**DSA Assignment | Session 2**

**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];

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

for (j=i+1;j<data.length;j++){

count=0;

if(data[i]==data[j]){

count++;

}

}

if(count>=1){

console.log(data[i])

}

}

Answer:  [5]

Time complexity:O(n2)

* **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 userRoles = ['user', 'admin']

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

function check(userRoles,requiredRoles){

for(let urole of userRoles){

for(let rrole of requiredRoles){

if(urole==rrole){

return true

}else{

}

}

}

return false

}

console.log(check(userRoles,requiredRoles));

Answer: true

Time complexity=O(n2)

* **Find Common Elements Between Two Arrays**

For eg:

const a1 =  [1,4,2,8,9]

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

const common=a1.filter(item=>a2.includes(item));

console.log(common)

Answer: [1, 4]

Time complexity o(1)

* **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']

const filtered=data.filter(item=>allowedKeys.includes(item.key));

console.log(filtered)

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

Time complexity o(1)

* **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 }

function check(o1,o2){

const key1=Object.keys(o1)

const key2=Object.keys(o2)

if(key1.length!=key2.length){

return false

}

for(key of key1){

if(o1[key]!==o2[key]){

return false

}

}

return true

}

console.log(check(d1,d2))

Answer: sameObject(d1,d2) => true

Answer: sameObject(d1,d3) => false

* **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 }

]

let newObj={}

for(product of products){

const catId=product.categoryId;

if(!newObj[catId]){

newObj[catId]=[]

}

newObj[catId].push(product)

}

console.log(newObj)

Answer:

{

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

  '2': [

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

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

  ]

}

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

function selectionSort(arr) {

let n = arr.length;

for (let i = 0; i < n - 1; i++) {

let minIndex = i;

for (let j = i + 1; j < n; j++) {

if (arr[j] < arr[minIndex]) {

minIndex = j;

}

}

if (minIndex !== i) {

[arr[i], arr[minIndex]] = [arr[minIndex], arr[i]];

}

}

console.log(`sorted array =${arr}`)

return arr;

}

const arr = [64, 25, 12, 22, 11];

const binSearch=(arr,target)=>{

let start=0;

let end=arr.length-1;

while(start<=end){

let mid=Math.floor((start+end)/2)

if(arr[mid]==target)return mid

else if(target<arr[mid]){

end=mid-1;

}else{

start=mid+1;

}

}

return "doesnt exit"

}

target=24;

console.log( `the index of ${target} is ${binSearch(selectionSort(arr),target)}`)

* **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.

class Node {

constructor(data) {

this.data = data;

this.next = null;

}

}

class SinglyLinkedList {

constructor() {

this.head = null;

this.size = 0;

}

append(data) {

const newNode = new Node(data);

if (!this.head) {

this.head = newNode;

} else {

let current = this.head;

while (current.next) {

current = current.next;

}

current.next = newNode;

}

this.size++;

}

prepend(data) {

const newNode = new Node(data);

newNode.next = this.head;

this.head = newNode;

this.size++;

}

removeAt(index) {

if (index < 0 || index >= this.size) {

return null;

}

let removedData;

if (index === 0) {

removedData = this.head.data;

this.head = this.head.next;

} else {

let current = this.head;

let prev = null;

let i = 0;

while (i < index) {

prev = current;

current = current.next;

i++;

}

removedData = current.data;

prev.next = current.next;

}

this.size--;

return removedData;

}

insertAt(data, index) {

if (index < 0 || index > this.size) {

return false;

}

const newNode = new Node(data);

if (index === 0) {

newNode.next = this.head;

this.head = newNode;

} else {

let current = this.head;

let prev = null;

let i = 0;

while (i < index) {

prev = current;

current = current.next;

i++;

}

newNode.next = current;

prev.next = newNode;

}

this.size++;

return true;

}

getSize() {

return this.size;

}

isEmpty() {

return this.size === 0;

}

search(data) {

let current = this.head;

let index = 0;

while (current) {

if (current.data === data) {

return index;

}

current = current.next;

index++;

}

return -1;

}

printList() {

let current = this.head;

let result = '';

while (current) {

result += current.data + (current.next ? ' -> ' : '');

current = current.next;

}

console.log(result);

}

}

* **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’]

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

class Node {

constructor(value) {

this.value = value;

this.left = null;

this.right = null;

}

}

class BST {

constructor() {

this.root = null;

}

insert(value) {

const newNode = new Node(value);

if (!this.root) {

this.root = newNode;

return;

}

let current = this.root;

while (true) {

if (value === current.value) return; // No duplicates

if (value < current.value) {

if (!current.left) {

current.left = newNode;

return;

}

current = current.left;

} else {

if (!current.right) {

current.right = newNode;

return;

}

current = current.right;

}

}

}

search(value) {

let current = this.root;

while (current) {

if (value === current.value) return true;

current = value < current.value ? current.left : current.right;

}

return false;

}

inOrderTraversal(node = this.root) {

if (node) {

this.inOrderTraversal(node.left);

console.log(node.value);

this.inOrderTraversal(node.right);

}

}

findMin(node = this.root) {

while (node.left) {

node = node.left;

}

return node.value;

}

findMax(node = this.root) {

while (node.right) {

node = node.right;

}

return node.value;

}

}