],[2,6,8,9])
```

---

14. Create a function which will accepts two arrays arr1 and arr2. The function should return true if every value in arr1 has its corresponding value squared in array2. The frequency of values must be same. (Effecient)

Inputs and outputs:

==========

[1,2,3],[4,1,9] ⇒ true

[1,2,3],[1,9] ==⇒ false

[1,2,1],[4,4,1] =⇒ false (must be same frequency)

```
function isSameFrequency(arr1,arr2){

  if(arr1.length !== arr2.length){
    return false;
  }

  let arrFreq1={};
  let arrFreq2={};

  for(let val of arr1){
   arrFreq1[val] = (arrFreq1[val] || 0) + 1;
  }
```

```
  for(let val of arr2){
    arrFreq2[val] = (arrFreq2[val] || 0) + 1;
  }

  for(let key in arrFreq1){
        if(!key*key in arrFreq2) return false;
        if(arrFreq1[key] !== arrFreq2[key*key]){
            return false
          }
  }
  return true;

}

console.log(isSameFrequency([1,2,5],[25,4,1]))
```

---

15. Given two strings. Find if one string can be formed by rearranging the letters of other string. (Effecient)

Inputs and outputs:

"aaz","zza" ⟹ false

"qwerty","qeywrt" ⟹ true

```
function isStringCreated(str1,str2){
  if(str1.length !== str2.length) return false
  let freq = {};

  for(let val of str1){
    freq[val] = (freq[val] || 0) + 1;
  }
```

```
  for(let val of str2){
    if(freq[val]){
      freq[val] -= 1;
    } else{
      return false;
    }
  }
  return true;
}

console.log(isStringCreated('anagram','nagaram'))
```

---

16. Write logic to get unique objects from below array ?

I/P: [{name: "sai"},{name:"Nang"},{name: "sai"},{name:"Nang"},{name: "111111"}];

O/P: [{name: "sai"},{name:"Nang"}{name: "111111"}

```
function getUniqueArr(array){
    const uniqueArr = [];
    const seen = {};
    for(let i=0; i<=array.length-1;i++){
        const currentItem = array[i].name;
        if(!seen[currentItem]){
            uniqueArr.push(array[i]);
            seen[currentItem] = true;
        }
    }
    return uniqueArr;
}
```

```
let arr = [{name: "sai"},{name:"Nang"},{name: "sai"},
{name:"Nang"},{name: "111111"}];
console.log(getUniqueArr(arr))
```

---

17. Write a JavaScript program to find the largest element in a nested array.

```
function findLargestElement(arr) {
    let max = Number.NEGATIVE_INFINITY;
    // Initialize max to smallest possible number

    // Helper function to traverse nested arrays
    function traverse(arr) {
        for (let i = 0; i < arr.length; i++) {
            if (Array.isArray(arr[i])) {
    // If element is array,recursively call traverse function
                traverse(arr[i]);
            } else {
    // If element is not an array, update max if needed
                if (arr[i] > max) {
                    max = arr[i];
                }
            }
        }
    }

    // Start traversing the input array
    traverse(arr);

    return max;
}
```

```
// Example usage:
const array = [
[3, 4, 58],
[709, 8, 9, [10, 11]], [111, 2]
];
console.log("Largest element:", findLargestElement(array));
```

Time complexity: O(N)

---

18. Given a string, write a javascript function to count the occurrences of each character in the string.

```
function countCharacters(str) {
    // Object to store character counts
    const charCount = {};
    const len = str.length;

// Loop through string & count occurrences of each character
    for (let i = 0; i < len; i++) {
        const char = str[i];
        // Increment count for each character
        charCount[char] = (charCount[char] || 0) + 1;
    }

    return charCount;
}

// Example usage:
const result = countCharacters("helaalo");
console.log(result);
// Output: { h: 1, e: 1, l: 2, o: 1 }
```

Time complexity: O(N)

19. Write a javascript function that sorts an array of numbers in ascending order.

```
function quickSort(arr) {
    // Check if the array is empty or has only one element
    if (arr.length <= 1) {
        return arr;
    }

    // Select a pivot element
    const pivot = arr[0];

    // Divide the array into two partitions
    const left = [];
    const right = [];

    for (let i = 1; i < arr.length; i++) {
        if (arr[i] < pivot) {
            left.push(arr[i]);
        } else {
            right.push(arr[i]);
        }
    }

    // Recursively sort the partitions
    const sortedLeft = quickSort(left);
    const sortedRight = quickSort(right);

  // Concatenate sorted partitions with the pivot & return
    return sortedLeft.concat(pivot, sortedRight);
}

// Example usage:
```

```
const unsortedArray = [5, 2, 9, 1, 3, 6];
const sortedArray = quickSort(unsortedArray);
console.log(sortedArray);
// Output: [1, 2, 3, 5, 6, 9]
```

Time complexity: O(n log n)

20. Write a javascript function that sorts an array of numbers in descending order.

```
function quickSort(arr) {
    if (arr.length <= 1) {
        return arr;
    }

    const pivot = arr[0];
    const left = [];
    const right = [];

    for (let i = 1; i < arr.length; i++) {
        if (arr[i] >= pivot) {
            left.push(arr[i]);
        } else {
            right.push(arr[i]);
        }
    }

    return [...quickSort(left), pivot, ...quickSort(right)];
}

const arr = [3, 1, 4, 1, 5, 9, 2, 6, 5];
const sortedArr = quickSort(arr);
```

```
console.log(sortedArr);
// Output: [9, 6, 5, 5, 4, 3, 2, 1, 1]
```

Time complexity: O(n log n)

21. Write a javascript function that reverses the order of words in a sentence without using the built-in reverse() method.

```
const reverseWords = (sampleString) => {
  let reversedSentence = "";
  let word = "";

  // Iterate over each character in the sampleString
  for (let i = 0; i < sampleString.length; i++) {
    /* If the character is not a space,
    append it to the current word */

    if (sampleString[i] !== ' ') {
      word += sampleString[i];
    } else {
      /* If it's a space, prepend the current word
       to the reversed sentence and
       reset the word */

      reversedSentence = word + ' ' + reversedSentence;
      word = "";
    }
  }

  // Append the last word to the reversed sentence
  reversedSentence = word + ' ' + reversedSentence;

  // Trim any leading or trailing spaces and log the result
  console.log(reversedSentence.trim());
```

```
};

// Example usage
reverseWords("ChatGPT is awesome");
//"awesome is ChatGPT"
```

```
function reverseWords(sentence) {
    // Split the sentence into words
    let words = [];
    let wordStart = 0;
    for (let i = 0; i < sentence.length; i++) {
        if (sentence[i] === ' ') {
            words.unshift(sentence.substring(wordStart, i));
            wordStart = i + 1;
        } else if (i === sentence.length - 1) {
            words.unshift(sentence.substring(wordStart,i+1));
        }
    }

    // Join the words to form the reversed sentence
    return words.join(' ');
}

// Example usage
const sentence = "ChatGPT is awesome";
console.log(reverseWords(sentence));
// Output: "awesome is ChatGPT"
```

Time complexity: O(N)

22. Implement a javascript function that flattens a nested array into a single-dimensional array.

```
function flattenArray(arr) {
    const stack = [...arr];
    const result = [];

    while (stack.length) {
        const next = stack.pop();
        if (Array.isArray(next)) {
            stack.push(...next);
        } else {
            result.push(next);
        }
    }

    return result.reverse();
    // Reverse the result to maintain original order
}

// Example usage:
const nestedArray = [1, [2, [3, 4], [7,5]], 6];
const flattenedArray = flattenArray(nestedArray);
console.log(flattenedArray);
// Output: [1, 2, 3, 4, 5, 6]
```

---

23. Write a function which converts string input into an object

// stringToObject("a.b.c", "someValue");

// output → {a: {b: {c: "someValue"}}}

```
function stringToObject(str, finalValue) {
  const keys = str.split('.');
  let result = {};
  let current = result;
```

```
  for (let i = 0; i < keys.length; i++) {
    const key = keys[i];
    current[key] = (i === keys.length - 1) ? finalValue : {};
    current = current[key];
  }

  return result;
}

// Test the function
const output = stringToObject("a.b.c", "someValue");
console.log(output);
// Output: {a: {b: {c: "someValue"}}}
```

---

24. Given an array, return an array where the each value is the product of the next two items: E.g. `[3, 4, 5]` -> `[20, 15, 12]`

```
function productOfNextTwo(arr) {
    const result = [];
    for (let i = 0; i < arr.length; i++) {
        if (i < arr.length - 1) {
            result.push(arr[i + 1] * arr[i + 2]);
        } else {
            result.push(arr[0] * arr[1]);
        }
    }
    return result;
}

// Example usage:
const inputArray = [3, 4, 5];
```

```
const outputArray = productOfNextTwo(inputArray);
console.log(outputArray); // Output: [20, 15, 12]
```

---

25. Find the 2nd largest element from a given array ? [100,20,112,22]

```
function findSecondLargest(arr) {
    if (arr.length < 2) {
        throw new Error(`Array must contain
         at least two elements.`);
    }

    let largest = -Infinity;
    let secondLargest = -Infinity;

    for (let i = 0; i < arr.length; i++) {
        if (arr[i] > largest) {
            secondLargest = largest;
            largest = arr[i];
        } else if (arr[i] > secondLargest && arr[i] < largest) {
            secondLargest = arr[i];
        }
    }

    if (secondLargest === -Infinity) {
        throw new Error(`There is no second largest
        element in the array.`);
    }

    return secondLargest;
}

// Example usage:
```

```
const array = [10, 5, 20, 8, 12];
console.log(findSecondLargest(array)); // Output: 12
```

26. Program challenge: Find the pairs from given input ?

input1 = [1, 2, 3, 4, 5, 6, 7, 8, 9];

input2 = 10;

output = [[4, 6], [3, 7], [2, 8], [1, 9]]

```
function findPairs(input1, input2) {
  const pairs = [];
  const seen = new Set();

  for (const num of input1) {
    const complement = input2 - num;
    if (seen.has(complement)) {
      pairs.push([complement, num]);
    }
    seen.add(num);
  }

  return pairs;
}

const input1 = [1, 2, 3, 4, 5, 6, 7, 8, 9];
const input2 = 10;

const output = findPairs(input1, input2);
console.log(output);
// [[1, 9], [2, 8], [3, 7], [4, 6], [5, 5]]
```

27. Write a javascript program to get below output from given input ?

I/P: abbcccddddeea

O/P: 1a2b3c4d2e1a

```
function encodeString(input) {
    if (input.length === 0) return "";

    let result = "";
    let count = 1;

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

    // Add the last sequence
    result += count + input[input.length - 1];

    return result;
}

const input = "abbcccddddeea";
const output = encodeString(input);
console.log(output);
// Outputs: 1a2b3c4d2e1a
```