### **DSA Assignment | Session 2 [Deadline: 2025-08-08]**

### **Solve these problems and calculate the time complexity.**

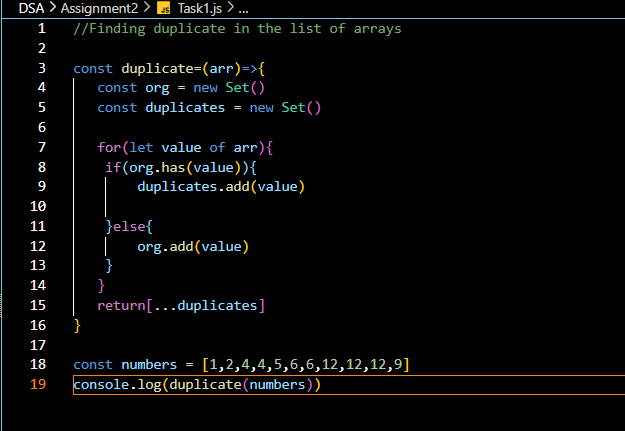
### **Find if a duplicate exists in a list of numbers.**

For eg:

const data = [1, 3, 5, 2, 4, 5];

Answer: [5]

Ans :



Output : 

Time complexity : O(n) because here we can see we have only used loop once through the array .

* **Check if any item from user roles exists in required roles.**

For eg:

Find if *user* and *editor* roles are present in requiredRoles.(roles can be any of length)

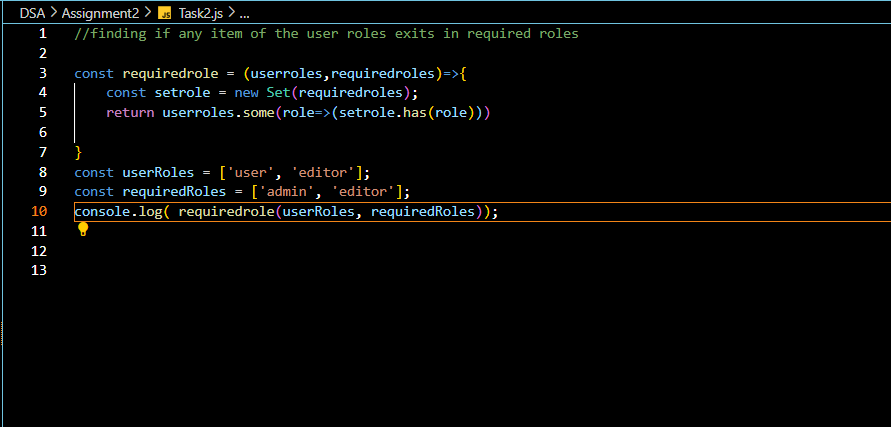
const roles = ['admin', ’manager’, supervisor‘’, 'editor', ‘viewer’, ‘owner’, ‘’]

const userRoles = ['user', 'editor']

const requiredRoles = ['admin', 'editor']

Answer: true

Ans:



Output :



Time Complexity : O(n+m)

Where n denotes the length of user roles and m denotes the length of required roles .

* **Find Common Elements Between Two Arrays**

For eg:

const a1 = [1,4,2,8,9]

const a2 = [7,5,0,4,1]

Answer: [1, 4]

Ans :

A screen shot of a computer program

AI-generated content may be incorrect.

Output :



Time complexity : O(n+m)

* **Filter items based on allowed keys**

For eg:

const data = [

{ key: 'name', value: ‘John’},

{ key: 'email', value:’john@example.com' },

{ key: 'age', value: 20 },

]

const allowedKeys = ['name', 'age']

Answer: [ { key: 'name', value: 'John' }, { key: 'age', value: 20 } ]

Ans :



Output :



Time Complexity = O(n)

* **Determine whether both keys and values of two objects are equal**

For eg:

const d1= {name:’John’, email:’john@example.com' , age: 20 }

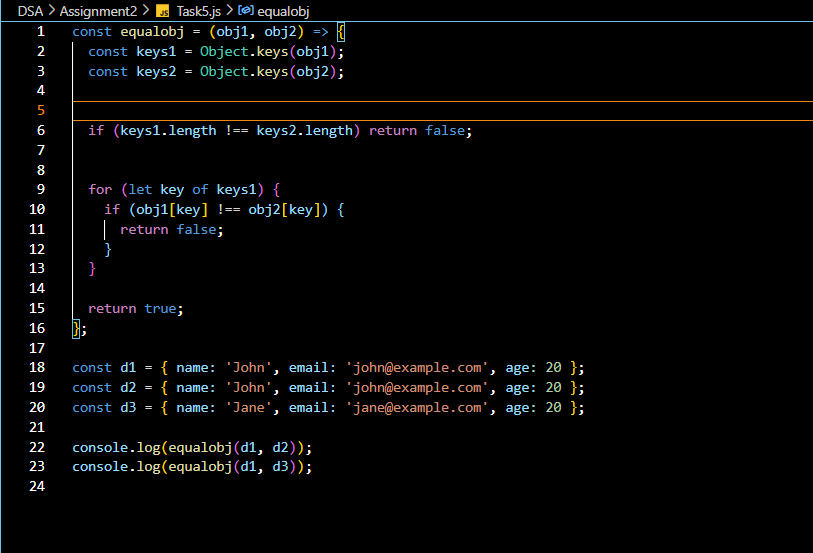
const d2= {name:’John’, email:’john@example.com' , age: 20 }

const d3= {name:’Jane’, email:’jane@example.com' , age: 20 }

Answer: sameObject(d1,d2) => true

Answer: sameObject(d1,d3) => false

Ans:



Output:



Time Complexity :

O(n) where n represent the number of key in the object.

* **Group the related data by category**

const products = [

{ id: 1, name: 'Phone', categoryId: 2 },

{ id: 2, name: 'Shirt', categoryId: 1 },

{ id: 3, name: 'Charger', categoryId: 2 }

]

Answer:

{

'1': [ { id: 2, name: 'Shirt', categoryId: 1 } ],

'2': [

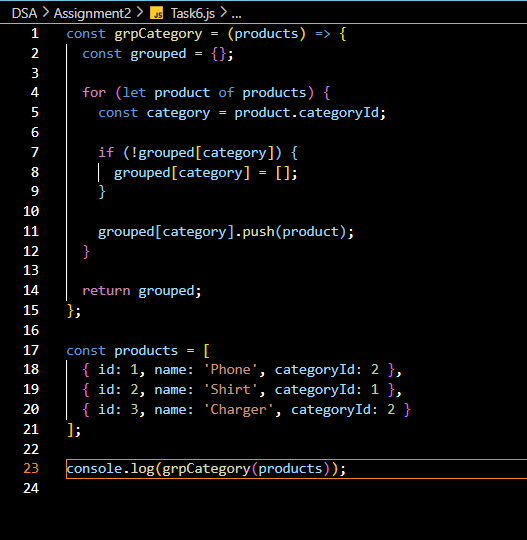
{ id: 1, name: 'Phone', categoryId: 2 },

{ id: 3, name: 'Charger', categoryId: 2 }

]

}

Ans :



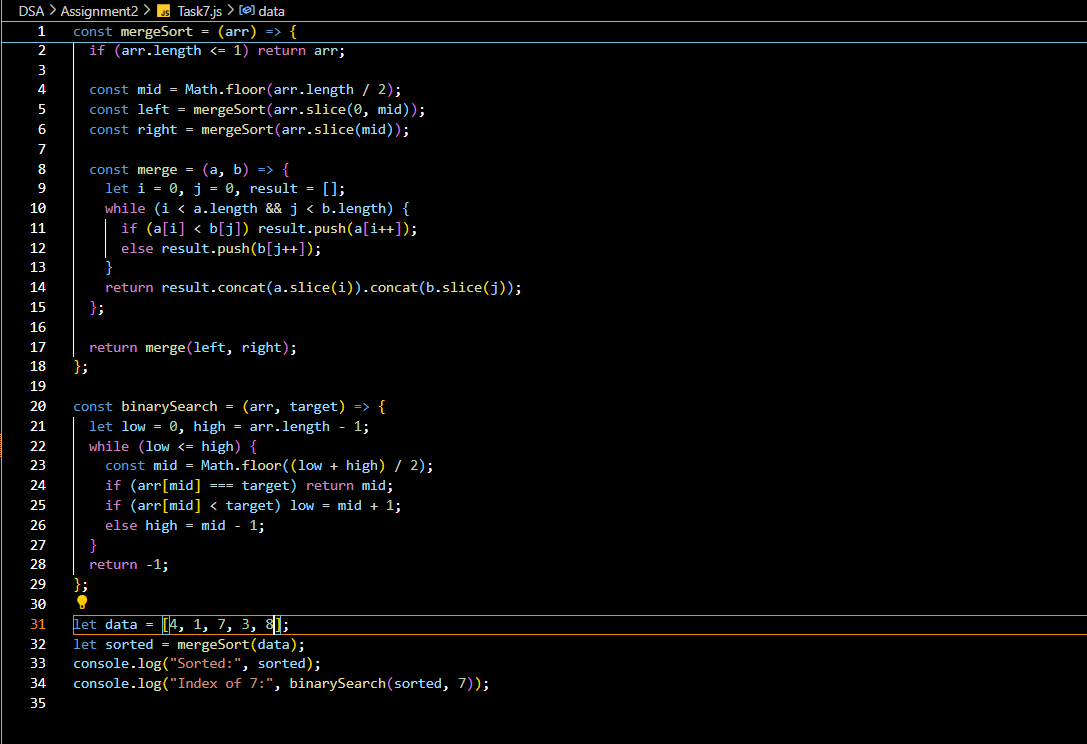
Output :

A screen shot of a computer

AI-generated content may be incorrect.

Time complexity : O(n) where n reperesents the number of products

* **Implement binary search algorithm. (If sorting is required, you should use one of the sorting algorithms: Selection, Insertion, Merge, Quick, Heap Sort)**

****

**Output :**

****

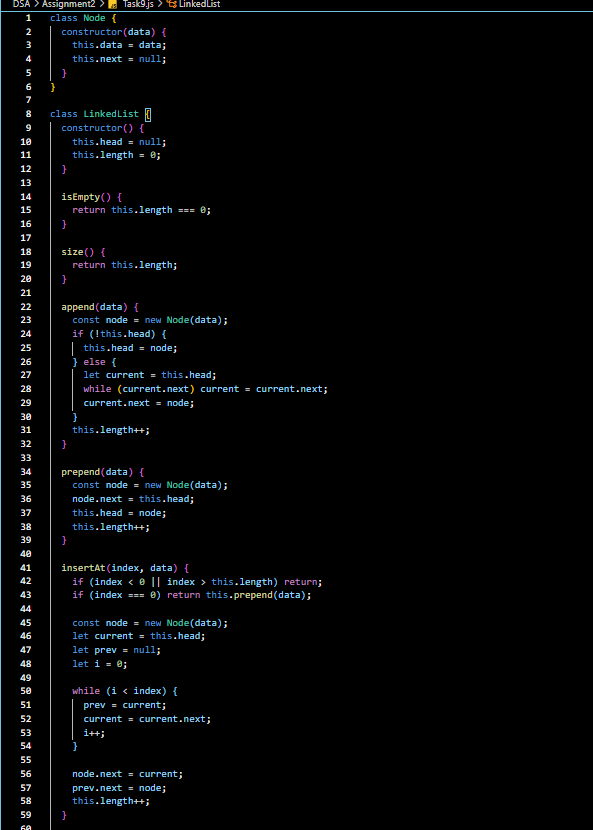
**Time complexity :**

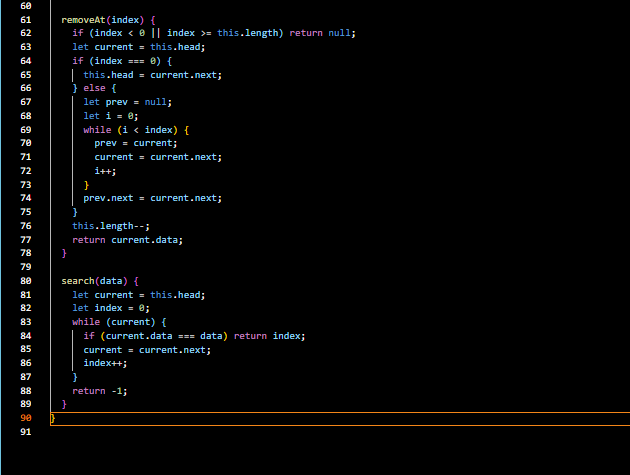
** Merge Sort: O(n log n)**

** Binary Search: O(log n)**

* **Implement singly linked list. Operations on the linked list should be:**
  + Append: Add data at the end
  + Prepend: Add data at the beginning
  + RemoveAt: Removes data from the given index
  + InsertAt: Add data at the given index
  + Size: returns the size of list
  + isEmpty: Checks if list is empty or not
  + Search: Finds data from linked list.

Ans

: 



A computer screen shot of a black screen with white text

AI-generated content may be incorrect.

Output :

A black screen with white text

AI-generated content may be incorrect.

Time complexity :

| **Operation** | **Time Complexity** |
| --- | --- |
| **isEmpty()** | O(1) |
| **size()** | O(1) |
| **append()** | O(n) |
| **prepend()** | O(1) |
| **insertAt()** | O(n) |
| **removeAt()** | O(n) |
| **search()** | O(n) |
| **clear()** | O(1) |
| **printList()** | O(n) |  |

* **Function to detect whether the dependency structure contains any cycles, where an item indirectly or directly depends on itself.(optional)**

const deps = {

A: { id: 1, dependsOn: [ { id: 2 } ] },

B: { id: 2, dependsOn: [ { id: 3 } ] },

C: { id: 3, dependsOn: [ { id: 1 } ] }

}

Answer: true

* **If there’s cycle, Find the Cycle Path**

Answer: [‘A’, ‘B’ , ‘C ‘, ‘ A’]

Ans :

A screen shot of a computer program

AI-generated content may be incorrect.

Output :

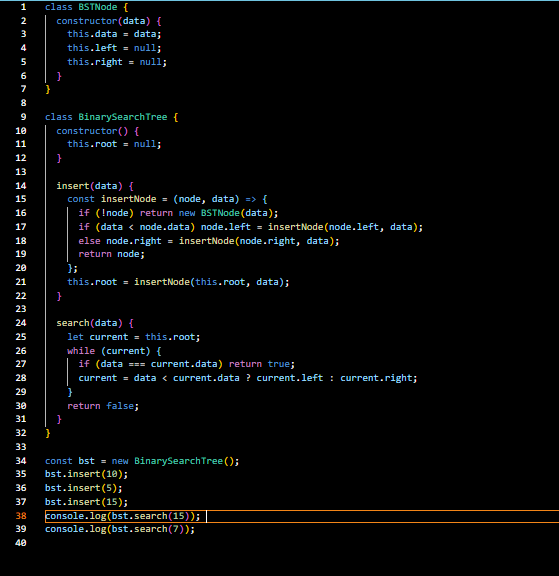


Time complexity : O(n+e)

* **Explain the concept of a Binary Search Tree and provide its implementation in JavaScript.**

**Binary Search is a search algorithm used to find the position of a target element in a sorted array.**

**Instead of checking every element (like linear search), binary search divides the array in half each time and eliminates half of the possibilities, making it much faster.**

****

**Output :**

****