

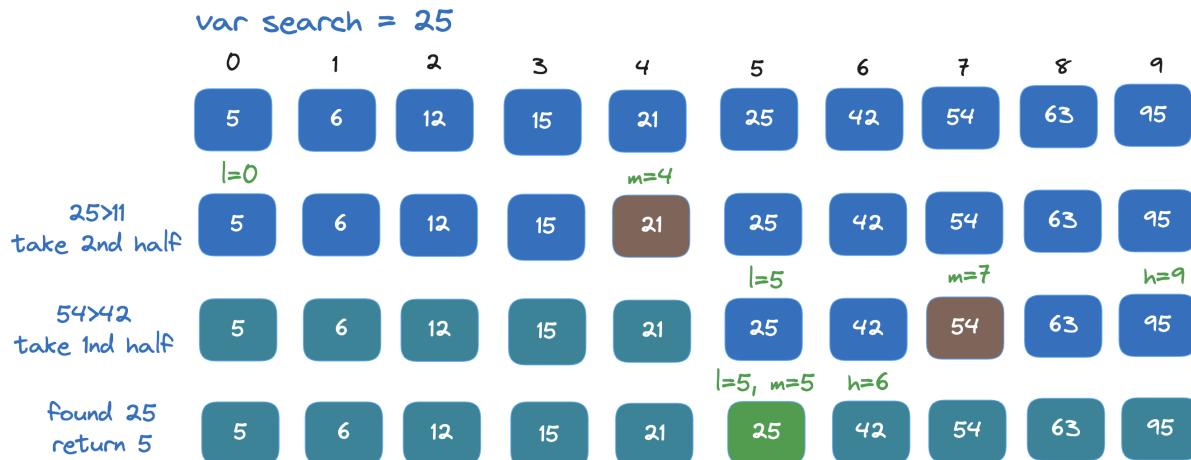
# Holy Theory

## Holy Theory project

### algorithms

#### Binary search

#### Binary search



#### Steps:

- Step 1 - Read the search element from the user.
- Step 2 - Find the middle element in the sorted list.
- Step 3 - Compare the search element with the middle element in the sorted list.
- Step 4 - If both are matched, then display "Given element is found!!!" and terminate the function.
- Step 5 - If both are not matched, then check whether the search element is smaller or larger than the middle element.
- Step 6 - If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- Step 7 - If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- Step 8 - Repeat the same process until we find the search element in the list or until sublist contains only one element.
- Step 9 - If that element also doesn't match with the search element, then returns -1;

## Time Complexity:

- Worst case:  $O(\log n)$
- Average case:  $O(\log n)$
- Best case:  $O(1)$

```
function binarySearch(nums: number[], target: number): number {  
    let left: number = 0;  
    let right: number = nums.length - 1;  
  
    while (left <= right) {  
        const mid: number = Math.floor((left + right) / 2);  
  
        if (nums[mid] === target) return mid;  
        if (target < nums[mid]) right = mid - 1;  
        else left = mid + 1;  
    }  
  
    return -1;  
}
```

```
class Solution {  
    private static int binarySearch(int[] array, int target) {  
  
        int low = 0;  
        int high = array.length - 1;  
  
        while(low <= high) {  
            int middle = low + (high - low) / 2;  
            int value = array[middle];  
  
            if(value < target) {  
                low = middle + 1;  
            } else if(value > target) {  
                high = middle - 1;  
            } else {  
                return middle;  
            }  
        }  
        return -1;  
    }  
}
```

```
def binary_search(list, item):  
    low = 0  
    high = len(list) - 1  
    while low <= high:  
        mid = (low+high)/2  
        guess = list[mid]  
        if guess == item:  
            return mid  
        if guess > item:
```

```

        high = mid - 1
    else:
        low = mid +1
    return None

my_list = [1, 3, 5, 7, 9]

res = binary_search(my_list, 3)

print(my_list[res])

```

Binary tree in order traversal

## Binary tree in order traversal

```

class Solution {

    List<Integer> getInOrderTraversal(Node root) {
        List<Integer> list = new ArrayList<Integer>();
        Stack<Node> stack = new Stack<>();
        Node node = root;

        while(node != null || !stack.isEmpty()) {
            while(node != null) {
                stack.push(node);
                node = node.left;
            }
            list.add(stack.peek().data);
            node = stack.pop().right;
        }

        return list;
    }
}

```

```

class TreeNode {
    data: number;
    left: TreeNode | null;
    right: TreeNode | null;

    constructor(data: number) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

function getInOrderTraversal(root: TreeNode | null): number[] {
    const list: number[] = [];

```

```

const stack: TreeNode[] = [];
let node: TreeNode | null = root;

while (node !== null || stack.length > 0) {
    while (node !== null) {
        stack.push(node);
        node = node.left;
    }
    list.push(stack[stack.length - 1].data);
    node = stack.pop()!.right;
}

return list;
}

```

Binary tree postorder traversal

## Binary tree postorder traversal

```

class Solution {

    void utility(Node root, List<Integer> traversal) {
        if(root == null) {
            return;
        }

        utility(root.left, traversal);
        utility(root.right, traversal);
        traversal.add(root.data);
    }

    List<Integer> getPostorderTraversal(Node root) {
        List<Integer> traversal = new ArrayList<Integer>();
        utility(root, traversal);
        return traversal;
    }
}

```

```

class Node {
    data: number;
    left: Node | null;
    right: Node | null;

    constructor(data: number) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

```

```

function utility(root: Node | null, traversal: number[]): void {
    if (root === null) {
        return;
    }

    utility(root.left, traversal);
    utility(root.right, traversal);
    traversal.push(root.data);
}

function getPostorderTraversal(root: Node | null): number[] {
    const traversal: number[] = [];
    utility(root, traversal);
    return traversal;
}

```

Binary tree preorder traversal

## Binary tree preorder traversal

```

class Solution {

    void utility(Node root, List<Integer> traversal) {
        if(root == null) {
            return;
        }

        traversal.add(root.data);
        utility(root.left, traversal);
        utility(root.right, traversal);
    }

    List<Integer> getPreorderTraversal(Node root) {
        List<Integer> traversal = new ArrayList<Integer>();
        utility(root, traversal);
        return traversal;
    }
}

```

```

class Node {
    data: number;
    left: Node | null;
    right: Node | null;

    constructor(data: number) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

```

```

function utility(root: Node | null, traversal: number[]): void {
    if (root === null) {
        return;
    }

    traversal.push(root.data);
    utility(root.left, traversal);
    utility(root.right, traversal);
}

function getPreorderTraversal(root: Node | null): number[] {
    const traversal: number[] = [];
    utility(root, traversal);
    return traversal;
}

```

## Breadth-first search

### Breadth-first search

Breadth-First Search (BFS) is a graph traversal algorithm that systematically explores all the vertices of a graph in breadthward motion, level by level. It starts at a chosen vertex and visits all its neighbors before moving on to their neighbors. BFS is commonly used to find the shortest path in an unweighted graph and to explore the structure of a graph.

#### How Breadth-First Search Works:

##### 1. Queue Initialization:

- Begin by selecting a starting vertex and enqueue it into a queue.

##### 2. Explore Neighbors:

- Dequeue a vertex from the front of the queue and explore all its neighbors.
- Enqueue any unvisited neighbors, marking them as visited to avoid duplication.

##### 3. Level-wise Exploration:

- Continue the process level by level, exploring all vertices at the current level before moving on to the next level.

##### 4. Termination:

- Repeat until the queue is empty, ensuring that all reachable vertices are visited.

#### Key Characteristics:

##### • FIFO Structure:

- BFS uses a First-In-First-Out (FIFO) queue to maintain the order in which vertices are discovered and processed.

##### • Visited Marking:

- To avoid revisiting vertices, mark each vertex as visited once it is dequeued from the queue.

- **Shortest Path:**

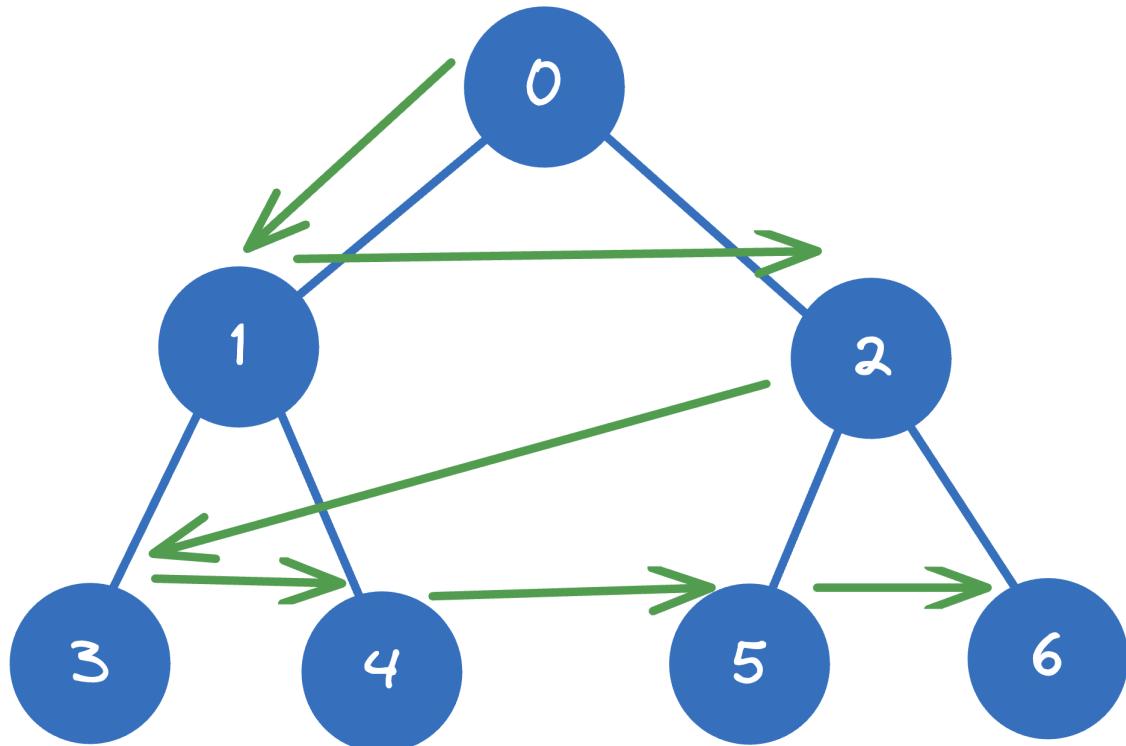
- BFS guarantees that the shortest path to any reachable vertex is discovered first, making it valuable for pathfinding in unweighted graphs.

**Applications:**

- Shortest Pathfinding.
- Connected Components.
- Web Crawling.
- Network Broadcasting.

**Time Complexity:**

- The time complexity of BFS is  $(O(V + E))$ , where  $(V)$  is the number of vertices and  $(E)$  is the number of edges. The algorithm visits each vertex and edge once.



```

class Graph {
    private adjacencyList: Map<string, string[]>;
    
    constructor() {
        this.adjacencyList = new Map();
    }

    addVertex(vertex: string) {
        if (!this.adjacencyList[vertex]) {
            this.adjacencyList[vertex] = [];
        }
    }

    addEdge(source: string, destination: string) {
        if (!this.adjacencyList[source]) {
            this.adjacencyList[source] = [];
        }
        if (!this.adjacencyList[destination]) {
            this.adjacencyList[destination] = [];
        }
        this.adjacencyList[source].push(destination);
        this.adjacencyList[destination].push(source);
    }

    bfs(start: string) {
        const queue: string[] = [start];
        const visited: Set<string> = new Set();
        const distances: Map<string, number> = new Map();
        const previous: Map<string, string | null> = new Map();

        distances.set(start, 0);
        previous.set(start, null);

        while (queue.length > 0) {
            const current = queue.shift();
            if (current === null) {
                break;
            }
            if (visited.has(current)) {
                continue;
            }
            visited.add(current);
            const neighbors = this.adjacencyList[current];
            for (const neighbor of neighbors) {
                if (!visited.has(neighbor)) {
                    queue.push(neighbor);
                    distances.set(neighbor, distances.get(current)! + 1);
                    previous.set(neighbor, current);
                }
            }
        }
        return { distances, previous };
    }
}
    
```

```

        if (!this.adjacencyList.has(vertex)) {
            this.adjacencyList.set(vertex, []);
        }
    }

    addEdge(vertex1: string, vertex2: string) {
        this.adjacencyList.get(vertex1)?.push(vertex2);
        this.adjacencyList.get(vertex2)?.push(vertex1);
    }

    bfs(startingVertex: string) {
        const visited: Set<string> = new Set();
        const queue: string[] = [];

        visited.add(startingVertex);
        queue.push(startingVertex);

        while (queue.length > 0) {
            const currentVertex = queue.shift()!;
            console.log(currentVertex);

            const neighbors = this.adjacencyList.get(currentVertex) || [];
            for (const neighbor of neighbors) {
                if (!visited.has(neighbor)) {
                    visited.add(neighbor);
                    queue.push(neighbor);
                }
            }
        }
    }

    // Example usage:
    const graph = new Graph();
    graph.addVertex("A");
    graph.addVertex("B");
    graph.addVertex("C");
    graph.addVertex("D");
    graph.addEdge("A", "B");
    graph.addEdge("A", "C");
    graph.addEdge("B", "D");

    graph.bfs("A");
}

```

Bubble sort

## Bubble sort

### Bubble Sort: A Simple Sorting Algorithm

Bubble Sort is one of the simplest sorting algorithms that works by repeatedly stepping through the list to

be sorted, comparing each pair of adjacent items, and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, indicating that the list is sorted.

#### **How Bubble Sort Works:**

##### **1. Comparing Adjacent Elements:**

- Bubble Sort starts by comparing the first two elements of an array. If the first element is greater than the second, they are swapped. If not, they remain in their positions.

##### **2. Iterative Process:**

- This process is then repeated for every pair of adjacent elements throughout the entire array. After the first iteration, the largest element will have "bubbled up" to the last position.

##### **3. Subsequent Passes:**

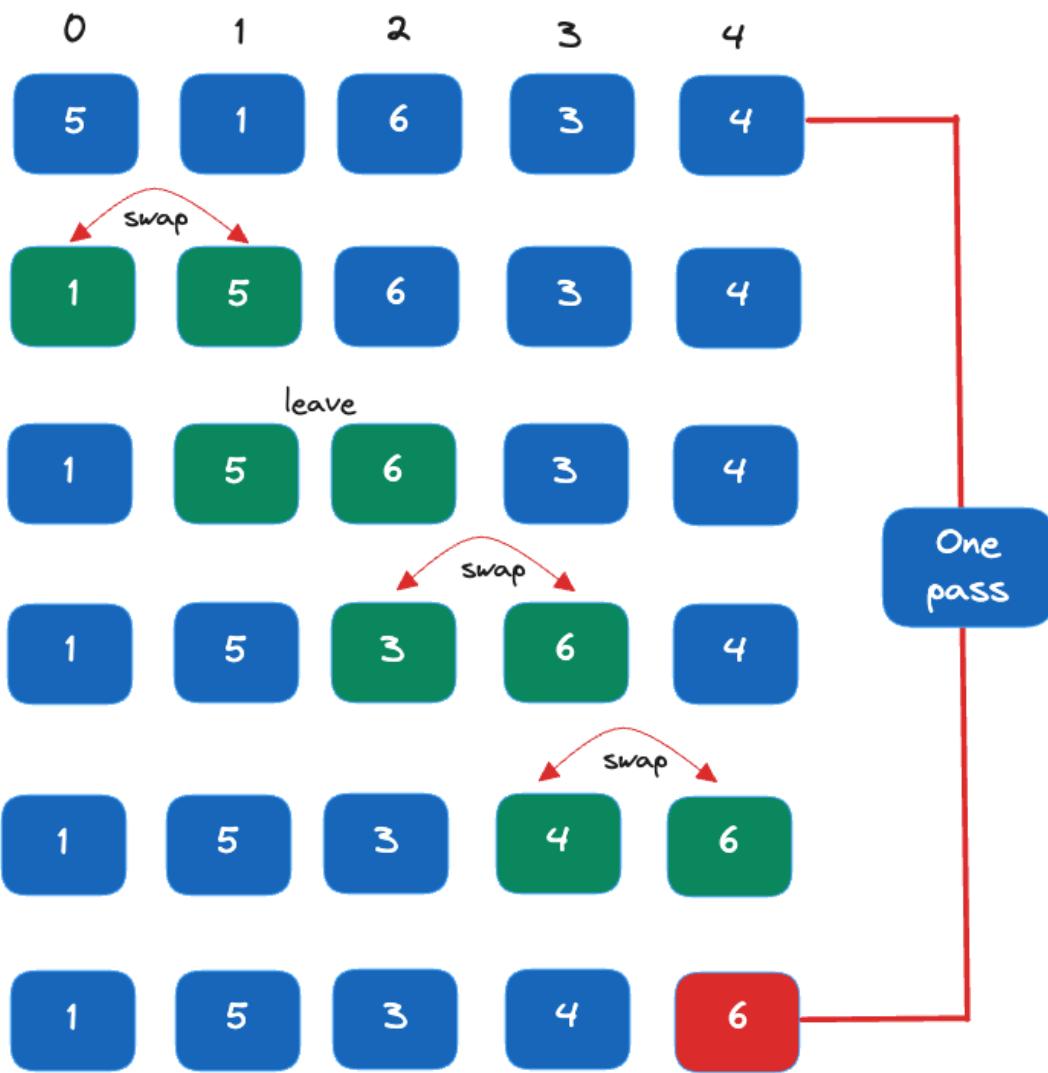
- The algorithm then repeats the process for the remaining elements (excluding the already sorted ones at the end of the array). In each pass, the next largest element is placed in its correct position.

##### **4. Termination:**

- The algorithm terminates when a pass through the entire array is made without any swaps, indicating that the array is now sorted.

#### **Time Complexity:**

- Bubble Sort has a time complexity of  $O(n^2)$  in the worst and average cases, where 'n' is the number of elements in the array. This makes it inefficient for large datasets but is useful for educational purposes due to its simplicity.



Repeat Process

```
function bubbleSort(array: number[] | string[]) {
    for (let i = 0; i < array.length; i++) {
        for (let j = 0; j < array.length - 1 - i; j++) {
            if (array[j] > array[j + 1]) {
                [array[j], array[j + 1]] = [array[j + 1], array[j]];
            }
        }
    }
    return array;
}

console.log(bubbleSort([2,5,2,6,7,2,22,5,7,9,0,2,3]))
```

```

public static void bubbleSort(int[] array) {
    for(int i = 0; i < array.length - 1; i++) {
        for(int j = 0; j < array.length - i - 1; j++) {
            if(array[j] > array[j + 1]) {
                int temp = array[j];
                array[j] = array[j + 1];
                array[j + 1] = temp;
            }
        }
    }
}

```

## Depth-first search

### Depth-first search

Depth-First Search (DFS) is a graph traversal algorithm that systematically explores the vertices of a graph by going as deep as possible along each branch before backtracking. It starts at a chosen vertex, explores as far as possible along one branch, and then backtracks to explore other branches. DFS is commonly used to detect cycles in a graph, topologically sort vertices, and solve problems related to connected components.

#### How Depth-First Search Works:

##### 1. Start at a Vertex:

- Begin by selecting a starting vertex and mark it as visited.

##### 2. Explore Neighbors:

- Move to an unvisited neighbor of the current vertex and repeat the process.
- If all neighbors are visited, backtrack to the previous vertex.

##### 3. Recursion or Stack:

- DFS can be implemented using recursion or an explicit stack to keep track of the vertices to be visited.

##### 4. Marking and Backtracking:

- Mark each visited vertex to avoid revisiting and use backtracking to explore other branches.

##### 5. Complete Exploration:

- Continue the process until all reachable vertices are visited.

#### Key Characteristics:

##### • LIFO Structure:

- DFS often uses a Last-In-First-Out (LIFO) stack or recursion to maintain the order in which vertices are visited.

##### • Visited Marking:

- Mark each vertex as visited once it is reached, preventing revisiting.

- **Backtracking:**

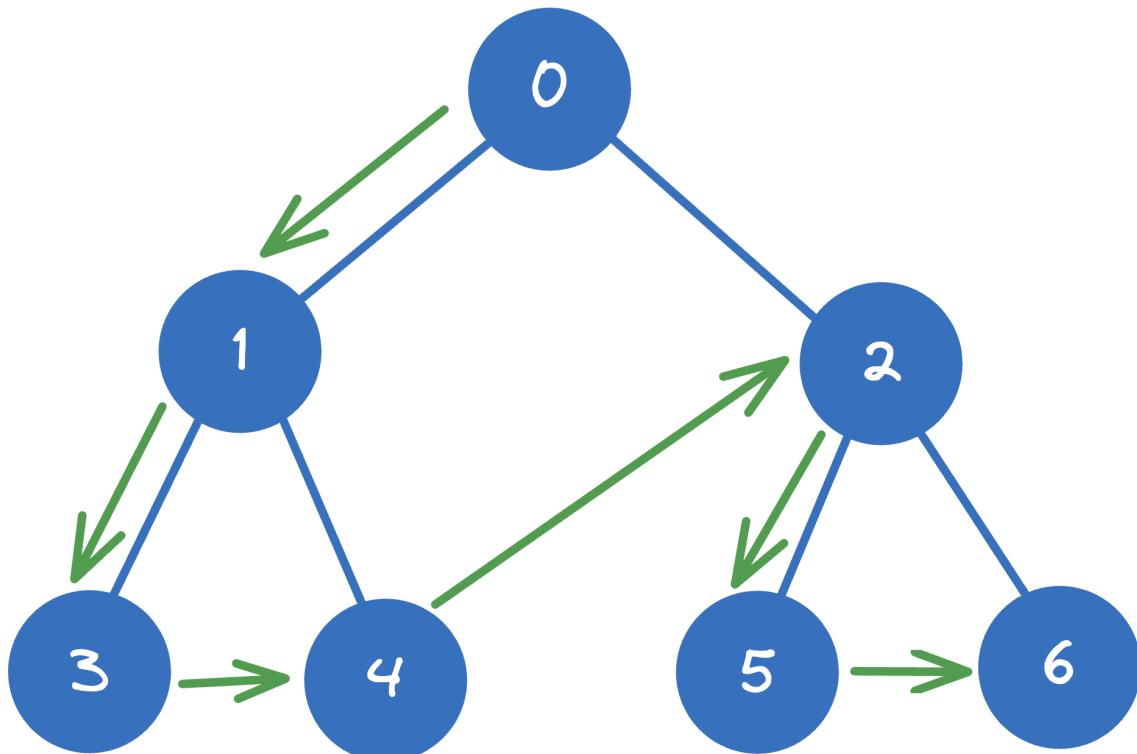
- Backtrack to the previous vertex when all neighbors are explored.

**Applications:**

- Topological Sorting.
- Cycle Detection.
- Connected Components.
- Maze Solving.

**Time Complexity:**

- The time complexity of DFS is  $(O(V + E))$ , where  $(V)$  is the number of vertices and  $(E)$  is the number of edges. The algorithm visits each vertex and edge once. Recursive DFS has a space complexity of  $(O(V))$  due to the call stack, while an explicit stack implementation can have a space complexity of  $(O(E + V))$ .



```
class Graph {
    private adjacencyList: Map<string, string[]>;
    constructor() {
        this.adjacencyList = new Map();
    }
}
```

```

addVertex(vertex: string) {
  if (!this.adjacencyList.has(vertex)) {
    this.adjacencyList.set(vertex, []);
  }
}

addEdge(vertex1: string, vertex2: string) {
  this.adjacencyList.get(vertex1)?.push(vertex2);
  this.adjacencyList.get(vertex2)?.push(vertex1);
}

dfs(startingVertex: string) {
  const visited: Set<string> = new Set();

  const dfsHelper = (vertex: string) => {
    console.log(vertex);
    visited.add(vertex);

    const neighbors = this.adjacencyList.get(vertex) || [];
    for (const neighbor of neighbors) {
      if (!visited.has(neighbor)) {
        dfsHelper(neighbor);
      }
    }
  };

  dfsHelper(startingVertex);
}
}

// Example usage:
const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge("A", "B");
graph.addEdge("A", "C");
graph.addEdge("B", "D");

graph.dfs("A");

```

**Diffie hellman algorithm**

## Diffie hellman algorithm

```

function power(a: any, b: any, p: any) {
  if(b === 1) {
    return 1
  }
}
```

```

    } else {
        return Math.pow(a,b) % p
    }
}

function DiffieHellman() {

    let P, G, x, a, y, b, ka, kb;

    P = 23

    console.log("The value of P :", P);

    G = 9;

    console.log("The value of G :", G);

    a = 4;

    console.log("The private key a for Alice : ", a);

    x = power(G,a,P);

    b = 3;

    console.log("The private key a for Bob : ", b);

    y = power(G,b,P);

    ka = power(y, a, P);
    kb = power(x, b, P);

    console.log("Secret key for the Alice is : ", ka);
    console.log("Secret key for the Bob is : ", kb);
}
}

DiffieHellman()

```

## Dijkstra's algorithm

### Dijkstra's algorithm

#### How Dijkstra's Algorithm Works:

##### 1. Initialization:

- Set the initial distance to the starting vertex as 0 and all other distances to infinity.
- Create a priority queue or a min-heap to store vertices based on their current tentative distances.

## 2. Explore Neighbors:

- While there are vertices in the priority queue, select the vertex with the smallest tentative distance.
- Explore its neighbors and update their tentative distances if a shorter path is found.

## 3. Relaxation:

- For each neighbor, calculate the sum of the tentative distance to the current vertex and the weight of the edge between them.
- If this sum is smaller than the current tentative distance to the neighbor, update the tentative distance.

## 4. Mark as Visited:

- Mark the current vertex as visited to avoid redundant calculations.

## 5. Repeat:

- Repeat steps 2-4 until all vertices are visited or the destination vertex is reached.

## 6. Result:

- The final result is an array of shortest distances from the starting vertex to all other vertices.

## Key Characteristics:

### • Greedy Strategy:

- Dijkstra's Algorithm employs a greedy strategy, always choosing the vertex with the smallest tentative distance.

### • Priority Queue or Min-Heap:

- Efficient implementations use a priority queue or min-heap to efficiently retrieve the vertex with the smallest tentative distance.

•

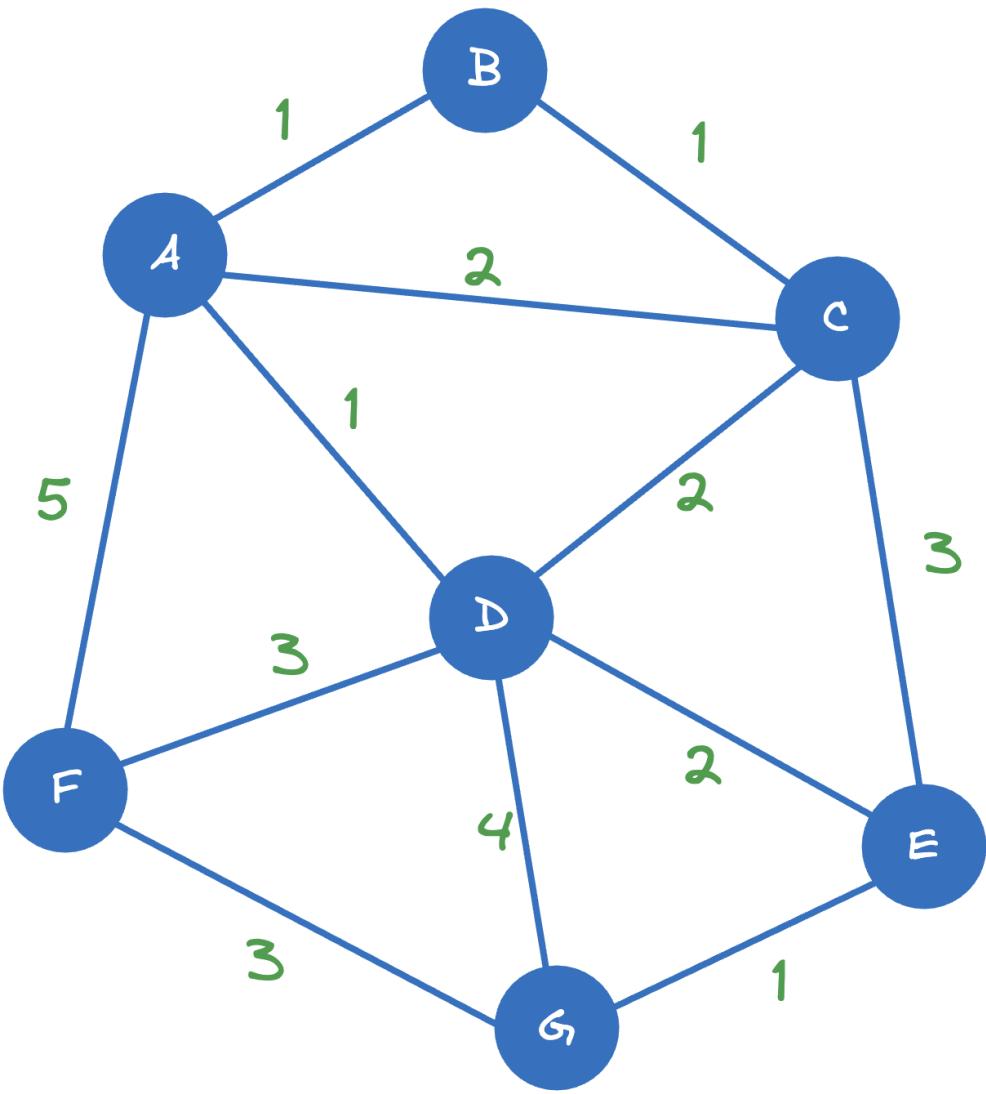
**Non-Negative Edge Weights:** - Dijkstra's Algorithm assumes non-negative edge weights. Negative weights can lead to incorrect results.

## Applications:

- Network Routing.
- Shortest Path Problems.
- Transportation and Logistics.

## Time Complexity:

- The time complexity of Dijkstra's Algorithm is  $O((V + E) \log V)$  using a priority queue or min-heap, where  $V$  is the number of vertices and  $E$  is the number of edges.



```

class Graph {
    private adjacencyList: Map<string, Map<string, number>>;
}

constructor() {
    this.adjacencyList = new Map();
}

addVertex(vertex: string) {
    if (!this.adjacencyList.has(vertex)) {
        this.adjacencyList.set(vertex, new Map());
    }
}

addEdge(vertex1: string, vertex2: string, weight: number) {
    this.adjacencyList.get(vertex1)?.set(vertex2, weight);
    this.adjacencyList.get(vertex2)?.set(vertex1, weight);
}

```

```

}

dijkstra(startingVertex: string) {
  const distances: Map<string, number> = new Map();
  const previous: Map<string, string | null> = new Map();
  const priorityQueue = new PriorityQueue();

  for (const vertex of this.adjacencyList.keys()) {
    distances.set(vertex, vertex === startingVertex ? 0 : Infinity);
    previous.set(vertex, null);
    priorityQueue.enqueue(vertex, distances.get(vertex)!);
  }

  while (!priorityQueue.isEmpty()) {
    const currentVertex = priorityQueue.dequeue()!;
    const neighbors = this.adjacencyList.get(currentVertex);

    if (neighbors) {
      for (const neighbor of neighbors.keys()) {
        const distance = distances.get(currentVertex)! + neighbors.get(neighbor)!;

        if (distance < distances.get(neighbor)!) {
          distances.set(neighbor, distance);
          previous.set(neighbor, currentVertex);
          priorityQueue.enqueue(neighbor, distance);
        }
      }
    }
  }

  return { distances, previous };
}

shortestPath(startingVertex: string, targetVertex: string) {
  const { distances, previous } = this.dijkstra(startingVertex);

  const path: string[] = [];
  let currentVertex = targetVertex;

  while (currentVertex !== null) {
    path.unshift(currentVertex);
    currentVertex = previous.get(currentVertex)!;
  }

  return { path, distance: distances.get(targetVertex) };
}
}

class PriorityQueue {
  private items: [string, number][] = [];

  enqueue(element: string, priority: number) {
    this.items.push([element, priority]);
  }
}

```

```

        this.sort();
    }

    dequeue() {
        return this.items.shift();
    }

    isEmpty() {
        return this.items.length === 0;
    }

    private sort() {
        this.items.sort((a, b) => a[1] - b[1]);
    }
}

// Example usage:
const graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge("A", "B", 1);
graph.addEdge("A", "C", 4);
graph.addEdge("B", "C", 2);
graph.addEdge("B", "D", 5);
graph.addEdge("C", "D", 1);

const { path, distance } = graph.shortestPath("A", "D");
console.log("Shortest Path:", path); // Output: Shortest Path: [ 'A', 'B', 'C', 'D' ]
console.log("Distance:", distance); // Output: Distance: 4

```

## Floyd-Warshall algorithm

### Floyd-Warshall algorithm

The Floyd-Warshall Algorithm is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. Unlike Dijkstra's algorithm and Bellman-Ford algorithm, Floyd-Warshall works with graphs that can have both positive and negative edge weights and can handle graphs with cycles. The algorithm iteratively updates the shortest path distances between all pairs until reaching the optimal solution.

#### How Floyd-Warshall Algorithm Works:

##### 1. Initialization:

- Create a matrix to represent the distances between all pairs of vertices. Initialize the matrix with the direct edge weights and set the distances to infinity where there is no direct edge.
- Initialize the diagonal of the matrix to zeros since the distance from a vertex to itself is zero.

##### 2. Iterative Updates:

- For each vertex 'k', iterate through all pairs of vertices 'i' and 'j'.

- Check if the path from 'i' to 'j' through 'k' is shorter than the current known path from 'i' to 'j'.
- If yes, update the distance from 'i' to 'j' with the shorter path.

### 3. Repeat:

- Repeat the process for all vertices as intermediate vertices ('k').
- After each iteration, the matrix reflects the shortest distances between all pairs of vertices considering all possible intermediate vertices.

### 4. Result:

- The final matrix contains the shortest distances between all pairs of vertices.

## Key Characteristics:

- **Dynamic Programming:**

– Floyd-Warshall is a dynamic programming algorithm that builds solutions to subproblems to solve the overall problem.

- **Negative Cycles:**

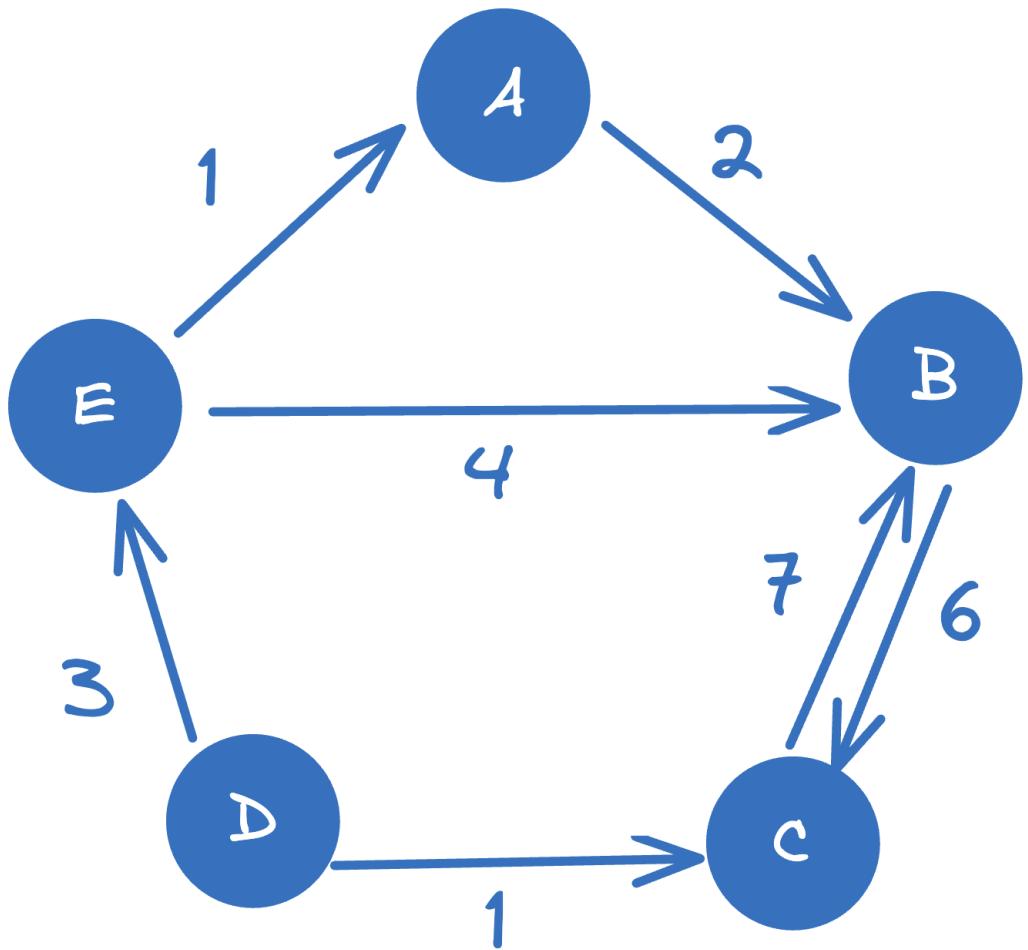
– The algorithm can detect negative cycles in the graph. If there exists a negative cycle, the algorithm won't converge to a solution.

## Applications:

- Network Routing.
- Shortest Path Problems.
- Traffic Engineering.

## Time Complexity:

- The time complexity of Floyd-Warshall Algorithm is ( $O(V^3)$ ), where ( $V$ ) is the number of vertices in the graph.



```

class Graph {
  private adjacencyMatrix: number[][];

  constructor(numVertices: number) {
    this.adjacencyMatrix = Array.from({ length: numVertices }, () =>
      Array(numVertices).fill(Infinity)
    );

    // Set diagonal elements to 0
    for (let i = 0; i < numVertices; i++) {
      this.adjacencyMatrix[i][i] = 0;
    }
  }

  addEdge(source: number, destination: number, weight: number) {
    this.adjacencyMatrix[source][destination] = weight;
  }
}
  
```

```

floydWarshall() {
    const numVertices = this.adjacencyMatrix.length;

    for (let k = 0; k < numVertices; k++) {
        for (let i = 0; i < numVertices; i++) {
            for (let j = 0; j < numVertices; j++) {
                if (
                    this.adjacencyMatrix[i][k] + this.adjacencyMatrix[k][j] <
                    this.adjacencyMatrix[i][j]
                ) {
                    this.adjacencyMatrix[i][j] =
                        this.adjacencyMatrix[i][k] + this.adjacencyMatrix[k][j];
                }
            }
        }
    }

    return this.adjacencyMatrix;
}
}

// Example usage:
const graph = new Graph(4);

graph.addEdge(0, 1, 3);
graph.addEdge(0, 2, 6);
graph.addEdge(1, 2, 1);
graph.addEdge(1, 3, 4);
graph.addEdge(2, 3, 2);

const result = graph.floydWarshall();

console.log("Shortest Path Matrix:");
for (const row of result) {
    console.log(row);
}

```

## Ford Fulkerson algorithm

### Ford Fulkerson algorithm

The Ford-Fulkerson Algorithm is an iterative method to compute the maximum flow in a flow network. It was initially designed to solve the max-flow min-cut problem, where the objective is to find the maximum amount of flow that can be sent from a designated source to a designated sink in a directed graph. The algorithm iteratively augments paths from the source to the sink, increasing the flow until it reaches its maximum value.

#### How Ford-Fulkerson Algorithm Works:

##### 1. Initialization:

- Begin with an initial flow of zero.

- Determine the residual graph, which represents the remaining capacity for each edge.

## 2. Augmenting Paths:

- Find an augmenting path from the source to the sink in the residual graph. An augmenting path is a path with available capacity on all its edges.

## 3. Flow Augmentation:

- Determine the maximum flow that can be added along the augmenting path. This is the minimum capacity value of the edges on the path.

## 4. Update Residual Graph:

- Update the residual graph by subtracting the flow added along the augmenting path and adding the reverse flow.

## 5. Repeat:

- Repeat steps 2-4 until there are no more augmenting paths.

## 6. Result:

- The final flow is the maximum flow in the network.

## Key Characteristics:

### • Residual Graph:

- The residual graph is crucial for the Ford-Fulkerson Algorithm. It represents the remaining capacity for each edge after the initial flow has been determined.

### • Augmenting Paths:

- The algorithm focuses on finding augmenting paths, paths in the residual graph with available capacity.

### • Termination:

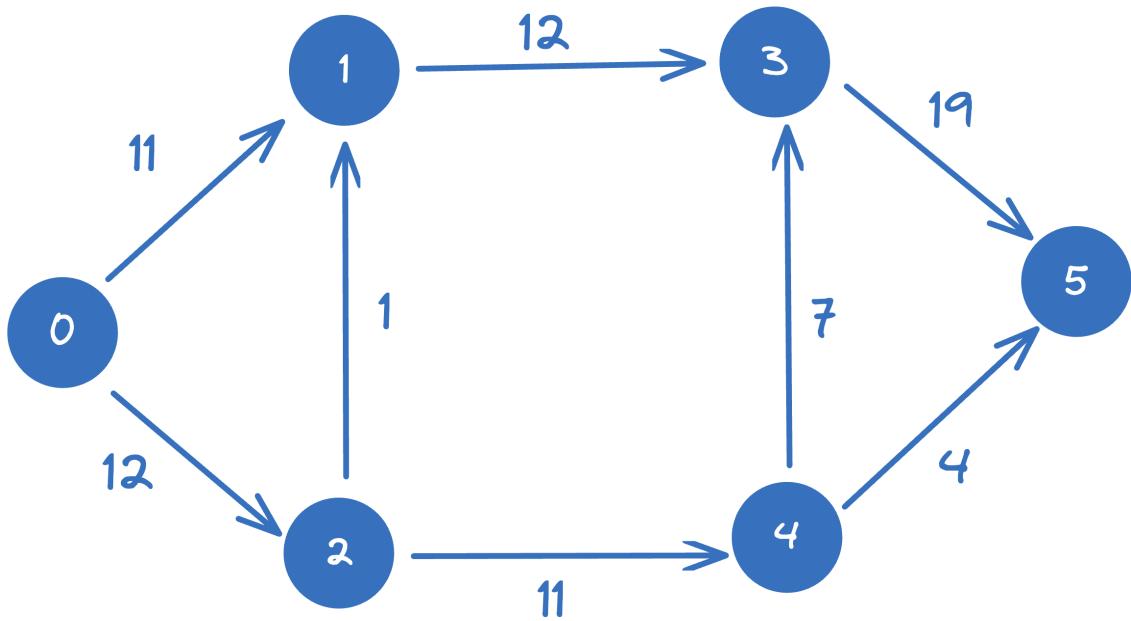
- The algorithm terminates when no more augmenting paths can be found in the residual graph.

## Applications:

- Network Flows.
- Transportation Networks.
- Telecommunication Networks.

## Time Complexity:

- The time complexity of the Ford-Fulkerson Algorithm is not strictly defined, as it depends on the choice of augmenting paths. In the worst case, the algorithm may not terminate if the paths are not chosen carefully. When implemented with the Edmonds-Karp variant, where the shortest augmenting paths are chosen using Breadth-First Search, the time complexity is  $(O(VE^2))$ , where  $(V)$  is the number of vertices and  $(E)$  is the number of edges.



```

class FordFulkerson {
    private graph: number[][];
    private numVertices: number;

    constructor(graph: number[][][]) {
        this.graph = graph;
        this.numVertices = graph.length;
    }

    fordFulkerson(source: number, sink: number): number {
        let maxFlow = 0;

        // Create a residual graph and initialize it with the original capacities.
        const residualGraph = this.graph.map((row) => [...row]);

        while (true) {
            const path = this.bfs(source, sink, residualGraph);
            if (!path) {
                break; // No augmenting path found, terminate the algorithm
            }

            // Find the minimum capacity along the augmenting path
            let minCapacity = Number.POSITIVE_INFINITY;
            for (let i = 0; i < path.length - 1; i++) {
                const u = path[i];
                const v = path[i + 1];
                minCapacity = Math.min(minCapacity, residualGraph[u][v]);
            }

            // Update residual capacities and reverse edges along the path
            for (let i = 0; i < path.length - 1; i++) {
                const u = path[i];
                const v = path[i + 1];
                residualGraph[u][v] -= minCapacity;
                residualGraph[v][u] += minCapacity;
            }
        }

        return maxFlow;
    }

    bfs(source: number, sink: number, residualGraph: number[][]): number[] | null {
        const visited = new Set();
        const queue: [number, number][] = [[source, 0]];
        const parent = new Map();

        while (queue.length) {
            const [current, distance] = queue.shift();
            if (current === sink) {
                return Array.from(parent.keys());
            }
            if (visited.has(current)) {
                continue;
            }
            visited.add(current);

            for (const neighbor of this.graph[current]) {
                if (neighbor === sink) {
                    parent.set(neighbor, current);
                } else if (residualGraph[current][neighbor] > 0) {
                    parent.set(neighbor, current);
                    queue.push([neighbor, distance + 1]);
                }
            }
        }

        return null;
    }
}

```

```

        for (let i = 0; i < path.length - 1; i++) {
            const u = path[i];
            const v = path[i + 1];
            residualGraph[u][v] -= minCapacity;
            residualGraph[v][u] += minCapacity;
        }

        // Add the flow of the augmenting path to the total flow
        maxFlow += minCapacity;
    }

    return maxFlow;
}

bfs(source: number, sink: number, graph: number[][]): number[] | null {
    const visited: boolean[] = new Array(this.numVertices).fill(false);
    const queue: number[] = [source];
    const parent: number[] = new Array(this.numVertices).fill(-1);

    while (queue.length > 0) {
        const u = queue.shift()!;

        for (let v = 0; v < this.numVertices; v++) {
            if (!visited[v] && graph[u][v] > 0) {
                queue.push(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
    }

    if (!visited[sink]) {
        return null; // No augmenting path found
    }

    const path: number[] = [];
    for (let v = sink; v !== source; v = parent[v]) {
        path.unshift(v);
    }
    path.unshift(source);

    return path;
}
}

// Example usage:
const graph = [
    [0, 16, 13, 0, 0, 0],
    [0, 0, 10, 12, 0, 0],
    [0, 4, 0, 0, 14, 0],
    [0, 0, 9, 0, 0, 20],
    [0, 0, 0, 7, 0, 4],
    [0, 0, 0, 0, 0, 0],
]

```

```
];
const fordFulkerson = new FordFulkerson(graph);
const maxFlow = fordFulkerson.fordFulkerson(0, 5);
console.log("Maximum Flow:", maxFlow);
```

Graph adjacency list

## Graph adjacency list

```
public class GraphList {

    ArrayList<LinkedList<Node>> alist;

    GraphList() {
        alist = new ArrayList<>();
    }

    public void addNode(Node node) {
        LinkedList<Node> currentList = new LinkedList<>();
        currentList.add(node);
        alist.add(currentList);
    }

    public void addEdge(int src, int dst) {
        LinkedList<Node> currentList = alist.get(src);
        Node dstNode = alist.get(dst).get(0);
        currentList.add(dstNode);

    }
    public boolean checkEdge(int src, int dst) {
        LinkedList<Node> currentList = alist.get(src);
        Node dstNode = alist.get(dst).get(0);

        for(Node node: currentList) {
            if(node == dstNode) {
                return true;
            }
        }
        return false;
    }

    public void print() {
        for(LinkedList<Node> currentList : alist) {
            for(Node node: currentList) {
                System.out.print(node.data + " -> ");
            }
            System.out.println();
        }
    }
}
```

```
    }
}
```

## Graph adjacency matrix

### Graph adjacency matrix

```
public class Graph {
    ArrayList<Node> nodes;
    int[][] matrix;

    Graph(int size) {
        nodes = new ArrayList<>();
        matrix = new int[size][size];
    }

    public void addNode(Node node) {
        nodes.add(node);
    }

    public void addEdge(int src, int dst) {
        matrix[src][dst] = 1;
    }

    public boolean checkEdge(int src, int dst) {
        if(matrix[src][dst] == 1) {
            return true;
        } else {
            return false;
        }
    }

    public void print() {
        System.out.print("  ");
        for(Node node : nodes) {
            System.out.print(node.data + " ");
        }
        System.out.println();

        for(int i = 0; i < matrix.length; i++) {
            System.out.print(nodes.get(i).data + " ");
            for(int j = 0; j < matrix[i].length; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

## **Insertion sort**

### **Insertion sort**

Insertion Sort is a straightforward sorting algorithm that builds the sorted array one element at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, it has some advantages: it is simple to implement, efficient for small datasets, and performs well for partially sorted arrays.

#### **How Insertion Sort Works:**

##### **1. Dividing the Array:**

- The algorithm starts with the first element of the array considered as the sorted part.

##### **2. Inserting Elements:**

- For each element in the unsorted part of the array, Insertion Sort compares it with the elements in the sorted part.
- It then inserts the element into its correct position in the sorted part, shifting the other elements if necessary.

##### **3. Iterative Process:**

- This process is repeated until all elements are sorted.

#### **Time Complexity:**

- Insertion Sort has a time complexity of  $O(n^2)$  in the worst case, where 'n' is the number of elements in the array. Despite its quadratic time complexity, Insertion Sort is often more efficient on small datasets or partially sorted arrays compared to other quadratic sorting algorithms. It's also an in-place sorting algorithm, meaning it doesn't require additional memory.



```
function insertionSort(array: number[] | string[]) {
    for (let i = 1; i < array.length; i++) {
        let curr = array[i];
        let j = i - 1;
        for (j; j >= 0 && array[j] > curr; j--) {
            array[j + 1] = array[j];
        }
        array[j + 1] = curr;
    }
    return array;
}
```

```

console.log(insertionSort([1, 4, 2, 8, 345, 123, 43, 32, 5643, 63, 123, 43, 2, 55, 1, 234, 92]));

class Solution {
    void insertionSort (int[] arr) {
        int n = arr.length;
        for(int i = 1; i < n; i++) {
            int current = arr[i];
            int position = i - 1;
            while(position >= 0 && arr[position] > current) {
                arr[position + 1] = arr[position];
                position--;
            }
            arr[position + 1] = current;
        }
    }
}

```

## Interpolation search

# Interpolation search

Interpolation Search is a an algorithm designed for finding a specific target value in a sorted array. Unlike linear or binary search, this algorithm utilizes the characteristics of the data distribution to make more informed decisions about where to look for the target. It is particularly effective when the data has a uniform distribution.

### How Interpolation Search Works:

#### 1. Linear Interpolation:

- Interpolation Search utilizes linear interpolation to estimate the likely position of the target value in the array.

#### 2. Estimate Position:

- Instead of evenly dividing the search space, as in binary search, Interpolation Search estimates the probable position of the target based on its value relative to the minimum and maximum values in the array.

#### 3. Calculation of Position:

- It calculates an estimate of the target's position by considering the relative location of the target with respect to the minimum and maximum values in the current search space.

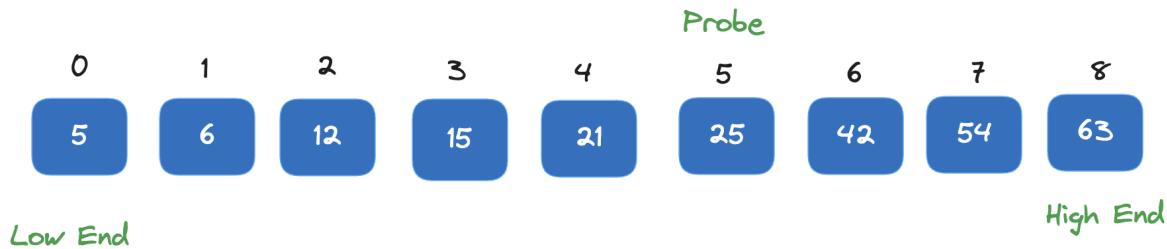
#### 4. Refine Search:

- Based on the calculated estimate, the algorithm narrows down the search space and repeats the process until the target is found or the search space is exhausted.

### Time Complexity:

The time complexity of Interpolation Search is  $O(\log \log n)$  on average, where "n" is the number of elements in the array. In the best case, it can be  $O(1)$ , and in the worst case, it can be  $O(n)$ . However, the average

case is often more relevant, and it is  $O(\log \log n)$  under certain assumptions about the distribution of the data.



```
class Solution {

    private static int interpolationSearch(int[] array, int value) {
        int low = 0;
        int high = array.length - 1;

        while(value >=array[low] && value <= array[high] && low <= high) {
            int probe = low + (high - low) * (value - array[low]) / (array[high] - array[low]);
            if(array[probe] == value) {
                return probe;
            } else if(array[probe] > value) {
                low = probe + 1;
            } else {
                high = probe -1;
            }
        }

        return -1;
    }
}
```

```
function interpolationSearch(array: number[], value: number): number {
    let low = 0;
    let high = array.length - 1;

    while (value >= array[low] && value <= array[high] && low <= high) {
        const probe =
            low + ((high - low) * (value - array[low])) / (array[high] - array[low]);
        const roundedProbe = Math.floor(probe);

        if (array[roundedProbe] === value) {
            return roundedProbe;
        } else if (array[roundedProbe] < value) {
            low = roundedProbe + 1;
        } else {
            high = roundedProbe - 1;
        }
    }
}
```

```
    return -1;
}
```

## Interval search

# Interval search

An interval search algorithm typically refers to searching for overlapping or containing intervals in a collection of intervals. One common approach for this task is to use an interval tree. Here's an explanation of the Interval Search algorithm using an interval tree:

### How Interval Search Works:

#### 1. Construct the Interval Tree:

- Begin by constructing an interval tree from the given set of intervals.
- Each node in the interval tree represents an interval, and the tree is recursively built to efficiently organize and store these intervals.

#### 2. Search for Overlapping Intervals:

- When searching for intervals that overlap with a given interval (query interval), start at the root of the interval tree.

#### 3. Traverse the Tree:

- Traverse the tree, comparing the query interval with the intervals represented by each node.
- If there is an overlap, the algorithm can either return the overlapping interval(s) immediately or continue searching in both left and right subtrees.

#### 4. Recursive Search:

- Recursively search in the left or right subtree based on the relationship between the query interval and the intervals represented by the current node.
- Continue this process until all potential overlapping intervals are found.

### Time Complexity:

- The time complexity of searching for overlapping intervals using an interval tree is typically  $O(\log n + k)$ , where ' $n$ ' is the number of intervals in the tree and ' $k$ ' is the number of intervals overlapping with the query interval. The construction of the interval tree initially takes  $O(n \log n)$  time, but subsequent searches are more efficient. Interval trees are particularly useful when there are many queries for overlapping intervals in a set.

```
type Interval = [number, number];

function intervalSearch(intervals: Interval[], queryInterval: Interval): number[] {
  const result: number[] = [];

  for (let i = 0; i < intervals.length; i++) {
    const [start, end] = intervals[i];
    const [queryStart, queryEnd] = queryInterval;
```

```

        if (start <= queryEnd && end >= queryStart) {
            result.push(i);
        }
    }

    return result;
}

```

## Jump search

# Jump search

Jump Search is a searching algorithm designed for sorted arrays. It is a block-based search algorithm that works by jumping ahead by fixed steps and then linearly searching within the block for the target element. Jump Search combines the efficiency of binary search with the simplicity of linear search.

### How Jump Search Works:

#### 1. Determine Jump Size:

- Determine the jump size by taking the square root of the array length. This ensures a balanced trade-off between the number of jumps and the linear search within a block.

#### 2. Jump Ahead:

- Start at the beginning of the array and jump ahead by the calculated jump size until finding a value that is greater than or equal to the target.

#### 3. Linear Search within Block:

- Perform a linear search within the block from the previous jump until finding the target element or determining that it is not present in the block.

#### 4. Repeat or Conclude:

- Repeat the process until the entire array is searched or the target element is found.

## Key Characteristics:

### • Efficiency:

- Jump Search has a time complexity of  $\sqrt{n}$ , making it more efficient than linear search ( $n$ ) and comparable to binary search ( $\log n$ ) for large datasets.

### • Sorted Arrays:

- Jump Search requires the array to be sorted.

### • Jump Size:

- The jump size is a critical factor in the efficiency of Jump Search. The optimal jump size is often calculated as the square root of the array length.

## Applications:

- Database Searching:

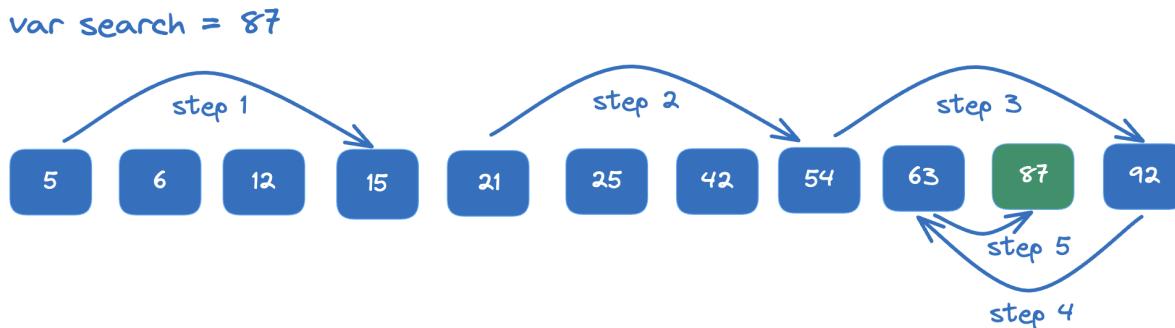
- Jump Search is used in database systems for searching in large datasets.

- Sorted Arrays:

- Useful when the data is sorted, and random access to elements is feasible.

## Time Complexity:

- The time complexity of Jump Search is  $\text{add formula-}$ , where  $\text{add formula-}$  is the size of the array. This makes it efficient for large datasets when compared to linear search but may be outperformed by binary search for certain scenarios.



```
function jumpSearch(arr: number[], target: number): number {
    const n = arr.length;
    const step = Math.floor(Math.sqrt(n));
    let prev = 0;

    while (arr[Math.min(step, n) - 1] < target) {
        prev = step;
        step += Math.floor(Math.sqrt(n));
        if (prev >= n) {
            return -1;
        }
    }

    for (let i = prev; i < Math.min(step, n); i++) {
        if (arr[i] === target) {
            return i;
        }
    }

    return -1;
}
```

## **Linear search**

# **Linear search**

Linear Search, also known as sequential search, is a simple searching algorithm that finds the position of a target value within a list or array. It works by iterating through the elements one by one until the target value is found or the entire list has been searched.

### **How Linear Search Works:**

#### **1. Start at the Beginning:**

- Linear Search begins by looking at the first element in the list.

#### **2. Compare with Target:**

- It compares the current element with the target value that we are searching for.

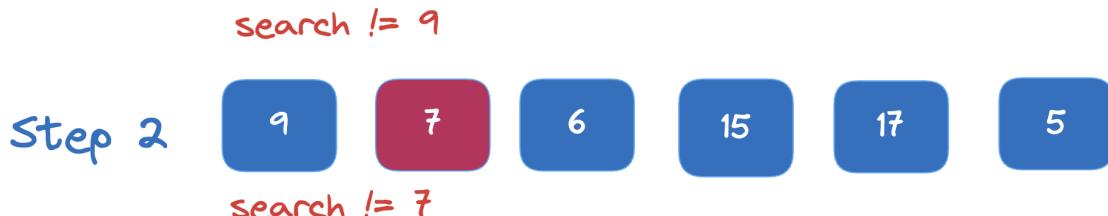
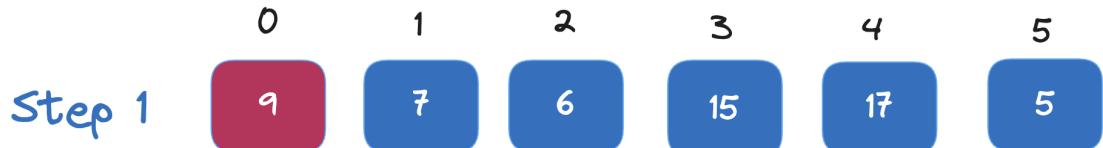
#### **3. Search Iteratively:**

- If the current element is equal to the target value, the search is successful, and the index or position of the element is returned.
- If the current element is not equal to the target value, the search continues by moving to the next element in the list.
- This process is repeated until either the target value is found or the end of the list is reached.

### **Time Complexity:**

The time complexity of Linear Search is  $O(n)$ , where ' $n$ ' is the number of elements in the array. In the worst case, the algorithm may need to iterate through the entire list to find the target value. While Linear Search is simple, it may not be the most efficient for large datasets, especially when compared to more advanced search algorithms like binary search on sorted lists. However, it is easy to understand and implement.

`var search = 15`



```
function linearSearch(arr: number[], target: number): number {
    for (let i = 0; i < arr.length; i++) {
        if (arr[i] === target) {
            return i;
        }
    }
    return -1;
}
```

Merge sort

## Merge sort

Merge Sort is a comparison-based sorting algorithm that follows the divide-and-conquer paradigm. It works by dividing the unsorted array into 'n' sub-arrays, each containing one element. It then repeatedly merges these sub-arrays to produce new sorted sub-arrays until there is only one sub-array remaining – the fully sorted array.

How Merge Sort Works:

1. Divide:

- The unsorted array is recursively divided into two halves until each sub-array contains only one element. This is the base case of the recursion.

## 2. Conquer:

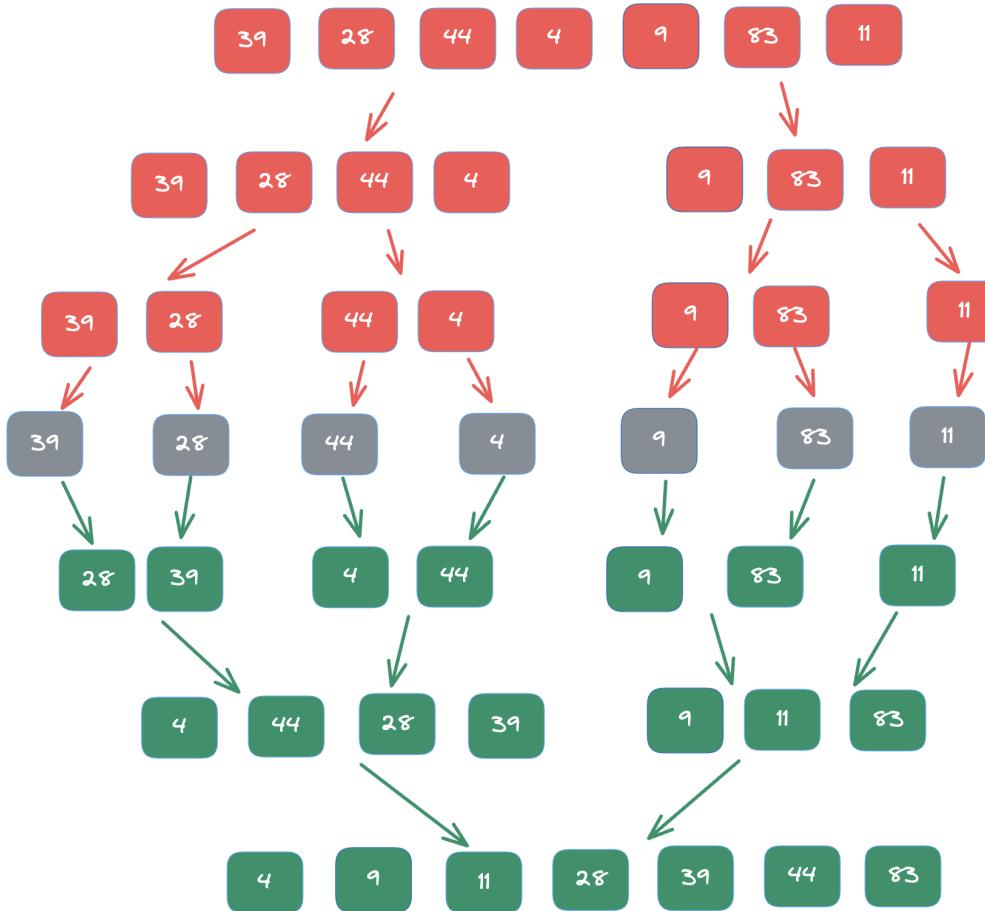
- The adjacent sub-arrays are then recursively merged to produce new sorted sub-arrays. This process continues until there is only one sub-array remaining – the fully sorted array.

## 3. Merge:

- The key operation in Merge Sort is the merging of two sorted sub-arrays to produce a single, sorted sub-array. This involves comparing elements from the two sub-arrays and placing them in the correct order.

### Time Complexity:

- Merge Sort has a consistent time complexity of  $O(n \log n)$  in all cases, where ' $n$ ' is the number of elements in the array. It is a stable sorting algorithm, meaning that equal elements maintain their relative order in the sorted output. While Merge Sort has a slightly higher space complexity due to the need for additional memory, its stability and predictable performance make it a widely used and reliable sorting algorithm.



```

class Solution {

    void merge(int[] arr, int low, int mid, int high) {
        int subArr1Size = mid - low + 1;
        int subArr2Size = high - mid;

        int [] subArr1 = new int[subArr1Size];
        int [] subArr2 = new int[subArr2Size];

        for (int i = 0; i < subArr1Size; i++) {
            subArr1[i] = arr[low + i];
        }
        for (int i = 0; i < subArr2Size; i++) {
            subArr2[i] = arr[mid + 1 + i];
        }
        int i = 0, j = 0, k = low;

        while(i < subArr1Size && j < subArr2Size) {
            if(subArr1[i] <= subArr2[j]) {
                arr[k] = subArr1[i];
                i++;
            } else {
                arr[k] = subArr2[j];
                j++;
            }
            k++;
        }
        while(i < subArr1Size) {
            arr[k++] = subArr1[i++];
        }
        while (j < subArr2Size) {
            arr[k++] = subArr2[j++];
        }
    }

    void mergesort(int[] arr, int low, int high){
        if(high > low) {
            int mid = (high + low) / 2;
            mergesort(arr, low, mid);
            mergesort(arr, mid + 1, high);
            merge(arr, low, mid, high);
        }
    }

    void mergeSort (int[] arr) {
        int n = arr.length;
        mergesort(arr, 0, n - 1);
    }
}

```

```

function mergeSort(arr: number[]): number[] {
    if (arr.length <= 1) {
        return arr;
    }
}

```

```

}

const middle = Math.floor(arr.length / 2);
const left = arr.slice(0, middle);
const right = arr.slice(middle);

return merge(mergeSort(left), mergeSort(right));
}

function merge(left: number[], right: number[]): number[] {
  let result: number[] = [];
  let leftIndex = 0;
  let rightIndex = 0;

  while (leftIndex < left.length && rightIndex < right.length) {
    if (left[leftIndex] < right[rightIndex]) {
      result.push(left[leftIndex]);
      leftIndex++;
    } else {
      result.push(right[rightIndex]);
      rightIndex++;
    }
  }

  return result.concat(left.slice(leftIndex)).concat(right.slice(rightIndex));
}

```

## Quick sort

# Quicksort

Quick Sort is an efficient, comparison-based sorting algorithm that follows the divide-and-conquer paradigm. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted.

### How Quick Sort Works:

#### 1. Choosing a Pivot:

- The algorithm selects a pivot element from the array. The choice of pivot can affect the efficiency of the algorithm.

#### 2. Partitioning:

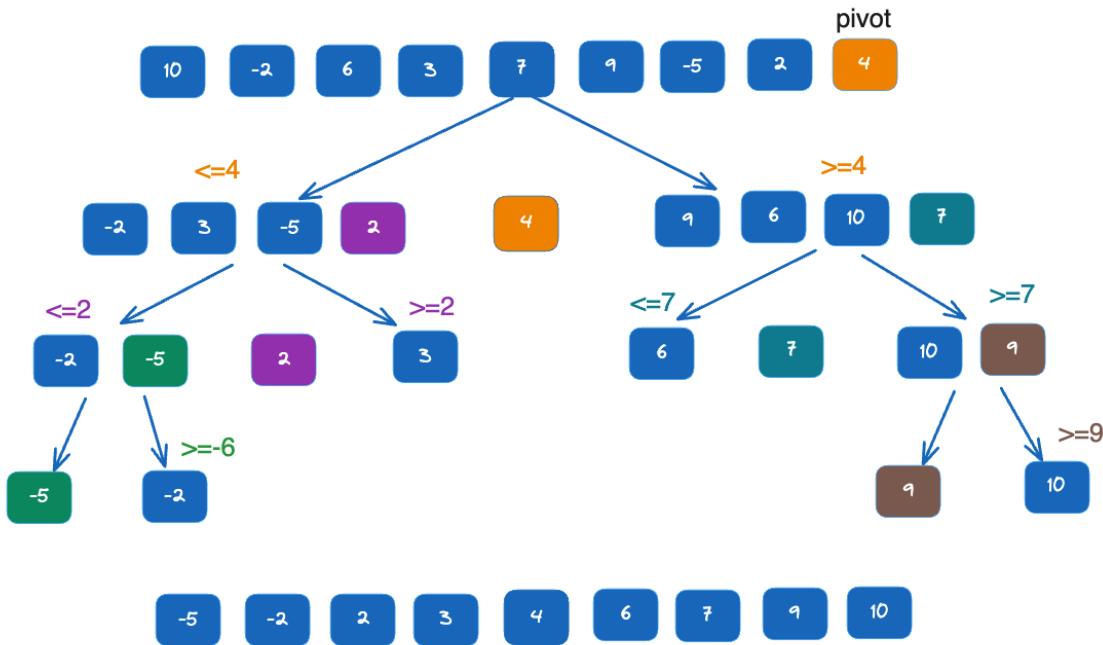
- Elements smaller than the pivot are moved to its left, and elements greater than the pivot are moved to its right. The pivot is now in its final sorted position.

#### 3. Recursive Sorting:

- The algorithm is applied recursively to the sub-arrays on the left and right of the pivot until the entire array is sorted.

### Time Complexity:

- Quick Sort has an average and best-case time complexity of  $O(n \log n)$ , where ' $n$ ' is the number of elements in the array. In the worst case, it is  $O(n^2)$ , but this is rare when a good pivot selection strategy is used. Quick Sort is often faster in practice than other  $O(n \log n)$  algorithms, and it is widely used in various applications due to its efficiency.



```

class Solution {

    int makePartition(int [] arr, int low, int high) {
        int pivot = arr[high];
        int currentIndex = low - 1;
        for(int i = low; i < high; i++) {
            if(arr[i] < pivot) {
                currentIndex++;
                int temp = arr[i];
                arr[i] = arr[currentIndex];
                arr[currentIndex] = temp;
            }
        }
        int temp = arr[high];
        arr[high] = arr[currentIndex + 1];
        arr[currentIndex + 1] = temp;
        return currentIndex + 1;
    }

    void quicksort(int[] arr, int low, int high) {
        if(low < high) {
            int pivot = makePartition(arr, low, high);
            quicksort(arr, low, pivot - 1);
            quicksort(arr, pivot + 1, high);
        }
    }
}

```

```

        }
    }

    void quickSort (int[] arr) {
        int n = arr.length;
        quicksort(arr, 0, n - 1);
    }
}

def quicksort(arr):
    if len(arr) < 2:
        return arr
    else:
        pivot = arr[len(arr)/2]
        less = [i for i in arr[1:] if i <= pivot]
        greater = [i for i in arr[1:] if i > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

print(quicksort([10,2,3,1,5,4]))

```

```

class Solution {
    static void swap(int[] array, int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    private static void quickSort(int[] array, int start, int end) {
        if(end <= start) return; // base case

        int pivot = partition(array, start, end);

        quickSort(array, start, pivot -1);
        quickSort(array, pivot + 1, end);
    }

    private static int partition(int[] array, int start, int end) {
        int pivot = array[end];

        int i = start - 1;

        for(int j = start; j <= end -1; j++) {
            if(array[j] < pivot) {
                i++;
                swap(array, i, j);
            }
        }
        i++;
        swap(array, i, end);

        return i;
    }
}

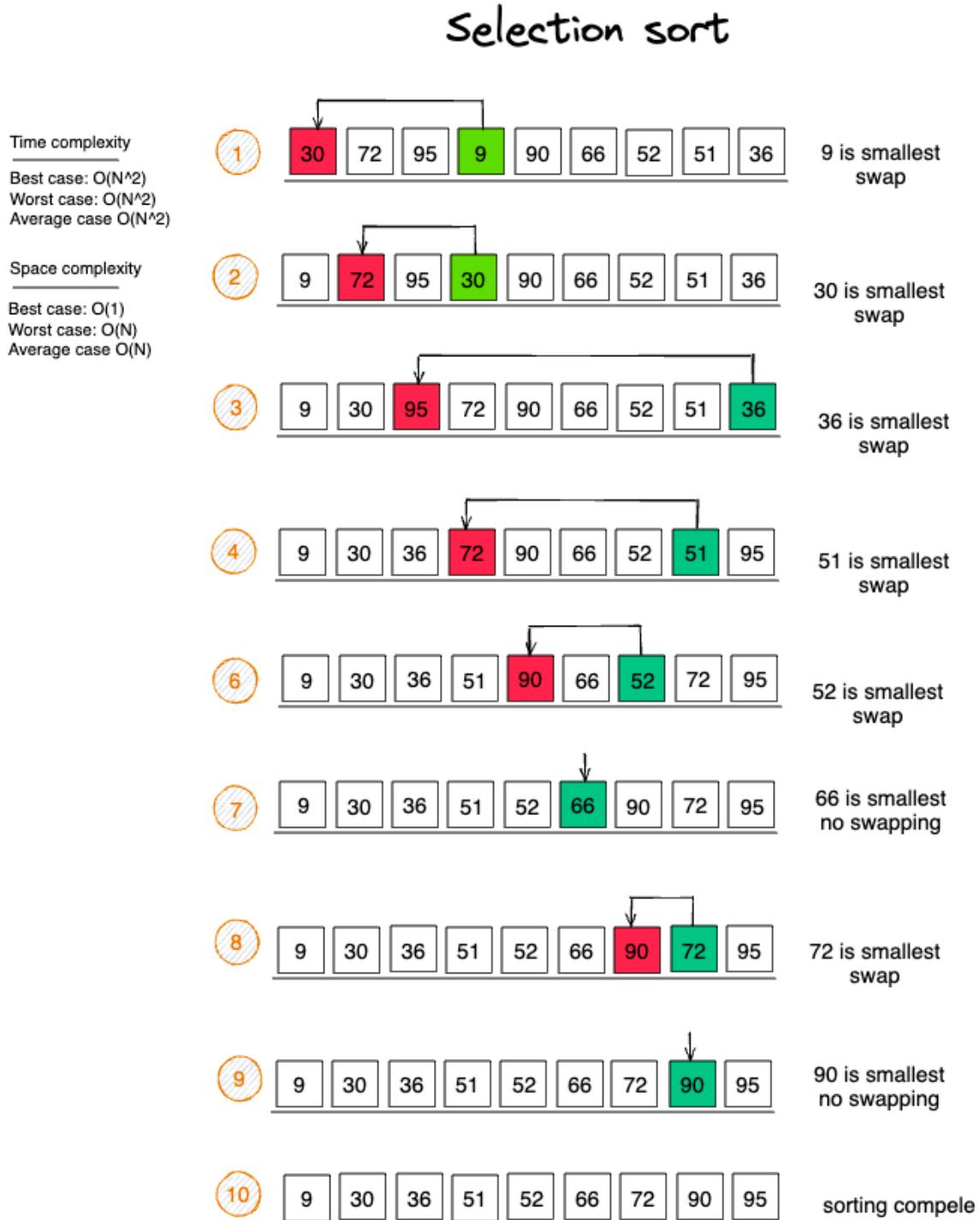
```

```
function quicksort(arr: number[]): number[] {
  if (arr.length < 2) {
    return arr;
  } else {
    const pivot = arr[Math.floor(arr.length / 2)];
    const less = arr.slice(1).filter((i) => i <= pivot);
    const greater = arr.slice(1).filter((i) => i > pivot);
    return [...quicksort(less), pivot, ...quicksort(greater)];
  }
}
```

- Go back

## Selection sort

### Selection sort



Selection Sort is a straightforward sorting algorithm that works by dividing the input array into two parts: the

sorted and the unsorted subarrays. The algorithm repeatedly selects the minimum (or maximum, depending on the sorting order) element from the unsorted subarray and swaps it with the first unsorted element. This process is iteratively applied until the entire array is sorted.

### How Selection Sort Works:

#### 1. Dividing the Array:

- The algorithm starts with the entire array considered as unsorted.

#### 2. Finding the Minimum Element:

- In each iteration, Selection Sort finds the minimum element from the unsorted part of the array.

#### 3. Swapping:

- Once the minimum element is identified, it is swapped with the first element in the unsorted part, effectively extending the sorted subarray.

#### 4. Iterative Process:

- The above steps are repeated for the remaining unsorted part of the array until the entire array is sorted.

```
function selectionSort(array: any[]) {  
    for (let i = 0; i < array.length - 1; i++) {  
        let min = i;  
        for (let j = i + 1; j < array.length; j++) {  
            if (array[min] > array[j]) min = j;  
        }  
        [array[i], array[min]] = [array[min], array[i]]  
    }  
    return array;  
}  
  
console.log(selectionSort([1, 4, 2, 8, 345, 123, 43, 32, 5643, 63, 123, 43, 2, 55, 1, 234, 92]));
```

```
public static void selectionSort(int[] array) {  
    for(int i = 0; i < array.length - 1; i++) {  
        int min = i;  
        for(int j = i + 1; j < array.length; j++) {  
            if(array[min] > array[j]) {  
                min = j;  
            }  
        }  
        int temp = array[i];  
        array[i] = array[min];  
        array[min] = temp;  
    }  
}
```

```
print('This is selection sort')  
  
def find_smallest(arr):  
    smallest = arr[0]
```

```

smallest_index = 0
for i in range(1, len(arr)):
    if arr[i] < smallest:
        smallest = arr[i]
        smallest_index = i
return smallest_index

def selection_sort(arr):
    newArr = []
    for i in range(len(arr)):
        smallest = find_smallest(arr)
        newArr.append(arr.pop(smallest))
    return newArr

print(selection_sort([5,4,6,2,1,123, 2, 3,1,23 ,1,1,]))

```

## Ternary search

# Ternary search

Ternary Search is a divide-and-conquer algorithm designed for efficiently finding the position of a target value in a sorted array. It operates by dividing the array into three parts and recursively narrowing down the search space until the target is found or determined to be absent.

### How Ternary Search Works:

#### 1. Divide the Array:

- Ternary Search starts by dividing the sorted array into three parts.

#### 2. Compare with the Target:

- It then compares the target value with the elements at two points within the array, dividing it into three segments.
- If the target is found at one of these points, the search is successful.

#### 3. Determine Search Space:

- Based on the comparisons, Ternary Search identifies whether the target lies in the first, second, or third segment of the array.

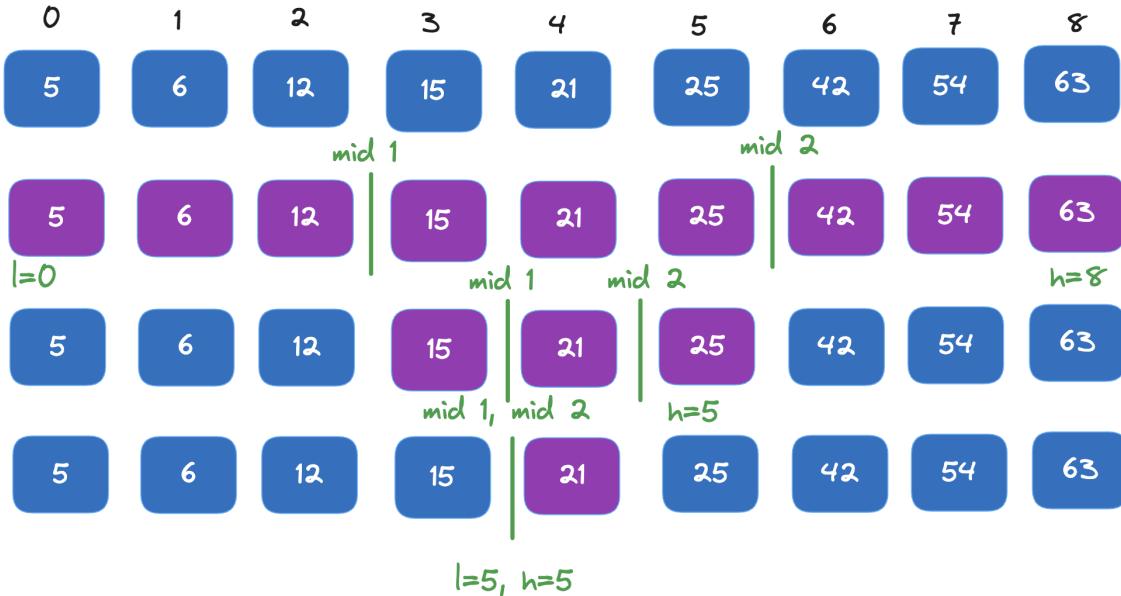
#### 4. Recursive Search:

- The algorithm then recursively applies the same process to the identified segment.
- This recursion continues until the target is found or the search space is reduced to an empty array, indicating that the target is not present.

### Time Complexity:

- Ternary Search has a time complexity of  $O(\log_3 n)$ , where ' $n$ ' is the size of the array. This is an improvement over binary search when the search space can be significantly reduced at each step. However, it's worth noting that constant factors play a role, and in practice, binary search might be faster for smaller datasets due to simpler arithmetic operations. Ternary Search is particularly beneficial when the dataset is large and the search space can be significantly reduced with each iteration.

`var search = 21`



```
function ternarySearch(func: (x: number) => number, left: number, right: number, epsilon: number): number {
  while (right - left > epsilon) {
    const mid1 = left + (right - left) / 3;
    const mid2 = right - (right - left) / 3;

    const value1 = func(mid1);
    const value2 = func(mid2);

    if (value1 < value2) {
      left = mid1;
    } else {
      right = mid2;
    }
  }

  return (left + right) / 2;
}
```

javascript

Array length property

## Array length property

What is the value of clothes[0]:

```
const clothes = ["jacket", "t-shirt"];
clothes.length = 0;
clothes[0];
```

Reducing the value of the length property has the side-effect of deleting own array elements whose array index is between the old and new length values. <https://262.ecma-international.org/6.0/#sec-properties-of-array-instances-length>

As result when JavaScript executes clothes.length = 0, all clothes items are deleted.

clothes[0] is undefined, because clothes array has been emptied.

- Go back

Different ways of declaring functions in JS

Different ways of declaring functions in JS

Different ways of declaring  
a function in JS

Function expression

Anonymous Functions

Immediately Invoked  
Function Expressions

Arrow Functions

Constructor Functions

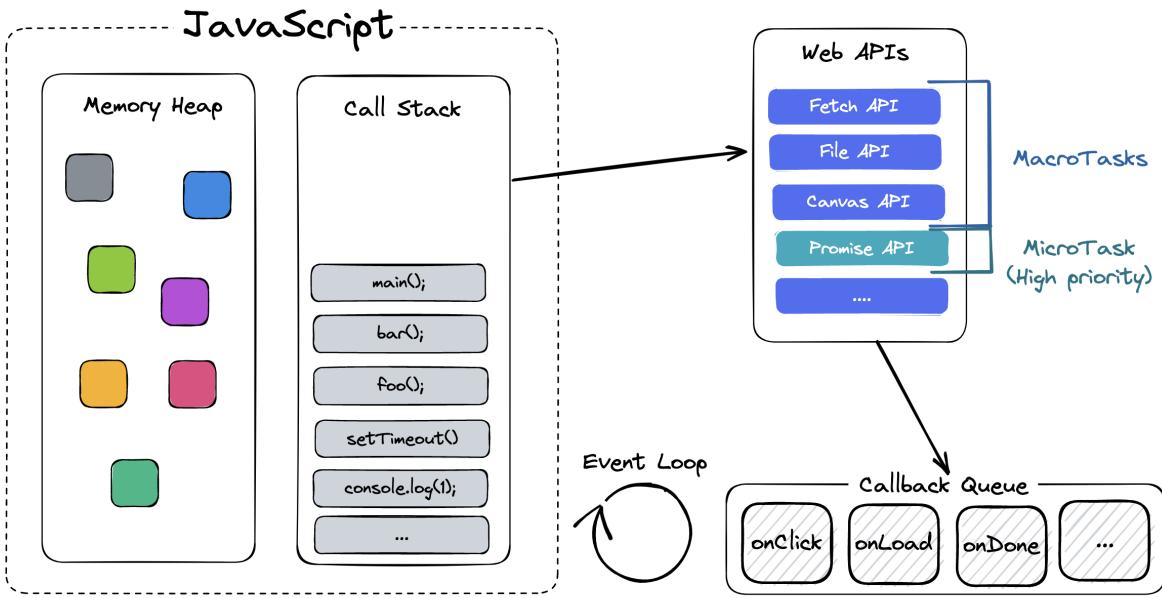
Getter Functions

Hoisting

- Go back

## Event Loop

### Event Loop



In JavaScript, the event loop is a fundamental concept for managing asynchronous operations. To understand the event loop, it's essential to grasp the other components involved:

1. **Memory Heap:** This is where memory allocation happens. Objects and variables are stored here.
2. **Call Stack:** It keeps track of the execution of functions. When a function is invoked, it's added to the call stack. When the function finishes executing, it's removed from the stack.
3. **Web APIs:** These are provided by the browser, such as DOM manipulation methods, `setTimeout`, `XMLHttpRequest`, etc. They are not part of the JavaScript language itself but are accessible through JavaScript. When you make an asynchronous request, it's handled by the Web APIs.
4. **Event Loop:** The event loop constantly checks the call stack and the callback queue to see if there's any function that needs to be executed. If the call stack is empty, it takes a callback from the queue and pushes it onto the call stack, initiating its execution.
5. **Callback Queue:** When asynchronous operations (like `setTimeout` or `XMLHttpRequest`) are completed, their callback functions are pushed into the callback queue.
6. **Macrotasks:** These are higher priority tasks in the callback queue. They include I/O operations, `setTimeout`, `setInterval`, etc. Macrotasks are processed after the call stack is empty.
7. **Microtasks:** Microtasks are tasks with a higher priority than macrotasks and include processes like Promise callbacks, `queueMicrotask`, `MutationObserver`, etc. Microtasks are executed after the current task and before the next macrotask.

Here's how they work together:

1. Functions are pushed onto the call stack and executed synchronously until the stack is empty.
2. Asynchronous functions are offloaded to the Web APIs.
3. When the asynchronous operation is completed, its callback function is placed in the callback queue.
4. The event loop checks the callback queue. If the call stack is empty, it pushes the callback function from the queue onto the stack, executing it.
5. Microtasks are executed between macrotasks, ensuring that certain asynchronous operations are prioritized and processed promptly.

This cycle allows JavaScript to handle asynchronous tasks efficiently, providing a smooth user experience in web applications.

- Go back

#### Give best practices for handling errors in asynchronous code.

### Give best practices for handling errors in asynchronous code.

Handling errors in asynchronous code is a critical aspect of developing reliable and secure applications. Here are some best practices for handling errors in asynchronous code:

1. **Use try-catch blocks:** Wrap asynchronous operations within a `try` block and handle errors within the corresponding `catch` block. This allows you to catch synchronous errors that might occur in asynchronous code.

```
try {
  // Asynchronous operation
} catch (error) {
  // Handle the error
}
```

2. **Use callbacks with the first argument as an error:** When writing functions with callbacks, always pass the error as the first argument in the callback. Functions typically follow a calling convention like `callback(error, result)`. If `error` is not `null` or `undefined`, an error has occurred.

```
function asyncFunction(callback) {
  // Asynchronous operation
  if (error) {
    return callback(error);
  }
  callback(null, result);
}
```

3. **Use Promises:** If possible, use Promises to perform asynchronous operations. Promises provide a convenient way to handle successful results and errors.

```
asyncFunction()
  .then((result) => {
    // Handle the successful result
  })
```

```
.catch((error) => {
  // Handle the error
});
```

4. **Use `async/await`:** With asynchronous functions in JavaScript, you can use the `async` and `await` keywords for a more convenient syntax when working with asynchronous operations and errors.

```
async function fetchData() {
  try {
    let data = await asyncFunction();
    // Handle the successful result
  } catch (error) {
    // Handle the error
  }
}
```

5. **Do not ignore errors:** Never ignore errors. Always handle them or propagate them up the call stack to be handled at a higher level.
6. **Log errors:** Log errors to quickly detect and troubleshoot issues in production. A robust logging system will help you track errors in your application.
7. **Use specific error classes:** Create custom error classes for different types of errors. This helps you understand the nature of the error and handle them more precisely.

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = "CustomError";
  }
}
```

Adhering to these practices will help you manage errors in asynchronous code more effectively and build more reliable applications.

**How can JavaScript codes be hidden from old browsers that do not support JavaScript?**

## **How can JavaScript codes be hidden from old browsers that do not support JavaScript?**

1. **Add `<!--` before the `<script>` tag:** This starts an HTML comment, which is ignored by old browsers that do not understand JavaScript.

```
<!--
<script type="text/javascript">
// JavaScript code here
</script>
-->
```

2. **Add `//-->` after the `<script>` tag:** This ends the HTML comment for old browsers. Modern browsers will treat `//-->` as a one-line JavaScript comment, effectively ignoring it.

```
<!--  
<script type="text/javascript">  
// JavaScript code here  
</script>  
//-->
```

By using this technique, the JavaScript code is effectively hidden from old browsers, but it will still be executed by modern browsers that support JavaScript. Keep in mind that this method is somewhat outdated, and modern web development practices often involve using feature detection and progressive enhancement to handle different browser capabilities.

## How JavaScript Engine works

# How JavaScript Engine works

JavaScript engines are responsible for executing JavaScript code on web browsers and other environments. They follow a multi-step process to parse, compile, optimize, and execute JavaScript code efficiently. Here's a simplified overview of how a typical JavaScript engine works:

### 1. Lexical Analysis:

When you load a web page, the JavaScript engine first performs lexical analysis or tokenization. This involves breaking the source code into tokens, which are the smallest units of meaning in the language. Tokens include keywords, operators, literals, and identifiers.

### 2. Parsing:

The tokens are then parsed to create an Abstract Syntax Tree (AST). The AST represents the grammatical structure of the code. Parsing involves organizing the tokens into a hierarchical tree that defines the order of operations and relationships between different parts of the code.

### 3. Compilation:

Once the AST is created, it goes through a compilation process. In this step, the engine translates the AST into intermediate code. This code is also known as bytecode or machine code. Some engines use Just-In-Time (JIT) compilation, where the intermediate code is further optimized for execution.

### 4. Optimization:

Modern JavaScript engines employ various optimization techniques to improve the performance of the code. These include inline caching, function inlining, loop unrolling, and many others. Engines can also analyze the code during execution and apply optimizations dynamically based on the actual usage patterns.

### 5. Execution:

The optimized intermediate code is executed by the engine. During execution, the engine manages the call stack, memory allocation, and variable scope. It interprets the bytecode and performs the operations specified by the code.

## **6. Garbage Collection:**

JavaScript engines have automatic memory management, including garbage collection. Unused objects and variables are identified and deallocated to free up memory. This process ensures efficient memory usage and prevents memory leaks.

## **7. Callbacks and Web APIs (in browser environments):**

In browser environments, JavaScript can interact with the DOM (Document Object Model) and perform asynchronous operations using callbacks and Web APIs (such as `setTimeout`, `fetch`, and `XMLHttpRequest`). These operations are offloaded to the browser's internal components and are executed asynchronously, allowing the main JavaScript thread to remain responsive.

## **8. Event Loop (in asynchronous environments):**

In asynchronous environments like web browsers, JavaScript uses an event loop to handle asynchronous operations. The event loop continuously checks the message queue for pending events (such as user interactions or completed network requests) and processes them, ensuring that asynchronous tasks are executed when their results are ready.

Different JavaScript engines might implement these steps differently, but the overall process remains similar. Understanding these fundamental concepts can help developers write more efficient JavaScript code and optimize their applications for better performance.

**How to get argv in nodejs?**

## **How to get argv in nodejs?**

In Node.js, you can access command-line arguments using the `process.argv` array. The `process.argv` array contains the command-line arguments that were passed when the Node.js process was launched. Here's how you can access and use `process.argv`:

### **1. Basic Usage:**

`process.argv` is an array where the first two elements are:

- `process.argv[0]`: The path to the Node.js executable
- `process.argv[1]`: The path to the JavaScript file being executed

The actual command-line arguments start from `process.argv[2]` onwards. For example:

```
// app.js
console.log(process.argv);
```

If you run this script from the command line with additional arguments:

```
node app.js arg1 arg2 arg3
```

The output will be an array:

```
[  
  '/path/to/node',  
  '/path/to/app.js',  
  'arg1',  
  'arg2',  
  'arg3'  
]
```

## 2. Parsing Command-Line Arguments:

If you want to extract specific arguments, you can access them by their index in the `process.argv` array. For example, if you want to access '`arg1`' from the above example, you would use `process.argv[2]`:

```
// app.js  
const arg1 = process.argv[2];  
console.log('Argument 1:', arg1);
```

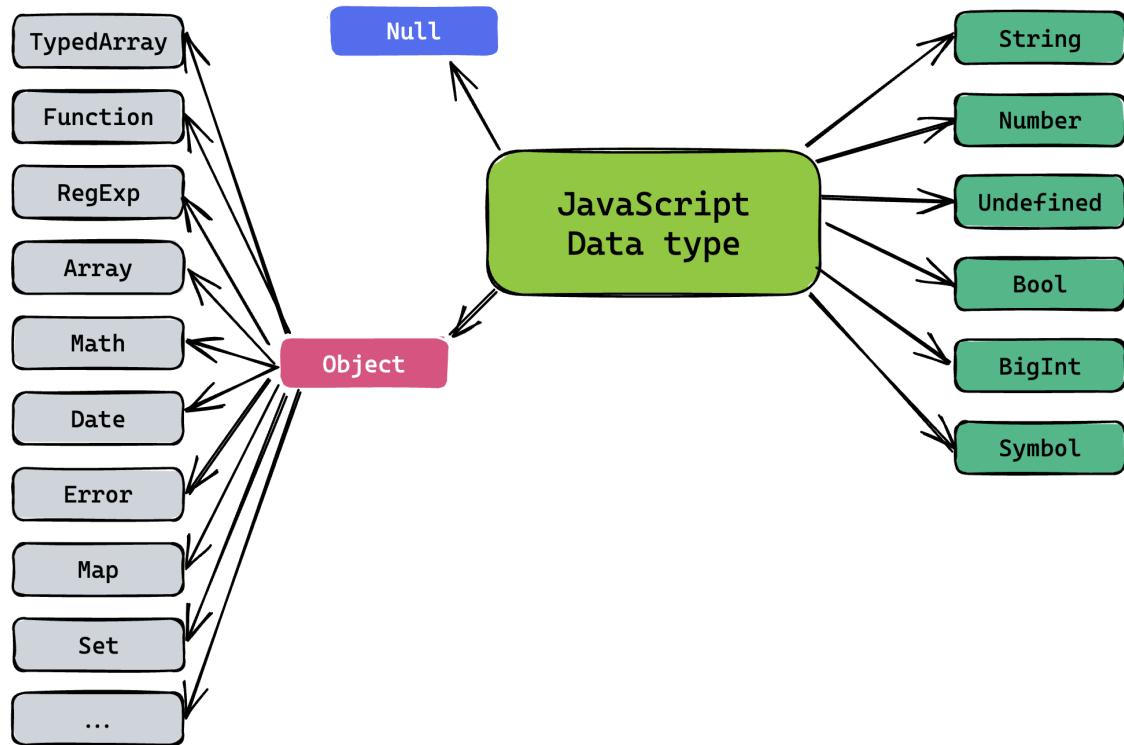
Running the script with the same command as before will output:

```
Argument 1: arg1
```

If you need to parse command-line arguments in a more structured way, you might want to consider using libraries like `yargs` or `commander`, which provide a more user-friendly and powerful interface for handling command-line arguments in Node.js applications. These libraries make it easier to define options, handle flags, and manage arguments in a more intuitive manner.

## JavaScript data types

### JavaScript data types



JavaScript number size summary

JavaScript number size summary

## JavaScript Number Size Summary

`Infinity`

`Number.POSITIVE_INFINITY`

`2^1024 - 1`

`Number.MAX_VALUE`

`9007199254740991`

`Number.MAX_SAFE_INTEGER`

`0`

`0`

`2^-1074`

`Number.MIN_VALUE`

`-9007199254740991`

`Number.MIN_SAFE_INTEGER`

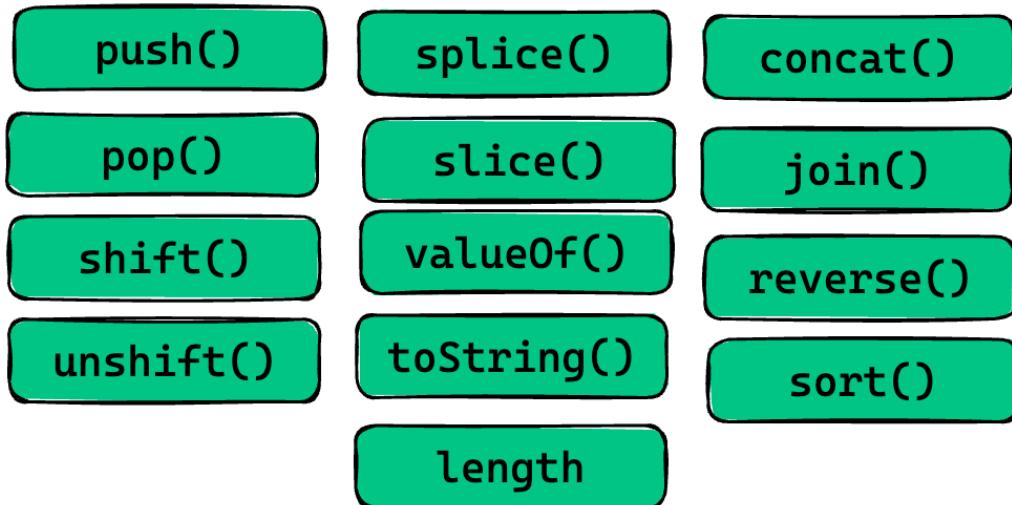
`-Infinity`

`Number.NEGATIVE_INFINITY`

## JavaScript Arrays cheat sheet

## JavaScript Arrays cheat sheet

# JS Arrays



## OOP in JavaScript

## OOP in JavaScript

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes for organizing code. JavaScript, although primarily a prototype-based language, supports object-oriented programming concepts. Here's how OOP concepts are implemented in JavaScript:

### 1. Objects:

In JavaScript, everything is an object (or a primitive data type). Objects are collections of key-value pairs. You can create objects using object literals or constructor functions.

```
// Object literal
let person = {
  name: "John",
  age: 30,
  sayHello: function() {
    console.log("Hello!");
  }
};
```

```
// Constructor function
function Person(name, age) {
    this.name = name;
    this.age = age;
}

let person1 = new Person("Alice", 25);
```

## 2. Classes (Introduced in ECMAScript 6):

JavaScript introduced the `class` keyword in ECMAScript 6, providing syntactical sugar for constructor functions. Classes are templates for creating objects with specific methods and properties.

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    sayHello() {
        console.log("Hello!");
    }
}

let person1 = new Person("Bob", 35);
person1.sayHello();
```

## 3. Inheritance:

JavaScript supports inheritance through prototype chaining. Objects can inherit properties and methods from other objects.

```
class Animal {
    constructor(name) {
        this.name = name;
    }

    makeSound() {
        console.log("Some sound");
    }
}

class Dog extends Animal {
    constructor(name, breed) {
        super(name);
        this.breed = breed;
    }

    makeSound() {
        console.log("Woof!");
    }
}
```

```

}

let myDog = new Dog("Buddy", "Golden Retriever");
myDog.makeSound(); // Output: Woof!

```

#### 4. Encapsulation:

JavaScript doesn't have built-in support for private variables, but closures can be used to achieve encapsulation.

```

function Counter() {
    let count = 0;

    this.increment = function() {
        count++;
    };

    this.getCount = function() {
        return count;
    };
}

let counter = new Counter();
counter.increment();
console.log(counter.getCount()); // Output: 1

```

#### 5. Polymorphism:

JavaScript allows objects of different classes to be treated as objects of a common superclass through dynamic typing and duck typing.

```

class Cat {
    makeSound() {
        console.log("Meow!");
    }
}

class Duck {
    makeSound() {
        console.log("Quack!");
    }
}

function makeAnimalSound(animal) {
    animal.makeSound();
}

let myCat = new Cat();
let myDuck = new Duck();

makeAnimalSound(myCat); // Output: Meow!
makeAnimalSound(myDuck); // Output: Quack!

```

JavaScript's object-oriented features provide flexibility and allow developers to create complex applications following the principles of OOP.

Possible types of function in JavaScript

Possible types of function in JavaScript

## Possible types of functions in JS

Lambda or arrow  
functions

First class function

First order function

High order function

Unary function

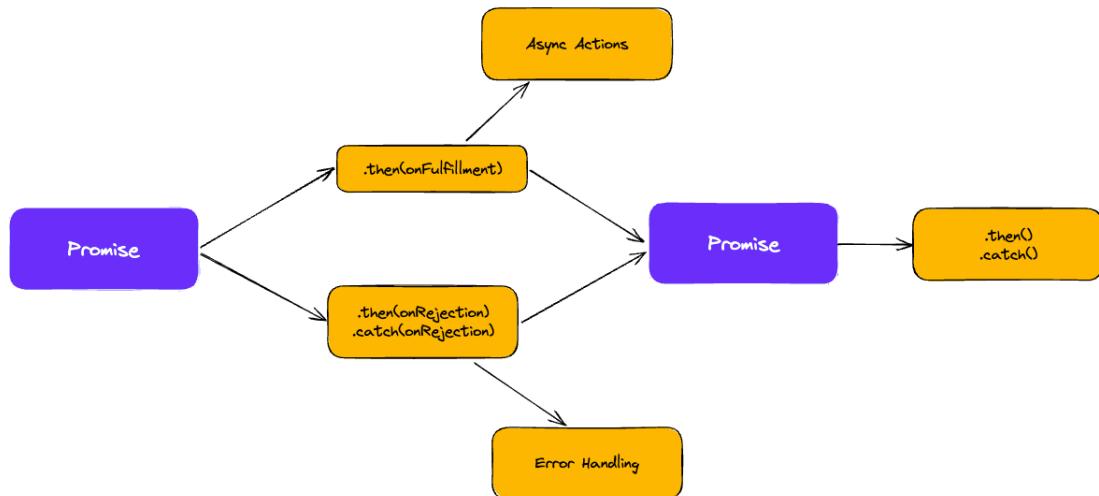
Getter Functions

Currying function

## Promise action flow

## Promise action flow

### The action flow of a promise



## What is event bubbling?

## What is event bubbling?

Event bubbling is a concept in JavaScript and many other programming languages where events triggered on an element will "bubble up" and trigger parent elements' event handlers as well. When an event occurs on a DOM (Document Object Model) element, it first runs the event's handler on that element, then on its parent element, and so on, propagating up through its ancestors in the DOM tree.

Consider the following HTML structure:

```
<div id="parent">
  <button id="child">Click me!</button>
</div>
```

If you have an event listener attached to both the parent and the child element like this:

```
const parentElement = document.getElementById('parent');
const childElement = document.getElementById('child');

parentElement.addEventListener('click', function() {
  console.log('Parent element clicked');
});

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

```
    console.log('Child element clicked');
    event.stopPropagation(); // Stops the event from bubbling up
});
```

If you click the button with the id "child," you might expect only "Child element clicked" to be logged. However, due to event bubbling, both messages will be logged because the click event bubbles up from the child element to its parent.

In the example above, `event.stopPropagation()` is used to stop the event from further propagation, preventing it from reaching the parent element. Without this line, the event would continue to bubble up to the parent element, triggering its event handler as well. This can be useful in certain cases, but it's important to understand event bubbling to control how events propagate through the DOM tree.

### What is the difference between Call and Apply?

## What is the difference between Call and Apply?

`call()` and `apply()` are two methods in JavaScript that allow you to invoke a function with a specified `this` value and arguments. They are similar in purpose but differ in how they accept arguments.

### `call()`

The `call()` method calls a function with a given `this` value and individual arguments passed one by one.

#### Syntax:

```
functionName.call(thisValue, arg1, arg2, ...);
```

- `functionName`: The function to be called.
- `thisValue`: The value to be passed as `this` to the function.
- `arg1, arg2, ...`: The arguments to be passed to the function individually.

#### Example:

```
function greet(message) {
  console.log(`#${message}, ${this.name}!`);
}

const person = {
  name: 'John'
};

greet.call(person, 'Hello');
// Output: Hello, John!
```

In this example, `call()` is used to invoke the `greet` function with `person` object as the `this` value and 'Hello' as the argument.

## apply()

The `apply()` method is similar to `call()`, but it accepts arguments as an array.

### Syntax:

```
functionName.apply(thisValue, [arg1, arg2, ...]);
```

- `functionName`: The function to be called.
- `thisValue`: The value to be passed as `this` to the function.
- `[arg1, arg2, ...]`: An array or array-like object containing the arguments to be passed to the function.

### Example:

```
function introduce(greeting, age) {
  console.log(`#${greeting}, I am ${this.name} and I am ${age} years old.`);
}

const person = {
  name: 'Alice'
};

introduce.apply(person, ['Hi', 30]);
// Output: Hi, I am Alice and I am 30 years old.
```

In this example, `apply()` is used to invoke the `introduce` function with `person` object as the `this` value and `['Hi', 30]` as the arguments passed in an array.

### Differences:

#### 1. Argument Format:

- `call()` accepts arguments individually.
- `apply()` accepts arguments as an array.

#### 2. Use Cases:

- Use `call()` when you know the exact number of arguments the function expects and you want to pass them individually.
- Use `apply()` when the number of arguments is not known in advance, or when the arguments are already in an array or array-like object.

In modern JavaScript, you can often use the more versatile spread operator (`...`) to achieve the same result as `apply()` while providing a cleaner syntax, especially when working with arrays. However, `apply()` can still be useful in certain situations, such as when dealing with variadic functions or functions that accept a variable number of arguments.

## What is the NaN

### What is the NaN

In JavaScript, `NaN` stands for "Not a Number." It is a special value representing an unrepresentable or undefined value resulting from operations that cannot produce a valid numeric result. For example, dividing zero by zero or trying to parse a string that does not represent a valid number results in `NaN`.

The concept of `NaN` in JavaScript can be traced back to the early versions of the language. JavaScript was created by Brendan Eich in 1995 while he was working at Netscape Communications Corporation. `NaN` was introduced as a way to handle undefined or unrepresentable values in mathematical operations.

When a mathematical operation in JavaScript can't produce a meaningful numeric result, the value is set to `NaN`. For instance:

```
var result = 0 / 0; // result is NaN because dividing zero by zero is undefined
```

It's important to note that `NaN` is not equal to itself according to JavaScript's specifications. This means that you cannot directly check if a value is `NaN` by using the equality operator (`==` or `===`). Instead, you should use the `isNaN()` function to determine whether a value is `NaN`:

```
isNaN(NaN); // true
isNaN(10); // false
isNaN("Hello"); // true, because trying to convert "Hello" to a number results in NaN
```

The `isNaN()` function has been part of JavaScript since its early versions and provides a way to reliably check for `NaN` values.

In summary, `NaN` has been a fundamental concept in JavaScript since its inception, providing a way to handle undefined or unrepresentable values in numeric operations. The introduction of `NaN` has helped improve the reliability and consistency of numerical computations in JavaScript.

## What is the difference between `var`, `const`, and `const`?

### What is the difference between `var`, `const`, and `const`?

In JavaScript, `var`, `let`, and `const` are used to declare variables, but they have different scoping rules and behaviors. Here's a breakdown of their differences:

#### 1. `var`:

- `var` declarations are globally scoped or function scoped.
- They are hoisted to the top of their scope and can be accessed before the declaration.
- `var` variables can be reassigned and updated.

Example:

```
function example() {
  if (true) {
    var x = 10;
  }
  console.log(x); // Outputs 10, because var is function scoped
}
```

## 2. `let`:

- `let` declarations are block-scoped (limited to the block, statement, or expression where it is defined).
- They are hoisted to the top of their scope but are not initialized until the interpreter reaches the declaration.
- `let` variables can be reassigned, but they cannot be redeclared in the same scope.

Example:

```
function example() {  
    if (true) {  
        let x = 10;  
        console.log(x); // Outputs 10, because let is block scoped  
    }  
    console.log(x); // Error: x is not defined outside the block  
}
```

## 3. `const`:

- `const` declarations are also block-scoped.
- They must be initialized during declaration and cannot be reassigned to a different value after initialization.
- Like `let`, `const` declarations are hoisted to the top of their scope but are not initialized until the interpreter reaches the declaration.

Example:

```
function example() {  
    const x = 10;  
    console.log(x); // Outputs 10, const variables cannot be reassigned  
    x = 20; // Error: Assignment to constant variable  
}
```

In modern JavaScript, it is generally recommended to use `const` by default, and only use `let` when you know the variable's value will change. Avoid using `var` unless you have a specific reason to use it, as its behavior can sometimes lead to unexpected results due to its function scope or global scope. Using `let` and `const` helps in writing more predictable and maintainable code.

## Weird and unexpected behavior of JavaScript

# Weird and unexpected behavior of JavaScript

JavaScript is a versatile and powerful programming language, but it has its fair share of quirks and behaviors that can be considered weird or unexpected. Here are more examples:

1. **Type Coercion:** JavaScript's loose typing system can lead to unexpected results when different types are used together:

```
console.log(1 + "1"); // "11"  
console.log(true + 1); // 2
```

2. **NaN**: The special value `NaN` can be perplexing:

```
console.log(typeof NaN); // "number"
console.log(NaN === NaN); // false
console.log(1 + NaN); // NaN
```

3. **Truthy and Falsy Values**: JavaScript's concept of truthy and falsy values can lead to surprises:

```
if ("false") {
  console.log("This will be executed."); // This will be executed.
}

if (0) {
  console.log("This will not be executed.");
}

console.log(Boolean ""); // false
console.log(Boolean 0); // false
```

4. **Hoisting**: JavaScript hoists variable and function declarations, which can lead to seemingly unusual behavior:

```
console.log(x); // undefined
var x = 5;
```

5. **Closures**: Closures can cause inner functions to access outer function variables even after the outer function has completed:

```
function outer() {
  var x = 10;

  function inner() {
    console.log(x);
  }

  return inner;
}

var closureFunc = outer();
closureFunc(); // Outputs 10
```

6. **NaN Equality**: `NaN` does not equal itself, making `NaN` comparisons tricky:

```
console.log(NaN == NaN); // false
console.log(isNaN(NaN)); // true
```

7. **Automatic Semicolon Insertion (ASI)**: JavaScript inserts semicolons in certain situations, affecting code behavior:

```
function example() {
  return;
}
```

```

        value: 42;
    }
}

console.log(example()); // undefined

```

8. **Variable Declarations Without var/let/const:** Missing variable declarations can lead to global scope pollution:

```

function noVarDeclaration() {
    x = 20; // This becomes a global variable!
}

noVarDeclaration();
console.log(x); // 20

```

These examples highlight some of the idiosyncrasies of JavaScript. Understanding these quirks is crucial for writing reliable and predictable JavaScript code.

- Go back

## patterns

### Event Bus

## Event Bus

```

interface IEventBus {
    channels: any,
    subscribe: (arg1: string, listener: (...args: any[]) => void) => void,
    publish: (arg1: string, arg2: any) => void
}

const EventBus: IEventBus = {
    channels: {},
    subscribe (channelName: string, listener: (...args: any[]) => void) {
        if (!this.channels[channelName]) {
            this.channels[channelName] = []
        }
        this.channels[channelName].push(listener);
    },
    publish (channelName: string, data: any) {
        const channel = this.channels[channelName];
        if (!channel || !channel.length) {
            return
        }
        channel.forEach((listener: (...args: any[]) => void) => listener(data))
    }
}

```

## Observer

### Observer

```
export class Observer {
    private observers: any[] = [];

    public subscribe(fn: any) {
        this.observers.push(fn);
    }

    public unsubscribe(fn: any) {
        this.observers = this.observers.filter(subscriber => subscriber !== fn);
    }

    public next(data: any) {
        this.observers.forEach(subscriber => subscriber(data));
    }
}
```

## Singleton

### Singleton

```
const singleton = (function() {
    let instance: any;

    class User {
        public name = '';
        public age = 0;

        constructor(name: string, age: number) {
            this.name = name;
            this.age = age;
        }
    }
    return {
        getInstance: function(name?: string, age?: number) {
            if (!instance) {
                const newName = name || "";
                const newAge = age || 0;
                instance = new User(newName, newAge);
            }
            return instance;
        }
    }
})();
```

## questions

### How to reverse a number?

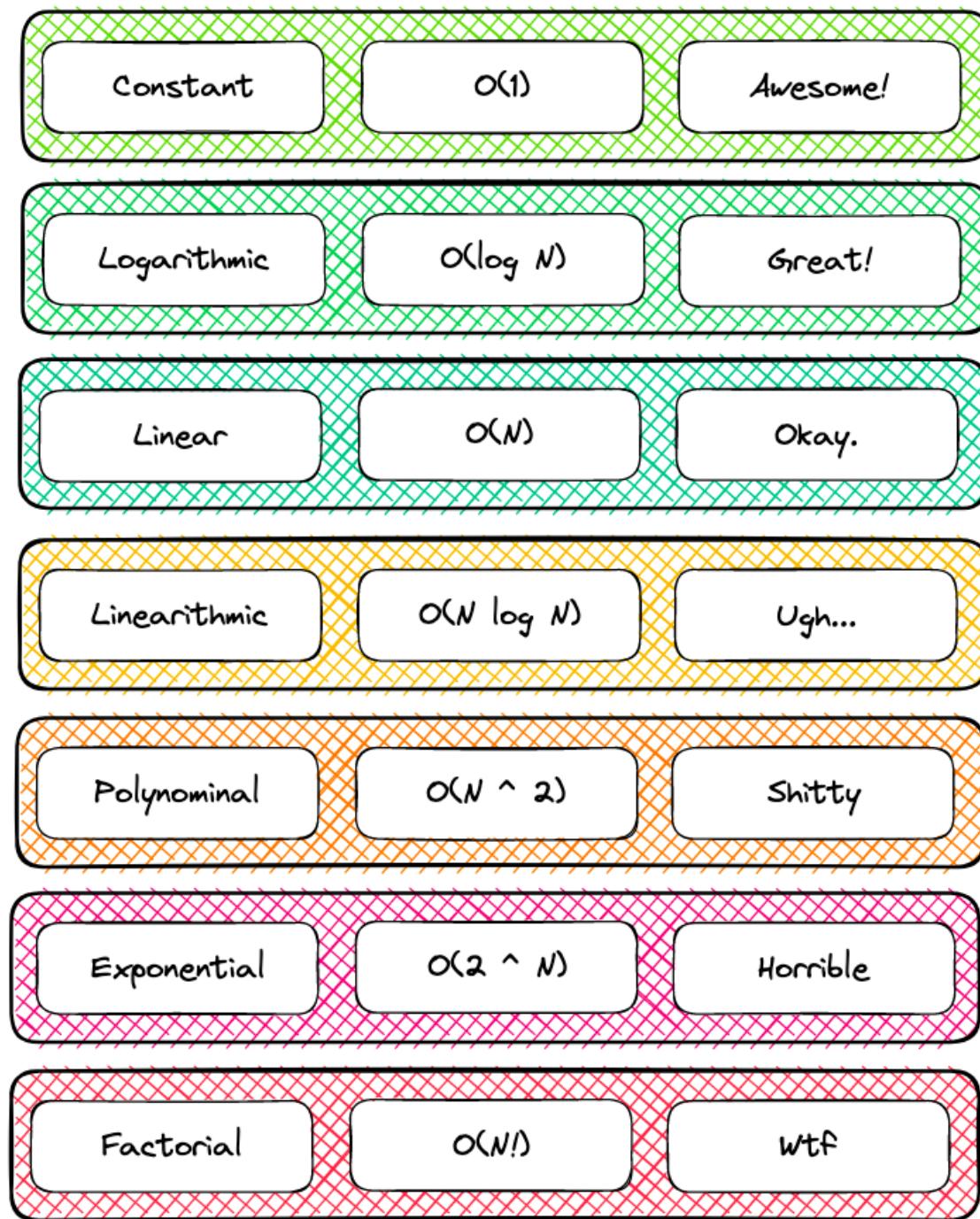
### How to reverse a number?

To invert a number in Java, you need to divide the number by 10 in the loop until it equals 0. And in the body of the loop find the remainder of the division by 10 and add to the result from the previous step, increased by a factor of 10.

```
int number = 12132;
int reverse = 0;
while(number > 0) {
    reverse = reverse * 10 + number % 10;
    number /= 10;
}
```

Big O

Big O



$N = 5$

$N = 10$

$N = 20$

$N = 30$

$O(1)$

1

1

1

1

$O(\log N)$

2.3219 ...

3.3219 ...

4.3219 ...  
70

4.9068 ...

$O(N)$

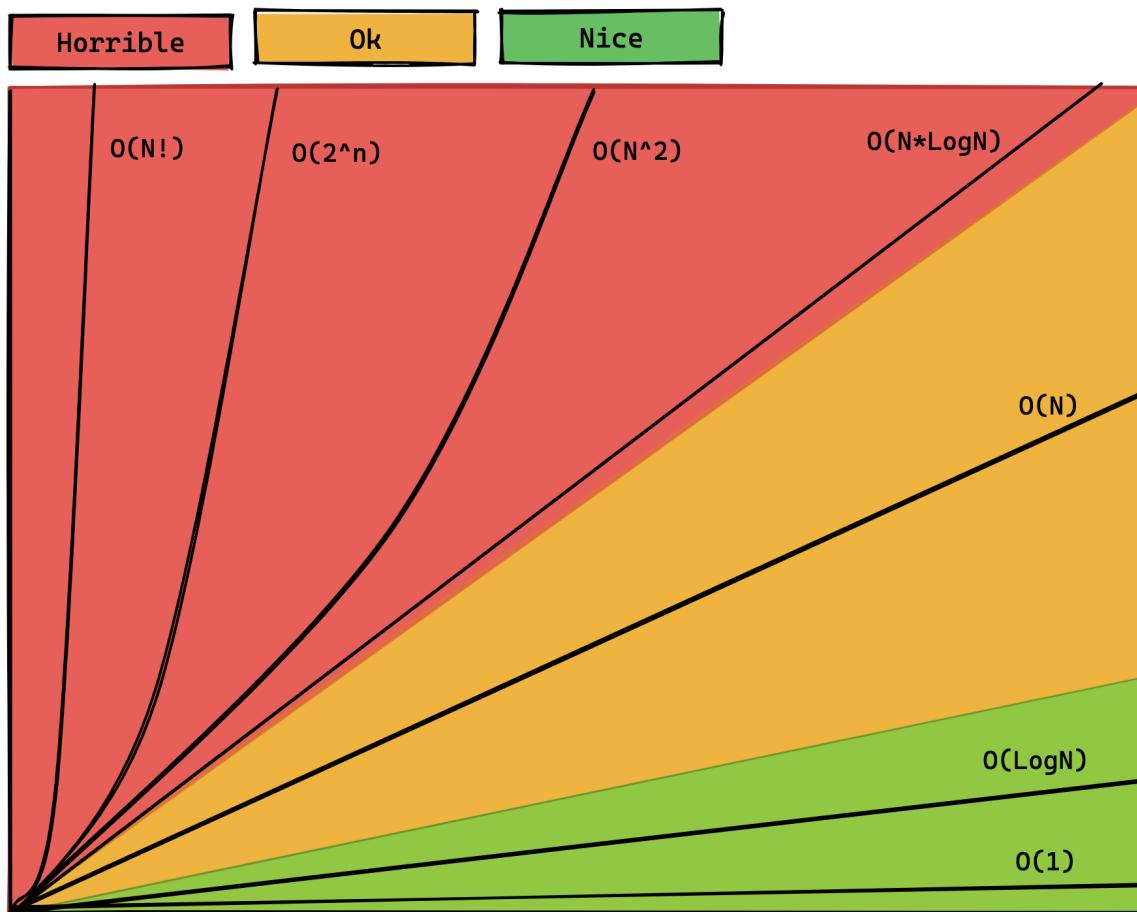
5

10

20

30

## Time complexity Graph



\*\* Source: <https://github.com/jamiebuilds/itsy-bitsy-data-structures/blob/master/itsy-bitsy-data-structures.js>\*\*

## OOP Principles

### OOP Principles

#### Abstraction

Abstract means a concept or an Idea which is not associated with any particular instance. Using abstract class/Interface we express the intent of the class rather than the actual implementation. In a way, one class should not know the inner details of another in order to use it, just knowing the interfaces should be good enough.

## **Inheritance**

Inheritance expresses “is-a” and/or “has-a” relationship between two objects. Using Inheritance, In derived classes we can reuse the code of existing super classes. In Java, concept of “is-a” is based on class inheritance (using extends) or interface implementation (using implements).

## **Encapsulation**

Encapsulation is the mechanism of hiding of data implementation by restricting access to public methods. Instance variables are kept private and accessor methods are made public to achieve this.

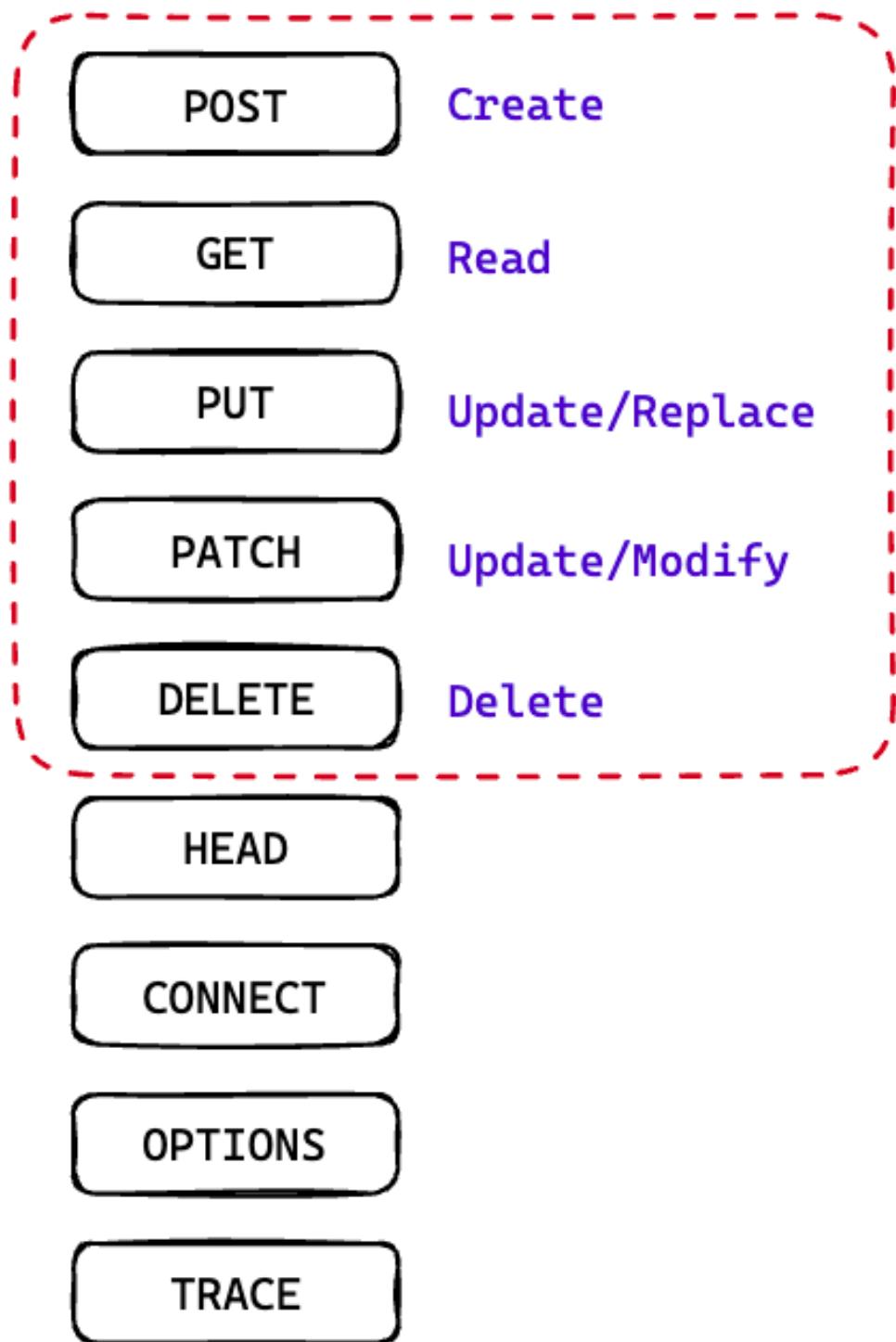
## **Polymorphism**

It means one name many forms. It is further of two types — static and dynamic. Static polymorphism is achieved using method overloading and dynamic polymorphism using method overriding. It is closely related to inheritance. We can write a code that works on the superclass, and it will work with any subclass type as well.

RESTful API methods

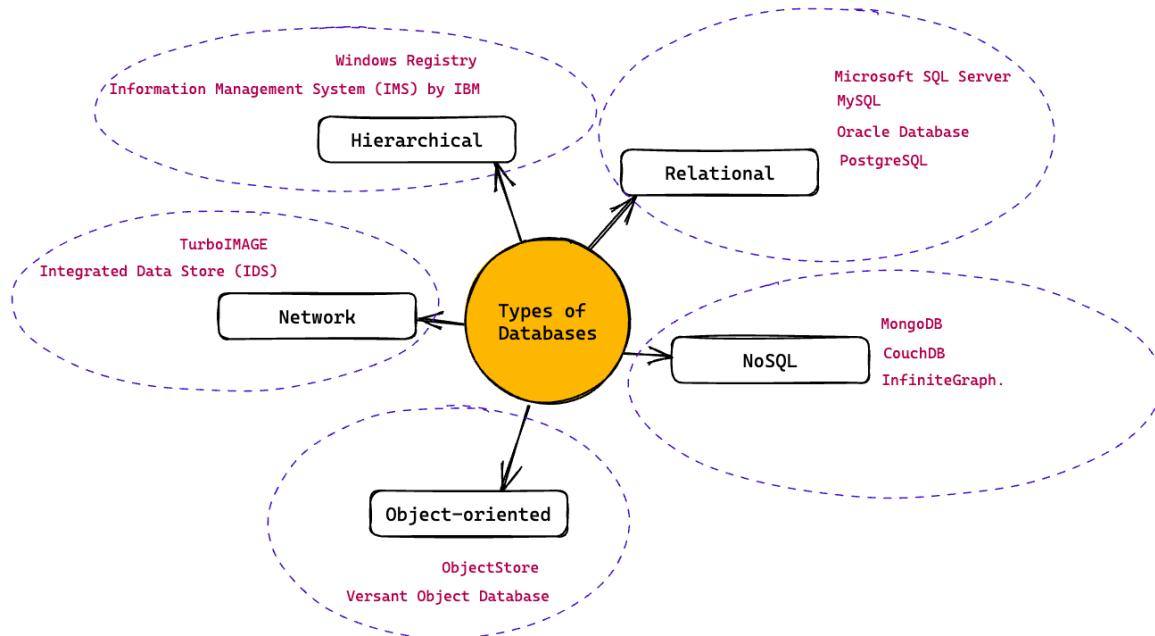
RESTful API methods

## RESTful API methods



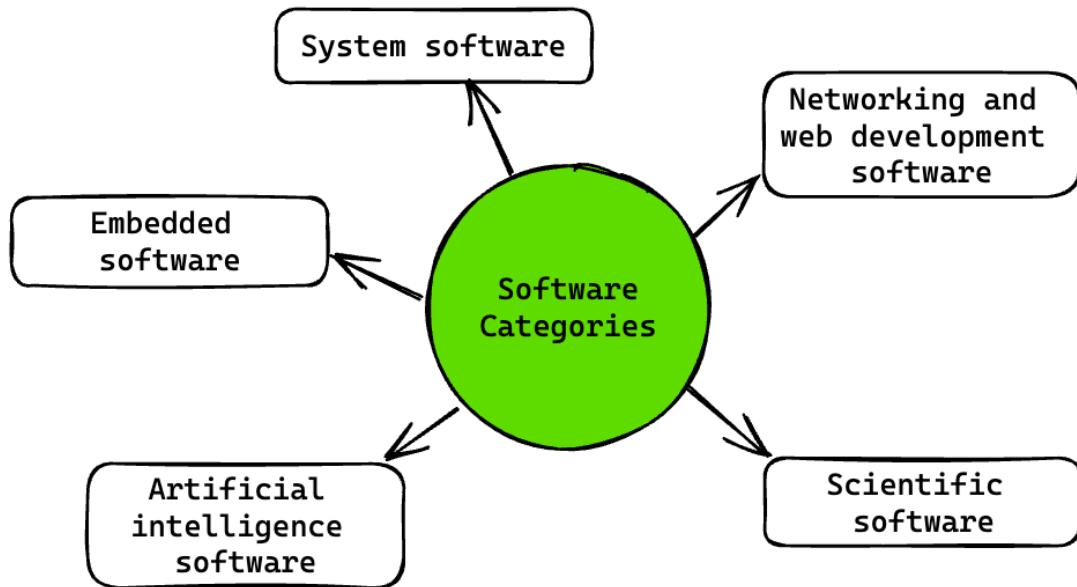
## Types of Databases

### Types of Databases



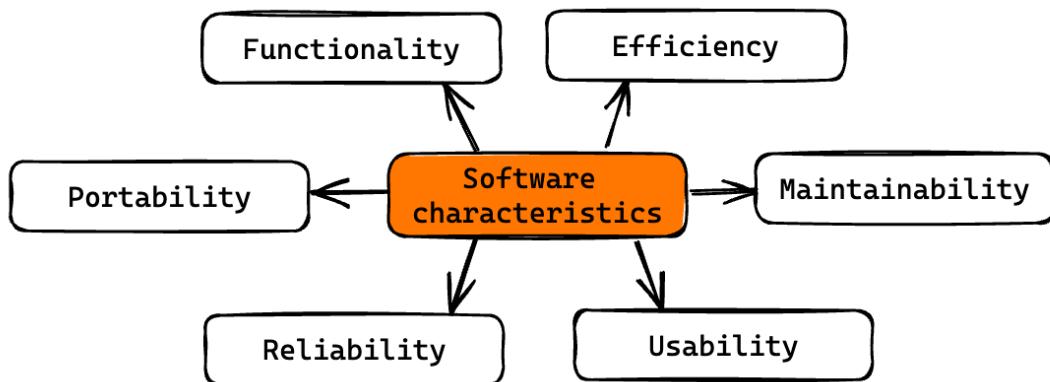
Software categories

## Software categories



Software characteristics

## Software characteristics



Software development life cycle

Software development life cycle

# Software Development Life Cycle

## SDLC

- 1 Requirements Gathering
- 2 Software Design
- 3 Software Development
- 4 Testing and Integration
- 5 Deployment
- 6 Operation and Maintenance

## Design patterns

# Design patterns

### Creational Patterns:

1. **Factory Method:** Creates objects without specifying the exact class by delegating the instantiation to subclasses. Provides a way to create objects based on certain conditions or parameters.
2. **Abstract Factory:** Provides an interface for creating families of related or dependent objects without specifying their concrete classes. Supports creating objects with different implementations but having a common theme.
3. **Builder:** Separates the construction of a complex object from its representation, allowing the same construction process to create different representations. Useful when there are multiple ways to construct an object.
4. **Prototype:** Clones existing objects to create new ones, avoiding the need for subclassing. Allows objects to be copied without making the code dependent on their classes.
5. **Singleton:** Ensures a class has only one instance and provides a global point of access to it. Useful when you need to have a single shared instance across the entire application.

**Structural Patterns:** 6. **Adapter:** Converts the interface of one class into another interface that clients expect, making incompatible classes work together.

7. **Bridge:** Decouples an abstraction from its implementation, allowing them to vary independently. Useful when you want to extend a class in two independent dimensions.
8. **Composite:** Treats individual objects and compositions of objects uniformly, allowing you to create complex tree-like structures.
9. **Decorator:** Dynamically adds behavior or responsibilities to objects without altering their code directly. Offers a flexible alternative to subclassing.
10. **Facade:** Provides a simplified interface to a complex subsystem, making it easier to use and understand.
11. **Flyweight:** Shares data between multiple objects to reduce memory usage when many similar objects exist.
12. **Proxy:** Acts as a placeholder for another object, controlling access to it, and providing additional functionality when required.

**Behavioral Patterns:** 13. **Chain of Responsibility:** Allows multiple objects to handle a request without specifying the receiver explicitly. The request is passed along the chain until it's handled.

14. **Command:** Encapsulates a request as an object, allowing parameterization of clients with different requests, queuing of requests, or logging of the requests.
15. **Iterator:** Provides a way to access elements of a collection without exposing its underlying representation.
16. **Mediator:** Defines an object that centralizes communication between multiple objects, reducing their direct interactions.

17. **Memento:** Captures and externalizes an object's internal state, allowing the object to be restored to that state later.
18. **Observer:** Allows an object to publish changes to its state, notifying dependent objects to update automatically.
19. **State:** Allows an object to change its behavior when its internal state changes.
20. **Strategy:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Enables selecting an algorithm at runtime.
21. **Template Method:** Defines the skeleton of an algorithm in a method, allowing subclasses to provide specific implementations of some steps.
22. **Visitor:** Separates algorithms from the objects on which they operate, enabling adding new operations without modifying the objects' classes.

## SOLID

## SOLID

The SOLID principles are a set of five design principles that aim to guide software developers in creating maintainable, scalable, and flexible code. Each principle focuses on a specific aspect of software design, and when applied together, they promote better software quality and reduce the risk of technical debt. Let's go through each SOLID principle:

### 1. Single Responsibility Principle (SRP):

- A class should have only one reason to change, i.e., it should have only one responsibility.
- This principle promotes the idea that a class should be focused on doing one thing well, making it easier to understand, maintain, and extend.
- By keeping classes small and focused, changes to one aspect of the system are less likely to affect other unrelated aspects.

### 2. Open/Closed Principle (OCP):

- Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.
- This means that you should be able to extend the behavior of a class or module without modifying its existing code.
- Achieving this often involves using abstractions (interfaces, abstract classes) and polymorphism, allowing you to add new functionality through inheritance or composition.

### 3. Liskov Substitution Principle (LSP):

- Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.
- In other words, a subclass should adhere to the behavior expected by the superclass and not violate the contract established by the superclass.
- Violating this principle can lead to unexpected behaviors when substituting objects and can undermine the correctness and reliability of the software.

### 4. Interface Segregation Principle (ISP):

- Clients should not be forced to depend on interfaces they do not use.

- This principle advocates for small, specific interfaces rather than large, general ones.
- By having fine-grained interfaces, clients can depend only on the functionality they require, reducing unnecessary dependencies and potential coupling.

### 5. Dependency Inversion Principle (DIP):

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.
- This principle encourages the use of interfaces or abstract classes to define contracts, allowing high-level modules to be decoupled from low-level implementation details.
- By relying on abstractions, it becomes easier to change implementations without affecting higher-level modules.

By adhering to the SOLID principles, developers can create code that is more maintainable, extensible, and easier to test. These principles contribute to the development of robust and adaptable software systems, which can evolve over time with fewer risks of introducing bugs or unintended side effects when making changes.

## GRASP

### GRASP

Certainly! GRASP stands for General Responsibility Assignment Software Patterns, a set of object-oriented design principles used to guide the assignment of responsibilities to classes and objects in a software system. These principles help in creating a flexible and maintainable design. Here are the GRASP principles:

#### 1. Creator:

- This principle suggests that a class should be responsible for creating instances of other classes if there is a strong relationship between them. For example, a factory class can be responsible for creating and initializing objects of a specific class.

#### 2. Information Expert:

- This principle states that a responsibility should be assigned to the class with the most information required to fulfill that responsibility. In other words, a class that has the necessary data and behavior to perform a task should be given the responsibility to do so.

#### 3. Low Coupling:

- This principle advocates for reducing the dependencies between classes by keeping them loosely coupled. Low coupling improves the maintainability and flexibility of the system as changes in one class are less likely to impact other classes.

#### 4. High Cohesion:

- This principle suggests that a class should have a single, well-defined purpose or responsibility. High cohesion within a class ensures that it is focused and easy to understand.

#### 5. Controller:

- The Controller pattern deals with managing the flow of information and requests between objects or components. It is responsible for interpreting user input and deciding which objects should handle the request.

6. Polymorphism:

- The Polymorphism principle involves the use of interfaces and abstract classes to allow multiple implementations for a single behavior or action. This increases the flexibility and extensibility of the system.

7. Indirection:

- The Indirection principle promotes the use of intermediaries to reduce direct dependencies between classes. It can be achieved through the use of interfaces or abstract classes, which enable more flexible and interchangeable components.

8. Protected Variations:

- This principle aims to protect a system from the impact of future changes by encapsulating the variations or volatile components within the system. It promotes using abstraction to isolate these variations.

9. Pure Fabrication:

- When no natural class fits the responsibility, a new class can be created solely to handle that specific responsibility. This class is considered a "pure fabrication" and helps in achieving a more maintainable design.

GRASP principles provide a set of guidelines for assigning responsibilities to classes, ensuring a well-structured and adaptable object-oriented design. Applying these principles can lead to more maintainable, robust, and flexible software systems.

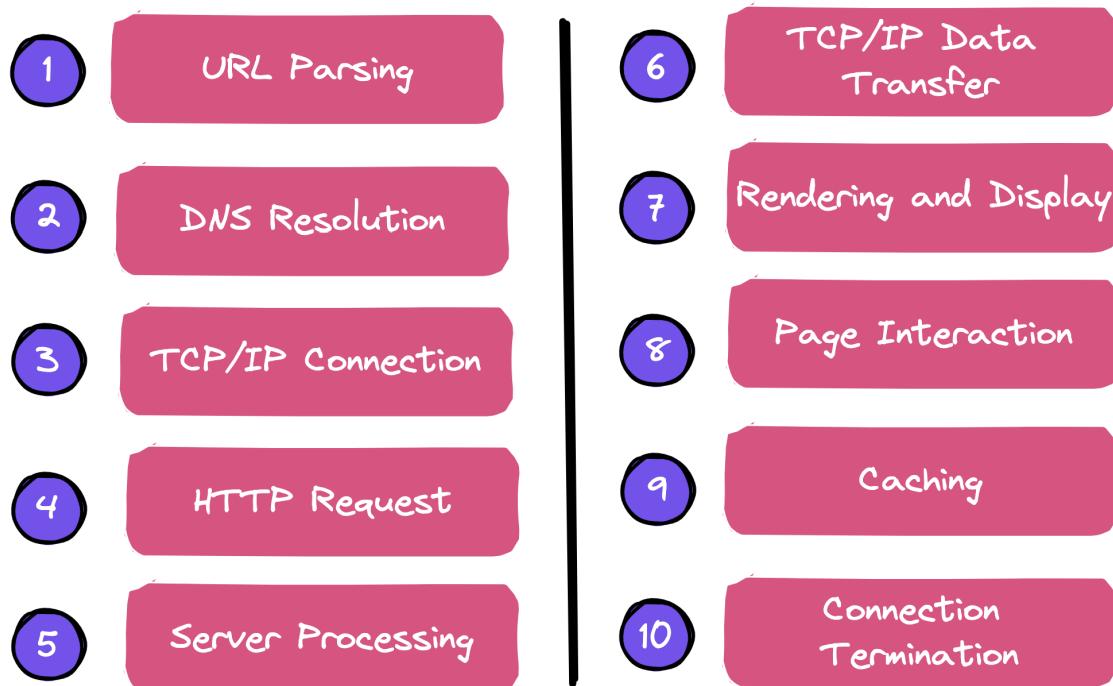
### **Browser URL Search**

## **Browser URL Search**

### **Question**

A person settled in front of their laptop, launching a browser before typing a URL into the search bar. What unfolds within the browser at the network level following this action?

A person settled in front of their laptop, launching a browser before typing a URL into the search bar. What unfolds within the browser at the network level following this action?



## Answer

- URL Parsing:** When a user enters a URL into the browser's search bar, the browser parses the URL to extract components like the protocol (e.g., "https"), the domain name (e.g., "www.example.com"), and the path (e.g., "/page"). The browser also checks the URL's validity.
- DNS Resolution:** To find the IP address associated with the domain name, the browser queries a DNS (Domain Name System) server. This process involves multiple steps, potentially starting with the local cache and progressing to the ISP's DNS server and authoritative DNS servers for the domain.
- TCP/IP Connection:** The browser initiates a secure TCP connection using TLS (Transport Layer Security) to the IP address of the web server hosting the website. A process known as the TLS handshake occurs, involving encryption negotiation, key exchange, and verification to establish a secure connection.
- HTTPs Request:** With the secure connection established, the browser sends an HTTPS (HTTP Secure) request to the web server. The request includes the HTTP method (e.g., GET, POST), headers (including user agent information), and the requested path.
- Server Processing:** The web server receives the HTTPS request, processes it, and generates an appropriate HTTPS response. This response includes a status code (e.g., 200 OK), headers (e.g., content type), and the requested content (HTML, images, etc.).
- TCP/IP Data Transfer:** The server sends the HTTPS response back to the browser over the established encrypted TCP connection.

7. **Rendering and Display:** The browser receives the response, interprets the HTML content, and renders the page. Additional resources referenced in the HTML (such as images, stylesheets, and JavaScript files) are fetched over the secure connection and processed similarly.
8. **Page Interaction:** Users interact with the rendered page, which may involve actions like clicking links or submitting forms. These actions trigger additional HTTPS requests for resources and data.
9. **Caching:** The browser may cache certain resources for improved performance during subsequent visits. Cached resources are reused to load pages faster.
10. **Connection Termination:** When the user closes the browser tab or navigates away from the page, the browser may gracefully terminate the encrypted TCP connection, involving a process to ensure proper closure.

It's important to note that the use of HTTPS adds an extra layer of security through encryption, helping to protect the confidentiality and integrity of the data exchanged between the browser and the web server.

## Rest API

# Rest API

### 1. Introduction to REST API:

- Begin by defining REST API. REST stands for Representational State Transfer, and it is an architectural style for designing networked applications.

### 2. Key Principles of REST:

- Mention the key principles of REST:
  - **Stateless:** Each request from a client to the server must contain all the information needed to understand and process the request.
  - **Client-Server:** The client and server are separate entities that communicate over a network.
  - **Uniform Interface:** A consistent and uniform way of interacting with resources (typically using HTTP methods).
  - **Resource-Based:** Everything in a REST API is treated as a resource, and each resource is identified by a unique URI.
  - **Representation:** Resources can have multiple representations (e.g., JSON, XML) to support different clients.

### 3. HTTP Methods:

- Explain the common HTTP methods used in REST:
  - **GET:** Retrieve data from the server.
  - **POST:** Create a new resource on the server.
  - **PUT:** Update an existing resource on the server.
  - **DELETE:** Remove a resource from the server.
  - **PATCH:** Partially update a resource.
  - **OPTIONS:** Retrieve information about the communication options available for a resource.

### 4. URI Structure:

- Discuss how URIs are structured in REST APIs.
- Explain the concept of endpoints and how they map to resources.

## **5. Status Codes:**

- Mention the significance of HTTP status codes (e.g., 200 for success, 404 for not found, 500 for server error) in REST APIs.

## **6. Data Formats:**

- Explain that data in REST APIs is typically exchanged in standardized formats like JSON or XML.
- Discuss how clients can request specific formats using the "Accept" header.

## **7. Authentication and Authorization:**

- Briefly touch on the importance of authentication and authorization mechanisms in securing REST APIs.

## **8. Statelessness and Scalability:**

- Emphasize how the statelessness of REST allows for easy scalability because each request contains all necessary information.

## **9. Advantages of REST API:**

- Highlight the benefits of using REST, such as simplicity, scalability, and ease of caching.

**10. Use Cases and Examples:** - Provide examples of real-world applications or services that use REST APIs (e.g., Twitter API for retrieving tweets).

**11. Drawbacks and Considerations:** - Mention any potential drawbacks or challenges, such as over-fetching data or lack of support for real-time updates.

**12. Conclusion:** - Summarize by reiterating the key points about REST APIs and their significance in modern web development.

Remember to tailor your response to the specific details of the interview question and your own experiences or knowledge of REST APIs. Providing concrete examples from your previous work or projects can also enhance your answer and demonstrate your practical understanding of REST.

## **Websockets**

# **WebSocket**

WebSocket is a communication protocol that provides full-duplex, bidirectional communication channels over a single TCP connection. It is designed to enable real-time, interactive communication between a client (typically a web browser) and a server. WebSocket is often used in web applications and other scenarios where low-latency, efficient communication is required.

Here's how WebSocket works:

1. **Handshake:** WebSocket communication begins with an initial handshake. This handshake is an HTTP request from the client to the server, usually in the form of a standard HTTP GET request. However, it includes an additional **Upgrade** header field with a value of "websocket" to indicate that the client wants to establish a WebSocket connection. The server must also support WebSocket and, if it does, it responds with a status code of 101 (Switching Protocols) to confirm the upgrade.
2. **WebSocket Connection Established:** Once the handshake is successful, the connection is upgraded from HTTP to WebSocket. From this point onward, the communication occurs over a single, persistent TCP connection, eliminating the need for repeated HTTP requests and responses.
3. **Data Exchange:** With the WebSocket connection established, both the client and server can send data to each other at any time, without the need for a request-response pattern. Data is sent as frames, which can be either text or binary. Each frame includes an opcode to indicate its type (e.g., text, binary, close, ping, pong), the payload data, and other control bits for framing.
4. **Full-Duplex Communication:** WebSocket supports full-duplex communication, meaning both the client and server can send messages independently without waiting for a response. This real-time, bidirectional communication makes it suitable for interactive applications, such as chat applications, online gaming, and live updates.
5. **Keep-Alive and Ping/Pong:** WebSocket includes built-in mechanisms for keeping the connection alive and detecting if it's still active. Periodically, one party may send a ping frame, and the other must respond with a pong frame. If one party fails to receive a pong response within a specified timeout, it can assume the connection is lost and take appropriate action.
6. **Close Connection:** Either party can initiate the closing of the WebSocket connection by sending a close frame. This allows for a graceful termination of the connection, and both sides can perform cleanup operations before disconnecting.

WebSocket offers several advantages over traditional HTTP communication for real-time applications, as it eliminates the overhead of repeated HTTP requests and reduces latency. It is widely used in web development, online gaming, financial trading platforms, and any application that requires low-latency, bi-directional communication between a client and server. Additionally, WebSocket libraries and APIs are available for various programming languages, making it relatively easy to implement in web and mobile applications.

## Graphql

# Graphql

GraphQL is a query language for APIs (Application Programming Interfaces) and a runtime for executing those queries by using a type system you define for your data. It was developed by Facebook in 2012 and open-sourced in 2015. GraphQL is designed to enable clients to request exactly the data they need, and nothing more, making it a powerful alternative to REST (Representational State Transfer) APIs for building flexible and efficient APIs. Here are the main principles and concepts of GraphQL:

1. **Hierarchical Structure:** GraphQL queries are hierarchical in nature, mirroring the shape of the response data. This allows clients to specify exactly what data they want, including nested fields and related data, in a single query.
2. **Strongly Typed:** GraphQL APIs are defined by a schema that explicitly defines the types of data that can be queried. These types can be scalars (integers, strings, etc.) or complex types (objects with fields). This schema provides a clear contract between the client and server.

3. **Single Endpoint:** Unlike REST, which often requires multiple endpoints for different resources, GraphQL typically has a single endpoint for all queries and mutations. This simplifies API requests and reduces over-fetching of data.
4. **Client-Defined Queries:** Clients are responsible for specifying the shape and structure of the data they need. This eliminates the problem of over-fetching or under-fetching data that can occur with REST APIs, where the server determines the response structure.
5. **No Over-fetching or Under-fetching:** With GraphQL, clients can request only the data they need, reducing the amount of data transferred over the network. This helps optimize performance, especially in mobile applications with limited bandwidth.
6. **Real-time Data:** GraphQL supports real-time data updates through subscriptions. Clients can subscribe to specific events or data changes and receive updates when those changes occur on the server.
7. **Batching:** Clients can send multiple queries in a single request to reduce the number of round-trips to the server. This is especially useful for optimizing network performance.
8. **Introspection:** GraphQL APIs are self-documenting. Clients can query the schema itself to discover the available types, queries, mutations, and their descriptions. This makes it easier to explore and interact with the API.
9. **Mutations:** GraphQL allows clients to modify data on the server using mutations. Mutations are defined in the schema and can be used to create, update, or delete data.
10. **Validation and Type Checking:** GraphQL servers perform validation and type checking on incoming queries to ensure that they adhere to the schema. This helps catch errors early in the development process.
11. **Security:** GraphQL provides mechanisms for controlling access to data, including authentication and authorization. Developers can implement custom logic to secure their GraphQL APIs.

In summary, GraphQL offers a more flexible and efficient way to interact with APIs by giving clients the power to request precisely the data they need, reducing over-fetching, and providing a strongly typed schema for clarity and validation. These principles make it a popular choice for modern web and mobile application development.

## Git + Gitflow

## Git + Gitflow

### 1. Purpose of Using GIT:

Git is a distributed version control system (DVCS) designed to track changes in source code during software development. Its main purposes are:

- a. **Version Control:** Git helps developers manage and track different versions of their codebase, allowing for easy collaboration and tracking changes over time.
- b. **Collaboration:** It enables multiple developers to work on the same project simultaneously, facilitating collaboration and reducing conflicts.
- c. **History Tracking:** Git records every change made to the codebase, making it easy to trace back to specific changes, find bugs, and understand the evolution of the project.

**d. Branching and Merging:** Git allows developers to create branches for experimenting or developing new features without affecting the main codebase. Merging these branches back into the main codebase is seamless.

**e. Backup and Recovery:** Git provides redundancy and a backup mechanism for your code, reducing the risk of data loss.

## 2. How Git Works:

Git works based on the following core concepts:

**a. Repository:** A repository is where your project's files and their history are stored. It can be local (on your computer) or remote (on a server like GitHub or GitLab).

**b. Commit:** A commit is a snapshot of changes made to your code. Commits have unique identifiers and capture the state of your project at a specific moment.

**c. Branch:** A branch is a separate line of development. You can create branches to work on features or bug fixes independently. Branches help prevent conflicts when multiple people work on the same codebase.

**d. Merge:** Merging combines changes from one branch into another, typically integrating a feature branch back into the main codebase.

**e. Pull and Push:** Pulling retrieves changes from a remote repository to your local repository, while pushing sends your local changes to a remote repository.

**f. Remote:** A remote is a repository hosted on a server, like GitHub or GitLab, that allows multiple developers to collaborate on a project.

## 3. Gitflow:

Gitflow is a branching model and workflow that defines a set of rules for managing branches in a Git repository. It was popularized by Vincent Driessen and helps teams organize their development process. It consists of the following main branches:

**a. Master (Main) Branch:** This branch contains production-ready code. It should always be stable and deployable.

**b. Develop Branch:** The develop branch is used for ongoing development. It's where new features and bug fixes are integrated before they reach the master branch.

**c. Feature Branches:** Feature branches are created for new features or enhancements. Each feature gets its own branch, allowing developers to work on features independently.

**d. Release Branches:** When it's time to prepare a new release, a release branch is created from the develop branch. Final testing and bug fixes are done here before merging into the master branch.

**e. Hotfix Branches:** Hotfix branches are used for critical bug fixes in the master branch. They are created from the master branch and merged back into both master and develop branches.

Gitflow provides a structured approach to managing code changes, ensuring a stable master branch while allowing for parallel development of new features and bug fixes. It enhances collaboration and coordination within development teams.

## What is functional programming

# What is functional programming

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. In functional programming, functions are first-class citizens, meaning they can be passed as arguments to other functions, returned as values from

other functions, and assigned to variables. This approach emphasizes immutability, purity, and higher-order functions.

Here are some key concepts in functional programming:

1. **Pure Functions:** Pure functions are functions where the output value is determined only by its input values, without observable side effects. This means if you call a function with the same arguments, it will always return the same result.
2. **Immutability:** In functional programming, data is immutable, which means once it is created, it cannot be changed. Instead of modifying existing data, functional programs create new data.
3. **First-Class and Higher-Order Functions:** Functions are treated as first-class citizens, which means they can be passed as arguments to other functions, returned as values from other functions, and assigned to variables. Higher-order functions are functions that can take other functions as arguments or return them as results.
4. **Referential Transparency:** An expression is called referentially transparent if it can be replaced with its value without changing the program's behavior. This is closely related to the concept of pure functions.
5. **Recursion:** Functional programming languages often rely heavily on recursion instead of loops for repetitive tasks.
6. **Lazy Evaluation:** Lazy evaluation delays the evaluation of an expression until its value is actually needed. This can help improve performance and allows for the creation of potentially infinite data structures.

Popular functional programming languages include Haskell, Lisp, Erlang, F#, and Scala. However, many other programming languages, like JavaScript, Python, and Java, also support functional programming paradigms to varying degrees, allowing developers to use functional programming concepts alongside other programming styles.

### What is REPL and how to use it?

## What is REPL and how to use it?

REPL stands for Read-Eval-Print Loop. It is an interactive programming environment that allows you to enter commands or code snippets, evaluate them, and immediately see the results. REPs are especially common in interpreted languages like Python, JavaScript, Ruby, and also in compiled languages like Swift and TypeScript.

To use a REPL, you typically open a terminal or an online tool, enter your code, and observe the output instantly. This interactive nature makes it great for learning, testing small pieces of code, and exploring language features.

### Example in TypeScript:

In TypeScript, you can utilize the TypeScript Playground, an online REPL environment provided by the TypeScript team.

1. **Access the TypeScript Playground:** Open your web browser and navigate to the TypeScript Playground at [TypeScript Playground](#).
2. **Write TypeScript Code:** In the left pane, you can write TypeScript code directly. For example:

```

function greet(name: string): string {
    return `Hello, ${name}!`;
}

const message: string = greet("TypeScript");
console.log(message);

```

3. **Observe Compilation and Output:** As you type, the TypeScript Playground will compile your code in real-time. The compiled JavaScript code appears in the right pane. You can also see the output in the console at the bottom.
4. **Experiment and Learn:** You can experiment with different TypeScript features, explore type annotations, interfaces, classes, and more. Any errors or issues in your code will be highlighted, helping you learn and correct mistakes on the fly.

Remember that the TypeScript Playground provides a sandboxed environment to try out TypeScript without any local setup, making it a valuable resource for learning and experimenting with the language.

#### Why do you need dto files?

### Why do you need dto files?

DTO files, or Data Transfer Object files, are a common pattern in software development used for transferring data between different parts of an application or between different applications. While the concept of DTOs doesn't specifically require dedicated files, developers often create separate DTO classes or files for various reasons:

1. **Separation of Concerns:** Keeping DTOs in separate files promotes a clear separation of concerns within the codebase. By isolating data transfer logic into distinct files, developers can manage data-related tasks independently of other parts of the application, making the codebase more organized and maintainable.
2. **Reusability:** DTOs often represent common data structures used across different parts of an application or even in multiple applications. Having DTOs in separate files allows them to be easily reused in various components or services without duplicating code.
3. **Code Organization:** Placing DTOs in dedicated files enhances code readability and organization. Developers and other team members can quickly locate and understand the data structures used for data transfer without having to search through large, complex files.
4. **Testing:** Separate DTO files simplify the testing process. Unit tests and integration tests can focus specifically on the data transfer logic by importing and testing individual DTOs, making it easier to verify that the data is being transferred correctly between components.
5. **Versioning and Compatibility:** When DTOs are used in APIs or services, having them in separate files can aid in versioning. New versions of APIs can introduce updated DTOs without affecting existing implementations, ensuring backward compatibility for clients using older versions of the API.
6. **Collaboration:** In collaborative development environments, having DTOs in separate files allows multiple developers to work on different parts of the application simultaneously. Each developer can focus on a specific DTO without conflicting with others, streamlining the development process.

In summary, while DTOs themselves don't require dedicated files, creating separate DTO files offers advantages in terms of separation of concerns, reusability, code organization, testing, versioning, and collaboration. These benefits contribute to more maintainable, scalable, and efficient software development practices.

**Can you explain the difference between REST API and GraphQL?**

## **Can you explain the difference between REST API and GraphQL?**

REST API and GraphQL are both technologies used for building APIs (Application Programming Interfaces) that allow different software systems to communicate with each other. However, they have distinct differences in terms of architecture and data fetching mechanisms.

### **REST API:**

REST (Representational State Transfer) is an architectural style that uses a set of constraints when building web services. REST APIs are based on the principles of statelessness, meaning each request from a client to a server must contain all the information needed to understand and process the request. REST APIs use standard HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources, which are identified by URIs (Uniform Resource Identifiers). Data is typically exchanged in JSON or XML format.

REST APIs have a fixed structure, and clients often end up over-fetching or under-fetching data. Over-fetching means fetching more data than needed, leading to wasteful bandwidth usage, while under-fetching means not getting enough data in a single request, resulting in multiple requests and reduced efficiency.

### **GraphQL:**

GraphQL, on the other hand, is a query language for APIs and a server-side runtime for executing those queries with existing data. It provides a more efficient, flexible, and powerful alternative to the REST API. Unlike REST, where the server determines the response structure, in GraphQL, the client specifies exactly what data it needs, and the server responds with only that data. Clients can request multiple resources in a single query, avoiding issues like over-fetching and under-fetching.

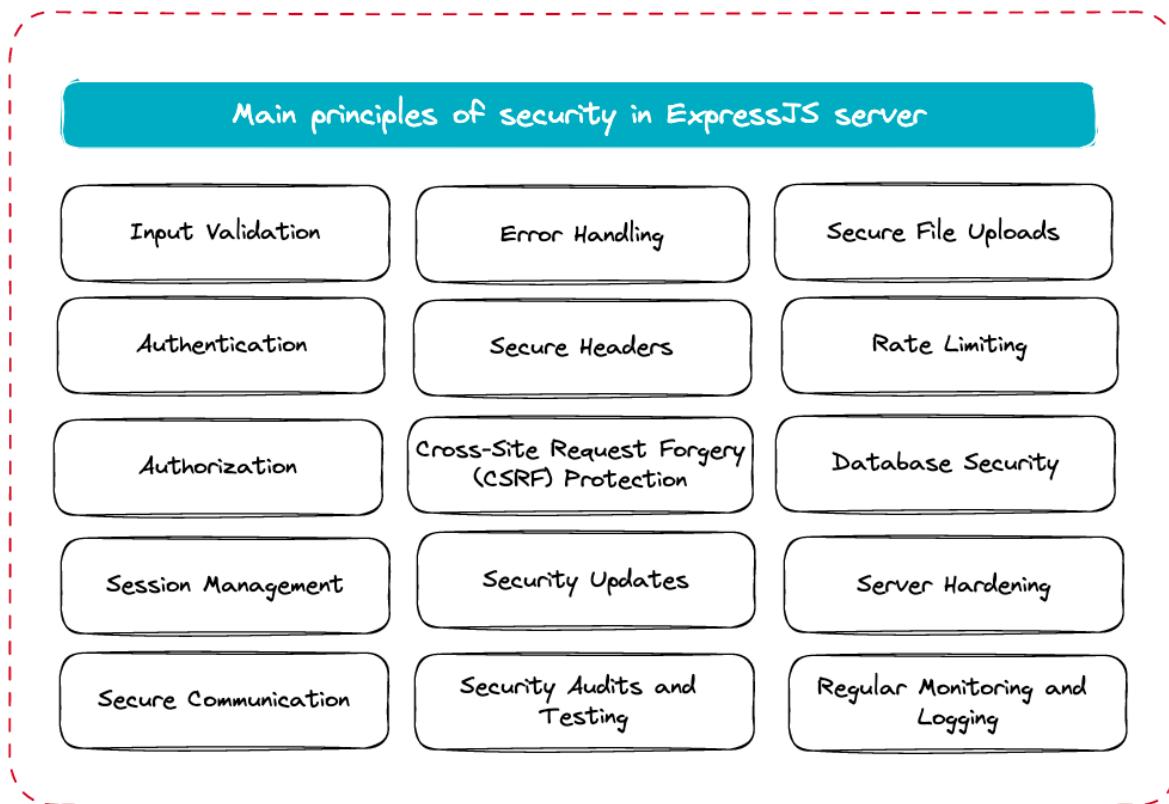
GraphQL APIs have a strong type system, allowing clients to specify the shape and structure of the response data. This enables faster iteration for frontend developers, as they can request the exact data they need without relying on changes in the backend API. GraphQL APIs are typically exposed via a single endpoint, and clients can request nested or related data in a single query, reducing the number of requests made to the server.

In summary, while REST APIs follow a fixed structure and can lead to over-fetching or under-fetching of data, GraphQL provides a more efficient and flexible approach by allowing clients to request only the specific data they need, leading to more efficient data retrieval and improved performance for applications.

## security

Main principles of security in ExpressJS server

### Main principles of security in ExpressJS server



1. **Input Validation:** Validate and sanitize all user input, including parameters, query strings, and request bodies, to prevent common vulnerabilities like SQL injection, cross-site scripting (XSS), and command injection.
2. **Authentication:** Implement proper user authentication mechanisms, such as username/password authentication, token-based authentication (JWT), or OAuth, to ensure that only authorized users can access protected resources.
3. **Authorization:** Enforce authorization checks to ensure that authenticated users have the appropriate permissions to access specific resources or perform certain actions.
4. **Session Management:** Use secure session management techniques, such as setting secure and HTTP-only cookies, implementing session expiration and regeneration, and guarding against session fixation attacks.
5. **Secure Communication:** Always use HTTPS (TLS/SSL) to encrypt communication between the server and clients, preventing eavesdropping, data tampering, and man-in-the-middle attacks.
6. **Error Handling:** Handle errors carefully to avoid leaking sensitive information in error messages and implement proper error logging to help identify and respond to potential security issues.

7. **Secure Headers:** Set secure HTTP response headers, including Content Security Policy (CSP), Strict-Transport-Security (HSTS), X-XSS-Protection, X-Content-Type-Options, and X-Frame-Options, to mitigate common web vulnerabilities.
8. **Cross-Site Request Forgery (CSRF) Protection:** Implement CSRF tokens and enforce their usage to protect against CSRF attacks, ensuring that requests made to your server originate from legitimate sources.
9. **Security Updates:** Regularly update and patch both Express.js and its dependencies to address any security vulnerabilities that may be discovered.
10. **Security Audits and Testing:** Conduct security audits and perform penetration testing to identify potential vulnerabilities and weaknesses in your server, and regularly test your application for security issues.
11. **Secure File Uploads:** Implement proper validation and handling of file uploads to prevent malicious files from being uploaded and executed on the server.
12. **Rate Limiting:** Implement rate limiting mechanisms to prevent brute force attacks, denial-of-service (DoS) attacks, and other forms of abuse.
13. **Database Security:** Implement secure database practices, such as parameterized queries or prepared statements, to prevent SQL injection attacks, and ensure proper access control and encryption of sensitive data.
14. **Server Hardening:** Secure the underlying server infrastructure by following best practices for server configuration, firewall settings, and restricting unnecessary services and ports.
15. **Regular Monitoring and Logging:** Implement logging and monitoring systems to detect and respond to security incidents, unusual activities, and potential threats in a timely manner.

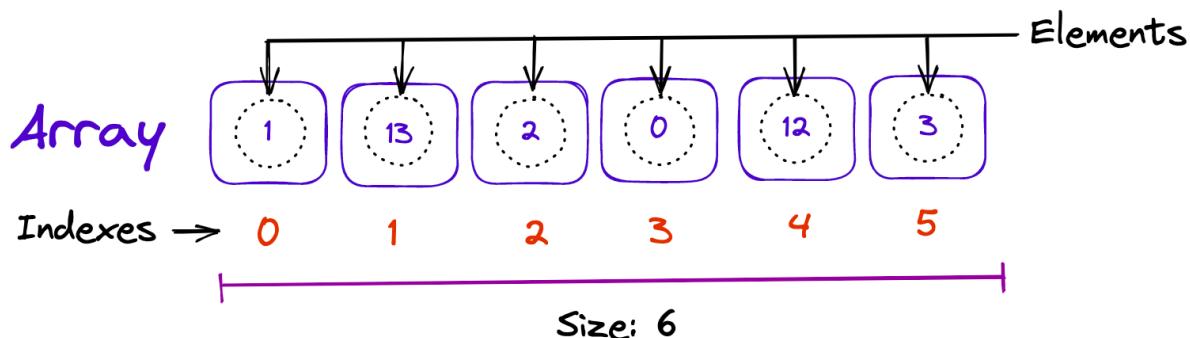
- Go back

## structures

### Array

### Array

An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



Basic Operations:

1. **Traverse** - print all the array elements one by one
2. **Insertion** - adds an element at the given index
3. **Deletion** - deletes an element at the given index
4. **Search** - searches an element using the given index or by the value
5. **Update** - updates an element at the given index

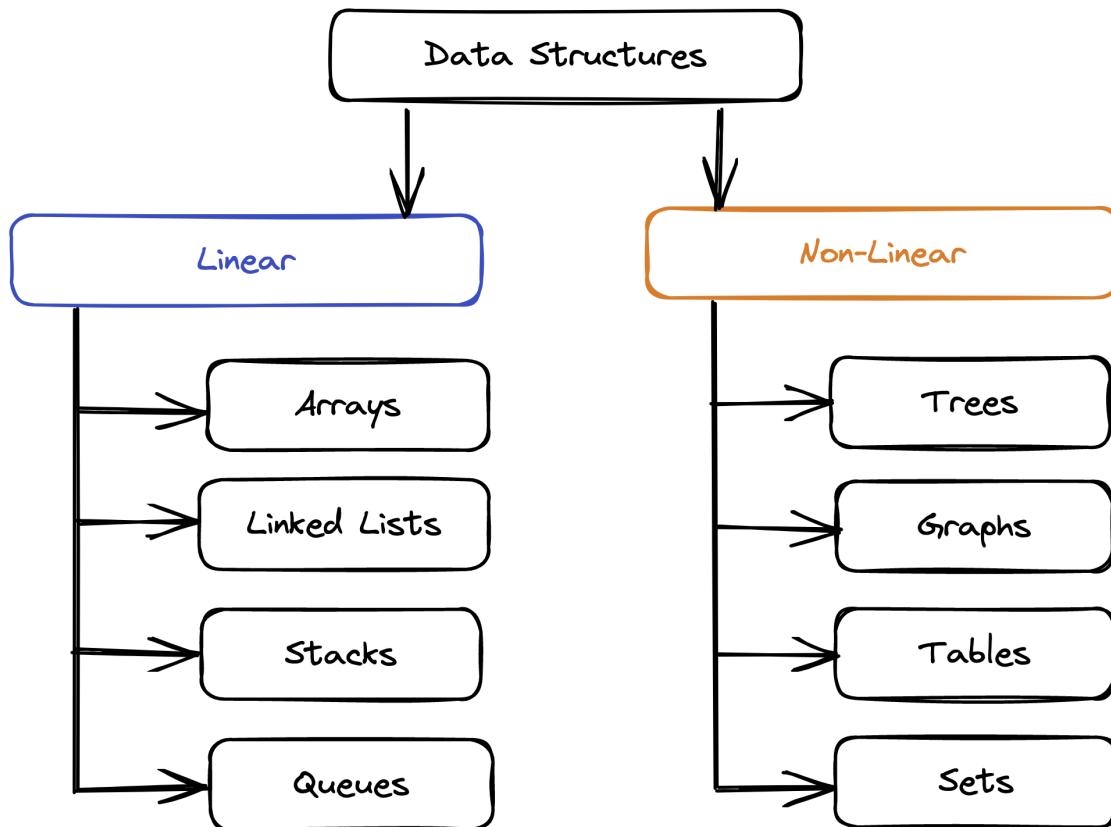
## Common Data Structures

# Common Data Structures

Data structure	Time Complexity								Space Complexity	
	Average				Worst					
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion		
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$	
Hash Table	-	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	-	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Cartesian Tree	-	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	-	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
Red-Black Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
Splay Tree	-	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	-	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
AVL Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
KD Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	

## Data structures types

### Data structures types



- **Linear Data Structure:** A data structure that includes data elements arranged sequentially or linearly, where each element is connected to its previous and next nearest elements, is referred to as a linear data structure. Arrays and linked lists are two examples of linear data structures.
- **Non-Linear Data Structure:** Non-linear data structures are data structures in which data elements are not arranged linearly or sequentially. We cannot walk through all elements in one pass in a non-linear data structure, as in a linear data structure. Trees and graphs are two examples of non-linear data structures.

#### Doubly Linked Lists

### Doubly Linked Lists

```
class Node<T> {  
    data: T;
```

```

prev: Node<T> | null;
next: Node<T> | null;

constructor(data: T, prev: Node<T> | null) {
    this.data = data;
    this.prev = prev;
    this.next = null;
}
}

class DoublyLinkedList<T> {
head: Node<T> | null;
tail: Node<T> | null;

constructor() {
    this.head = null;
    this.tail = null;
}

append(data: T): void {
    if (!this.head) {
        this.head = new Node(data, null);
        this.tail = this.head;
    } else {
        const newNode = new Node(data, this.tail);
        this.tail!.next = newNode;
        this.tail = newNode;
    }
}

prepend(data: T): void {
    if (!this.head) {
        this.head = new Node(data, null);
        this.tail = this.head;
    } else {
        const newNode = new Node(data, null);
        newNode.next = this.head;
        this.head!.prev = newNode;
        this.head = newNode;
    }
}

delete(data: T): void {
    let current = this.head;

    while (current) {
        if (current.data === data) {
            if (current.prev) {
                current.prev.next = current.next;
            } else {
                this.head = current.next;
            }
        }
    }
}

```

```

        if (current.next) {
            current.next.prev = current.prev;
        } else {
            this.tail = current.prev;
        }

        return;
    }

    current = current.next!;
}
}

display(): void {
    let current = this.head;

    while (current) {
        console.log(current.data);
        current = current.next!;
    }
}

reverse(): void {
    let current = this.head;
    let temp: Node<T> | null = null;

    while (current) {
        temp = current.prev;
        current.prev = current.next;
        current.next = temp;
        current = current.prev!;
    }

    temp = this.head;
    this.head = this.tail;
    this.tail = temp!;
}
}

```

## Doubly Linked Lists

A linked list is a fundamental data structure that consists of a sequence of elements, each connected to the next by pointers or references. Among the variations of linked lists, the doubly linked list stands out for its bidirectional navigation, providing easy traversal in both forward and backward directions. Let's explore the concept of a doubly linked list using a TypeScript implementation.

### Anatomy of a Doubly Linked List

The TypeScript code provided defines two classes: `Node` and `DoublyLinkedList`.

## Node Class

The `Node` class represents an individual element in the doubly linked list. It contains three properties:

- `data`: Holds the actual data of the node.
- `prev`: Points to the previous node in the sequence.
- `next`: Points to the next node in the sequence.

## DoublyLinkedList Class

The `DoublyLinkedList` class is the container for the nodes. It contains two pointers:

- `head`: Points to the first node in the list.
- `tail`: Points to the last node in the list.

Additionally, the class has methods for common operations:

- `append(data: T)`: Adds a new node with the given data to the end of the list.
- `prepend(data: T)`: Inserts a new node with the given data at the beginning of the list.
- `delete(data: T)`: Removes the node containing the specified data from the list.
- `display()`: Displays the elements of the linked list.
- `reverse()`: Reverses the order of the linked list.

## Example Usage

The code concludes with an example demonstrating the usage of the doubly linked list:

```
const doublyLinkedList = new DoublyLinkedList<number>();
doublyLinkedList.append(1);
doublyLinkedList.append(2);
doublyLinkedList.prepend(0);
doublyLinkedList.display(); // Output: 0 1 2
doublyLinkedList.reverse();
doublyLinkedList.display(); // Output: 2 1 0
```

This example initializes a doubly linked list, appends and prepends nodes, displays the list, performs a reversal, and then displays the reversed list.

## Advantages of Doubly Linked Lists

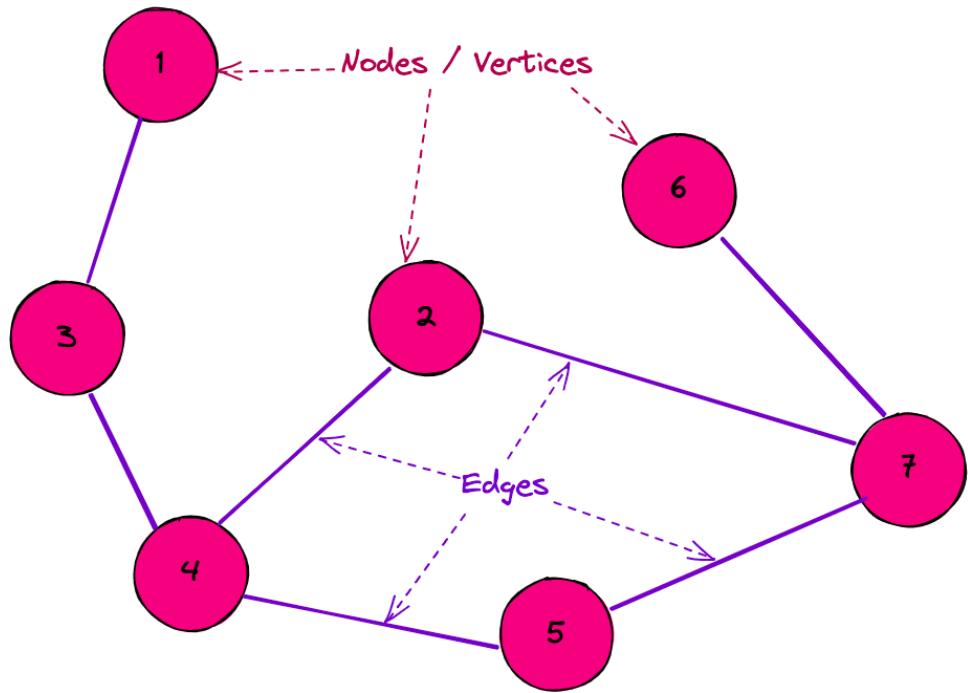
Doubly linked lists offer advantages in certain scenarios. The bidirectional navigation allows for efficient traversal in both directions, facilitating operations that involve searching, insertion, or deletion near a specific element.

Understanding and implementing doubly linked lists in TypeScript provides valuable insights into data structures and algorithms. The versatility of linked lists makes them powerful tools for various applications, from low-level memory management to high-level algorithmic problem-solving.

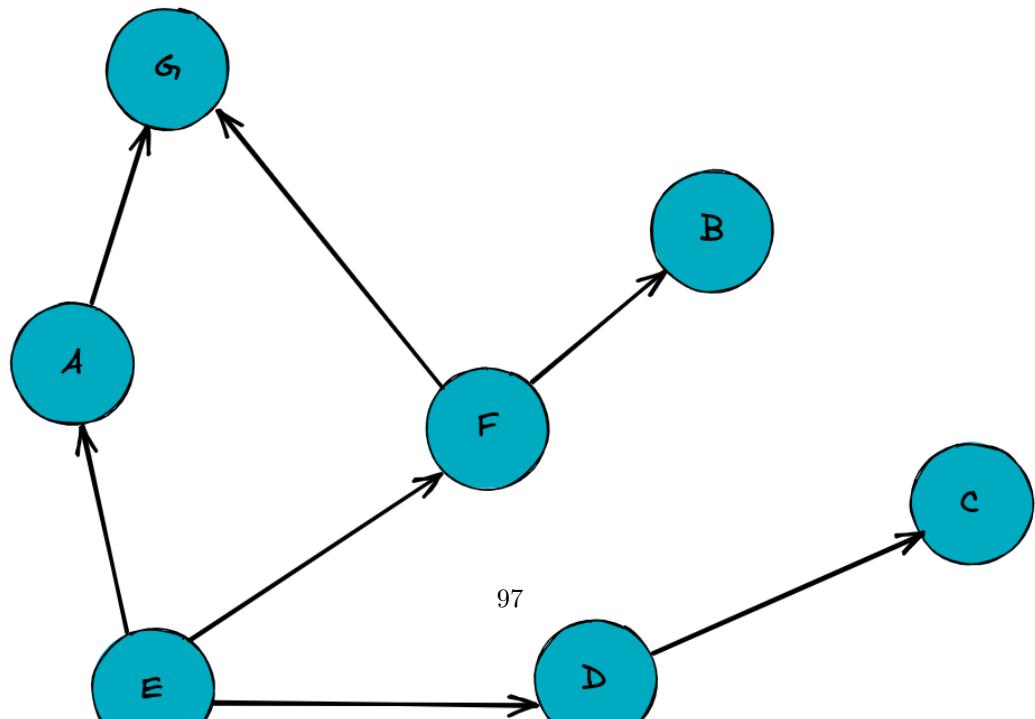
Graph

Graph

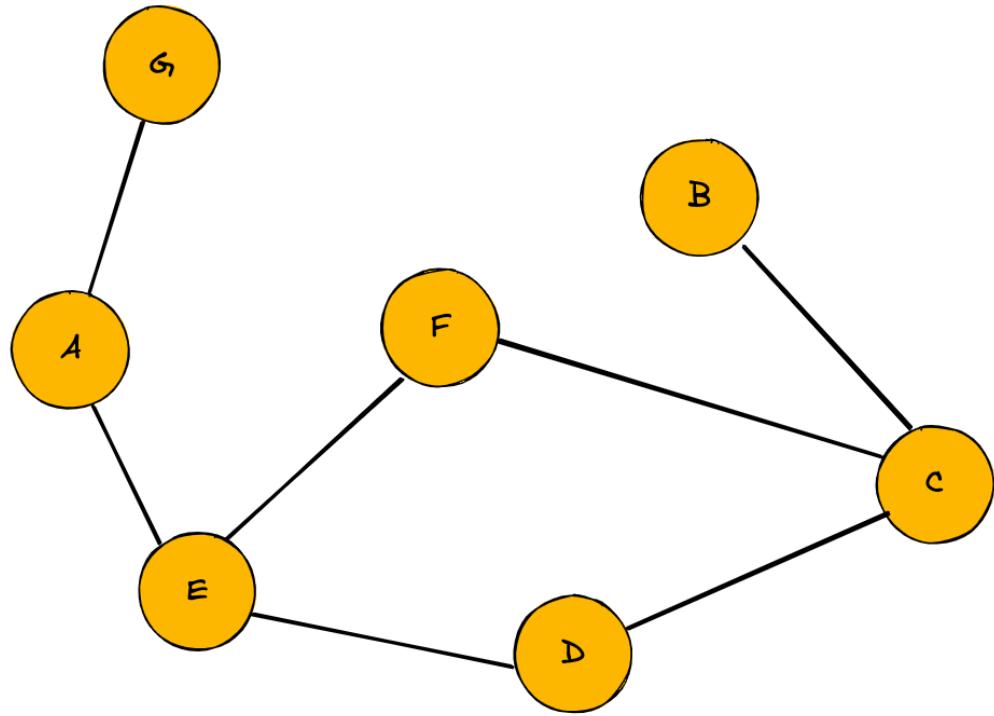
Graph



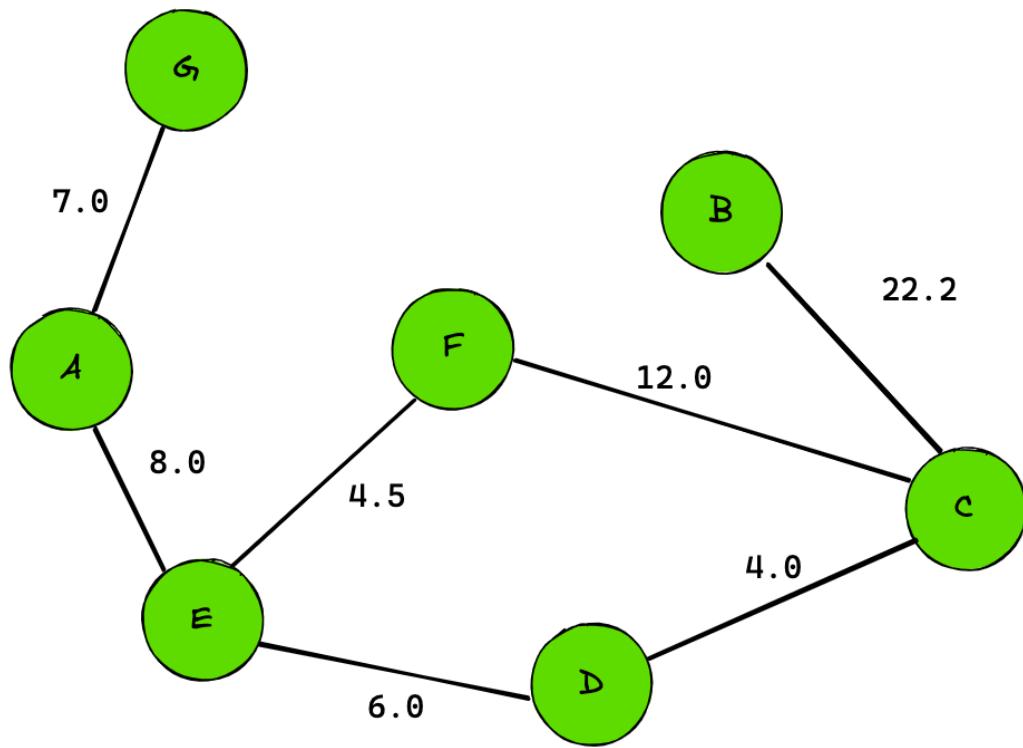
Directed graph



Undirected graph



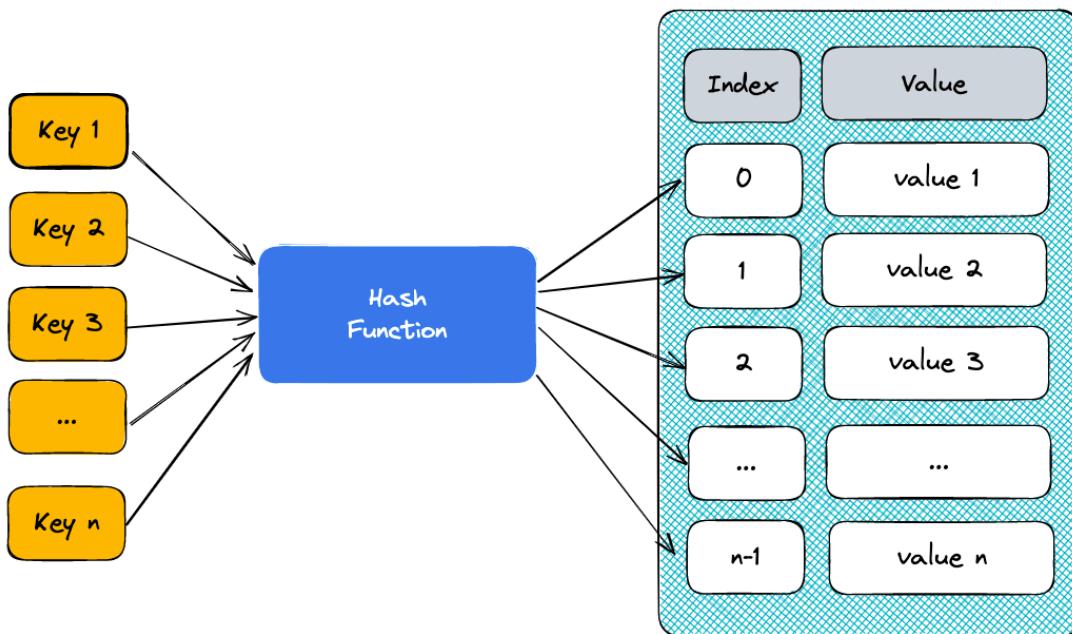
## Weighted graph



Basic Operations:

## Hash table

### Hash table



```
const superPrimitiveHashingFunc = (string: string) => {
    var hash = 0;
    for (var i = 0; i < string.length; i++) { hash += string.charCodeAt(i); }
    return hash;
}

export class HashTable {
    private collection: any = {}

    public add(key: any, value: any) {
        const theHash = superPrimitiveHashingFunc(key);
        if (!this.collection.hasOwnProperty(theHash)) {
            this.collection[theHash] = {};
        }
        this.collection[theHash][key] = value;
    }

    public remove(key: any) {
        const hashedObj = this.collection[superPrimitiveHashingFunc(key)];
        if (hashedObj.hasOwnProperty(key)) {
            delete hashedObj[key];
        }
        if (!Object.keys(hashedObj).length) {
    }
```

```

        delete this.collection[superPrimitiveHashingFunc(key)];
    }
}

public lookup(key: any) {
    var theHash = superPrimitiveHashingFunc(key);
    if (this.collection.hasOwnProperty(theHash)) {
        return this.collection[theHash][key];
    }
    return null;
}

```

A hash table is a data structure that allows you to store and retrieve values associated with a specific key in an efficient manner. It uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found.

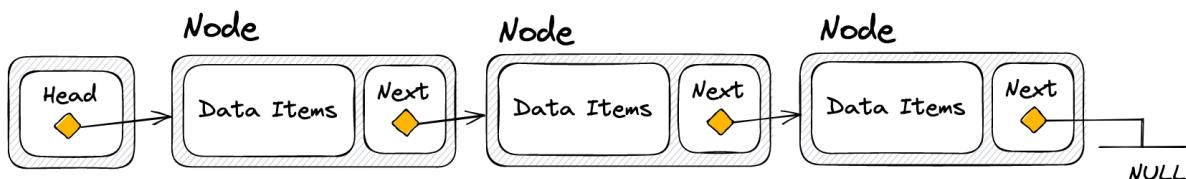
Here's how it works:

- 1. Hash Function:** When you want to store data in a hash table, a hash function is applied to the key. The hash function converts the key into an index, a numeric value that represents the position in the hash table array where the associated value will be stored.
- 2. Array of Buckets:** The hash table consists of an array (often called a hash table array) where values are stored. Each index in the array is known as a bucket or slot.
- 3. Collisions:** Since hash functions can produce the same index for different keys (a situation known as a collision), hash tables must have a strategy for handling these collisions. Common techniques include chaining (where each bucket contains a linked list of key-value pairs) and open addressing (where the hash table searches for the next open slot in the array).
- 4. Efficient Operations:** Hash tables provide fast insertion, deletion, and lookup operations. When you want to find a value associated with a specific key, the hash function calculates the index, and then the hash table quickly retrieves the value from the corresponding bucket. The time complexity for these operations is often considered to be  $O(1)$  on average, making hash tables very efficient for handling large datasets.

Hash tables are widely used in various applications such as databases, caches, symbol tables in compilers, and in many other scenarios where fast data retrieval based on a key is required. The efficiency of hash tables largely depends on the quality of the hash function and the method used to handle collisions.

## Linked list

### Linked list



```

export class Noddy {
    public element = null;
    public next = null;
    constructor(element: any) {
        this.element = element;
    }
}

export class LinkedList {
    private length = 0;
    private _head: any = null;

    public size() {
        return this.length;
    }

    public head() {
        return this._head;
    }

    public add(element: any) {
        const node = new Noddy(element);
        if (this._head === null) {
            this._head = node;
        } else {
            let currentNode = this._head;
            while (currentNode.next) {
                currentNode = currentNode.next;
            }
            currentNode.next = node;
        }
        this.length++;
    }

    public remove(element: any) {
        let currentNode = this._head;
        let previousNode = null;
        if (currentNode.element === element) {
            this._head = currentNode.next;
        } else {
            while (currentNode.element !== element) {
                previousNode = currentNode;
                currentNode = currentNode.next;
            }
            previousNode.next = currentNode.next;
        }
        this.length -= 1;
    }

    public isEmpty() {
        return !(this.size() > 0);
    }
}

```

```

}

public indexOf(element: any) {
    if (this._head === null) return -1;
    let current = this._head;
    let index = 0;
    while (current.element !== element && current.next !== null) {
        current = current.next;
        index++
    }
    if (current.element !== element && current.next === null) {
        return -1
    }
    return index;
}

public elementAt(index: number) {
    if (this._head === null) return undefined;
    let current = this._head;
    let currentIndex = 0;
    while (currentIndex !== index && current.next !== null) {
        current = current.next;
        currentIndex++
    }
    if (currentIndex !== index && current.next === null) {
        return undefined;
    }
    return current.element;
}

public removeAt(index: number) {
    if (this._head === null) return undefined;
    let current = this._head;
    let currentIndex = 0;
    while (currentIndex !== index && current.next !== null) {
        current = current.next;
        currentIndex++
    }
    if (currentIndex !== index && current.next === null) {
        return null;
    }
    this.remove(current.element)
    return current.element;
}

public addAt(index: number, element: any) {
    let node = new Noddy(element);
    if (index < 0 || index > length) {
        return false;
    }
    if (index == 0) {
        node.next = this._head
        this._head = node
    }
}

```

```

        length++;
        return true;
    } else {
        let currentnode = this._head;
        let currentindex = 0;
        let previousnode;
        while (index !== currentindex) {
            previousnode = currentnode;
            currentnode = currentnode.next;
            currentindex++;
        }
        previousnode.next = node;
        node.next = currentnode;
        length++;
        return true
    }
}
}

```

## Map

### Map

```

export class Map {

    private collection: any = {}

    public add(key: any, value: any) {
        this.collection[key] = value;
    }

    public delete(key: any) {
        delete this.collection[key]
    }

    public get(key: any) {
        return this.collection[key]
    }

    public has(key: any) {
        return Object.keys(this.collection).includes(key);
    }

    public values() {
        return Object.values(this.collection)
    }
}

```

```

public size() {
    return Object.keys(this.collection).length
}

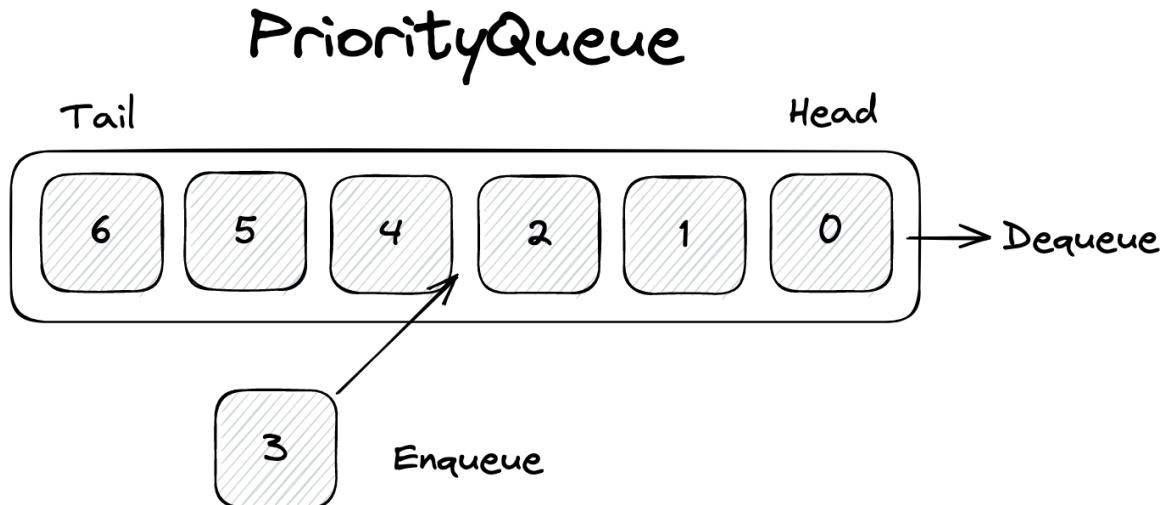
public clear() {
    this.collection = []
}

}

```

## Priority Queue

### Priority Queue



```

export class PriorityQueue {
    private collection: any[] = []

    public printCollection() {
        return this.collection;
    }

    public enqueue(element: any) {
        if (this.isEmpty()) {
            return this.collection.push(element);
        }
        this.collection = this.collection.reverse()
        let index = this.collection.findIndex((item) => {
            return element[1] >= item[1];
        });
        if (index === -1) {

```

```

        this.collection.push(element);
    } else {
        this.collection.splice(index, 0, element);
    }
    this.collection = this.collection.reverse()
}

public dequeue() {
    if (!this.isEmpty()) {
        return this.collection.shift()[0];
    } else {
        return "The queue is empty.";
    }
}

public front():any {
    return this.collection[0][0]
}

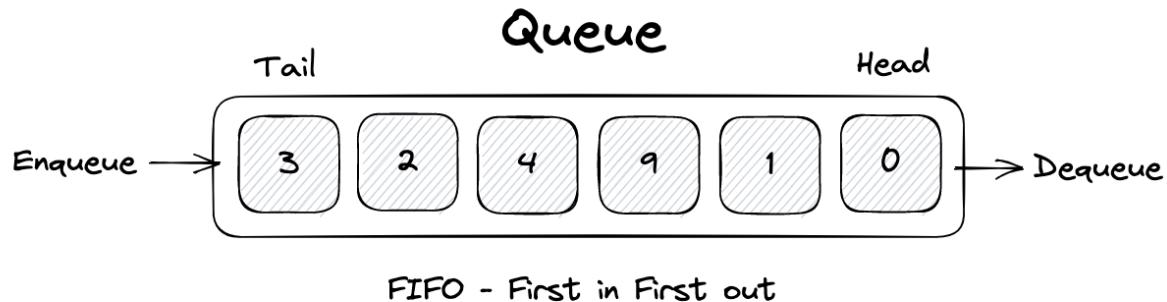
public size(): number {
    return this.collection.length
}

public isEmpty(): boolean {
    return !this.size();
}
}

```

## Queue

### Queue



```

export class Queue {
    private collection: any[] = [];
}

```

```

public print() {
    return this.collection;
}

public enqueue(element: any) {
    this.collection.push(element);
    return element;
}

public dequeue() {
    return this.collection.shift();
}

public front() {
    return this.collection[0];
}

public size() {
    return this.collection.length
}

public isEmpty() {
    return !this.collection.length
}
}

```

**Set**

**Set**

```

export class Set {
    private dictionary: any = {}
    private length = 0
    constructor(...elements: any) {
        if (elements.length > 0) {
            elements.forEach((el: any) => {
                this.add(el);
            });
        }
    }

    public has(element: any) {
        return this.dictionary[element] !== undefined;
    }

    public values() {
        return Object.keys(this.dictionary);
    }

    public add(element: any) {
        if (!this.has(element)) {

```

```

        this.dictionary[element] = true;
        this.length++;
        return true;
    }

    return false;
}

public delete(element: any) {
    if (this.has(element)) {
        delete this.dictionary[element];
        this.length--;
        return true;
    }

    return false;
}

public size() {
    return this.length;
}

public union(set: any) {
    const newSet = new Set();
    this.values().forEach((value: any) => {
        newSet.add(value);
    })
    set.values().forEach((value: any) => {
        newSet.add(value);
    })

    return newSet;
}

public intersection(set: any) {
    const newSet = new Set();

    let big: any;
    let small: any;
    if (this.dictionary.length > set.length) {
        big = this;
        small = set;
    } else {
        big = set;
        small = this;
    }

    small.values().forEach((value: any) => {
        if (big.dictionary[value]) {
            newSet.add(value);
        }
    })
}

```

```

        return newSet;
    }

    public difference(set:any) {
        const newSet = new Set();

        let big: any;
        let small: any;
        if (this.dictionary.length > set.length) {
            big = this;
            small = set;
        } else {
            big = set;
            small = this;
        }

        small.values().forEach((value: any) => {
            if (!big.dictionary[value]) {
                newSet.add(value);
            }
        })
    }

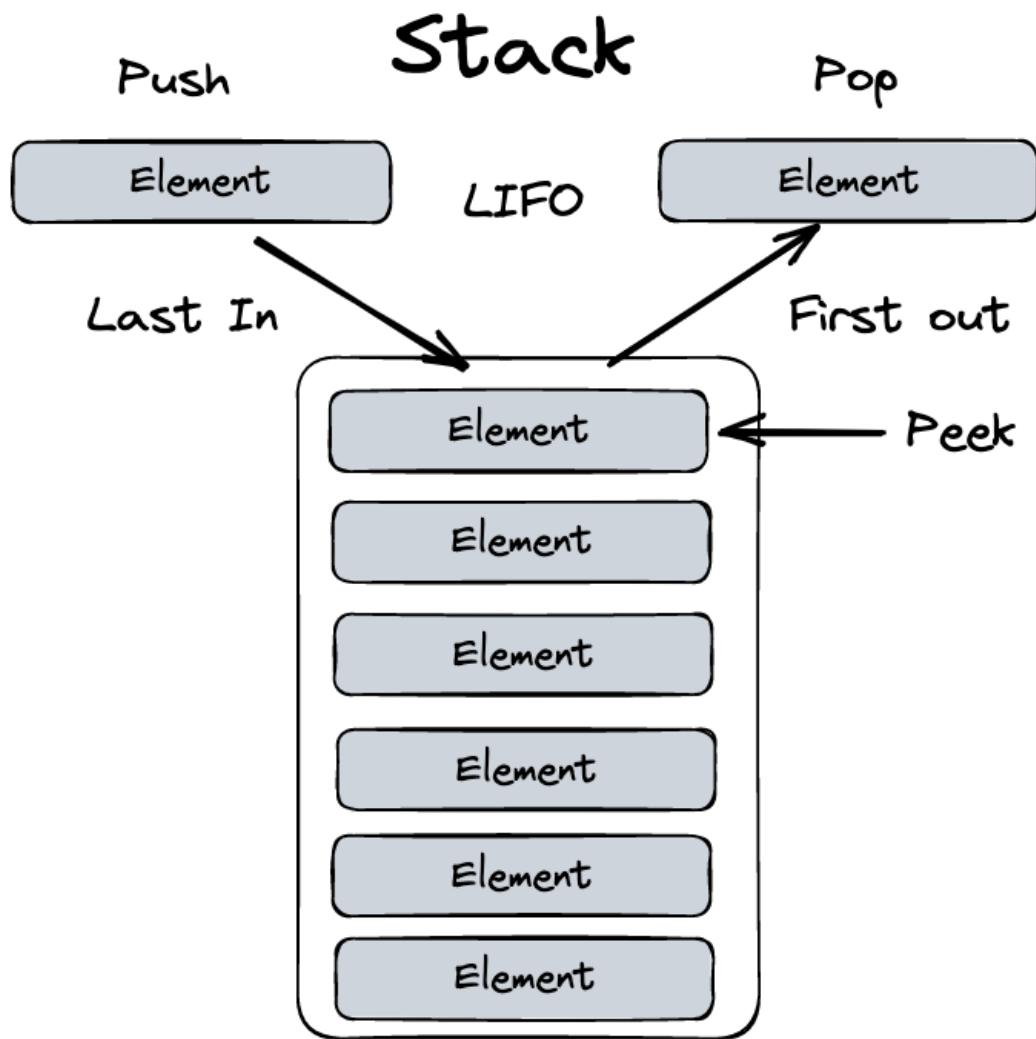
    return newSet;
}

public isSubsetOf(set: any) {
    let counter = 0;
    this.values().forEach((el) => {
        if (set.has(el)) {
            counter++;
        }
    });
    return this.values().length === counter;
}
}

```

Stack

Stack



```
export class Stack {  
    private collection:any[] = [];  
  
    public print() {  
        return this.collection;  
    }  
  
    public push(element: any) {  
        this.collection.push(element);  
        return element;  
    }  
}
```

```

    public pop() {
        return this.collection.pop();
    }

    public peek() {
        return this.collection[this.collection.length - 1];
    }

    public isEmpty() {
        return !this.collection.length;
    }

    public clear() {
        this.collection = []
    }
}

```

```

class Stack {

    int[] stack;
    int top, capacity, currentSize;

    public Stack (int capacity) {
        this.capacity = capacity;
        top = -1;
        currentSize = 0;
        stack = new int[this.capacity];
    }

    public boolean isEmpty() {
        return currentSize == 0;
    }

    public int size() {
        return currentSize;
    }

    public int top() {
        if(top < 0){
            return -1;
        }
        return stack[top];
    }

    public void push(int element) {
        if(top >= capacity - 1) {
            return;
        }
        stack[++top] = element;
        currentSize++;
    }

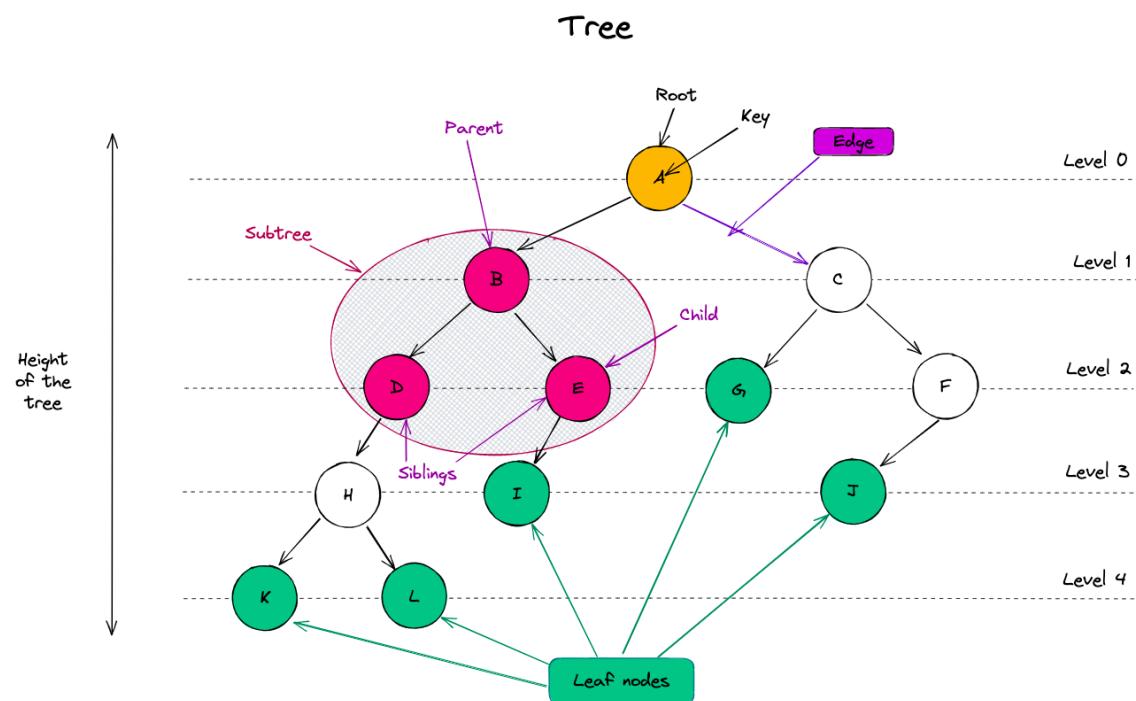
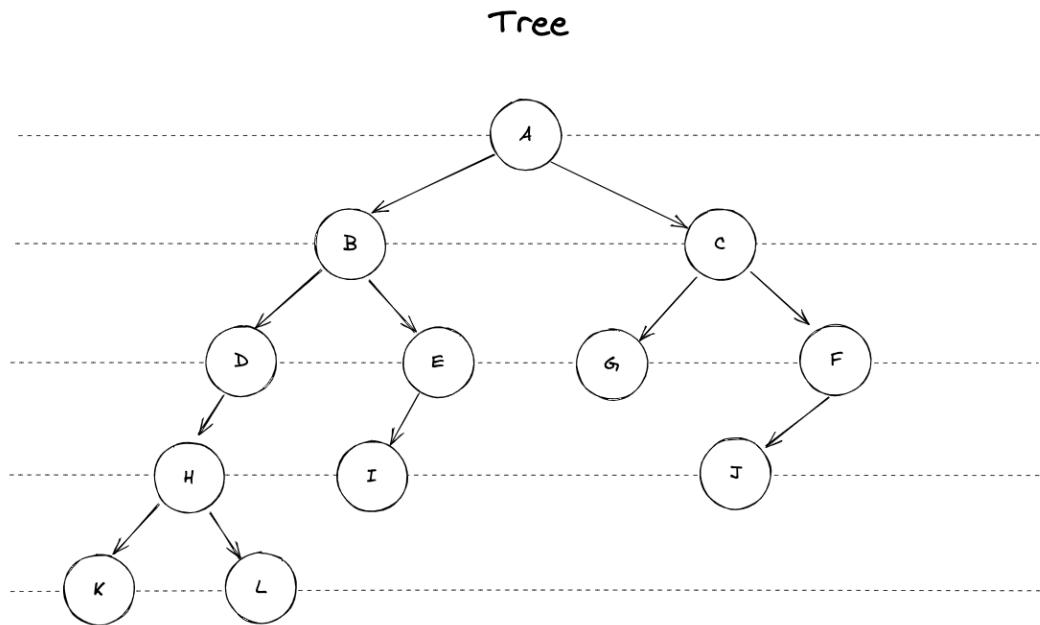
    public void pop() {

```

```
    if(top < 0) {
        return;
    }
    top--;
    currentSize--;
}
}
```

Tree

Tree



## Tree

```
class Tree {
  constructor() {
    this.root = null;
  }

  traverse(callback) {
    function walk(node) {
      callback(node);
      node.children.forEach(walk);
    }
    walk(this.root);
  }

  add(value, parentValue) {
    let newNode = {
      value,
      children: []
    };

    if (this.root === null) {
      this.root = newNode;
      return;
    }

    this.traverse(node => {
      if (node.value === parentValue) {
        node.children.push(newNode);
      }
    });
  }
}
```

## Binary search tree

```
class BinarySearchTree {

  constructor() {
    this.root = null;
  }

  contains(value) {
    let current = this.root;

    while (current) {

      if (value > current.value) {
        current = current.right;
      } else if (value < current.value) {
        current = current.left;
      } else {
        return true;
      }
    }

    return false;
  }
}
```

```

        current = current.left;

    } else {
        return true;
    }
}

return false;
}

add(value) {
    let node = {
        value: value,
        left: null,
        right: null
    };

    if (this.root === null) {
        this.root = node;
        return;
    }

    let current = this.root;

    while (true) {

        if (value > current.value) {

            if (!current.right) {
                current.right = node;
                break;
            }

            current = current.right;

        } else if (value < current.value) {

            if (!current.left) {
                current.left = node;
                break;
            }

            current = current.left;

        } else {
            break;
        }
    }
}
}

```

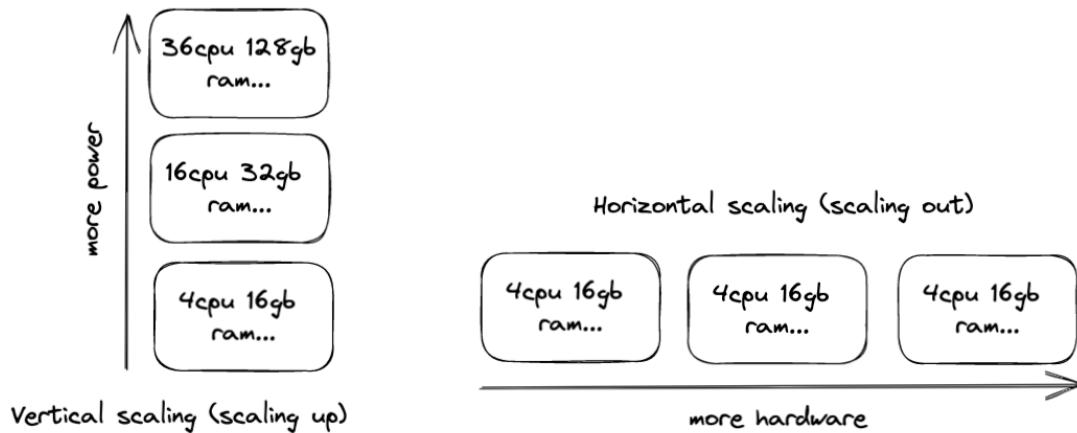
## system-design

### Horizontal and vertical scaling

## Horizontal and vertical scaling

Scalability refers to an application's ability to handle and withstand an increased workload without sacrificing latency. An application needs solid computing power to scale well. The servers should be powerful enough to handle increased traffic loads. There are two main ways to scale an application: horizontally and vertically.

Horizontal scaling, or scaling out, means adding more hardware to the existing hardware resource pool. It increases the computational power of the system as a whole. Vertical scaling, or scaling up, means adding more power to your server. It increases the power of the hardware running the application.



## CAP theorem

### CAP theorem

CAP theorem, also known as Brewer's theorem, has become a fundamental principle in the design and operation of distributed systems. Let's delve deeper into each aspect of the theorem and explore real-world examples and implications.

#### 1. Consistency:

Consistency in the context of CAP theorem means that all nodes in a distributed system have the same data at any given point in time. Ensuring consistency often involves coordination and synchronization between nodes. Strongly consistent systems guarantee that once a piece of data is written, all subsequent reads will return that value.

- **Example:** Traditional relational databases (like MySQL, PostgreSQL) often prioritize consistency. When a transaction is committed, the database ensures that the changes are immediately visible to all subsequent queries.

## **2. Availability:**

Availability refers to the ability of the system to respond to every request, without any downtime. Highly available systems remain operational and responsive, even in the face of failures.

- **Example:** NoSQL databases like Cassandra and Couchbase emphasize availability. They are designed to operate without interruption even if some of their nodes fail, ensuring that the service is always accessible.

## **3. Partition Tolerance:**

Partition tolerance means that the system continues to function even if network partitions occur, i.e., if communication between nodes is unreliable or slow. Partition tolerance is crucial for distributed systems because network failures are inevitable in real-world scenarios.

- **Example:** Distributed systems like Hadoop and Apache Kafka prioritize partition tolerance. They are built to handle network partitions and communication delays between nodes, ensuring the system's stability under adverse network conditions.

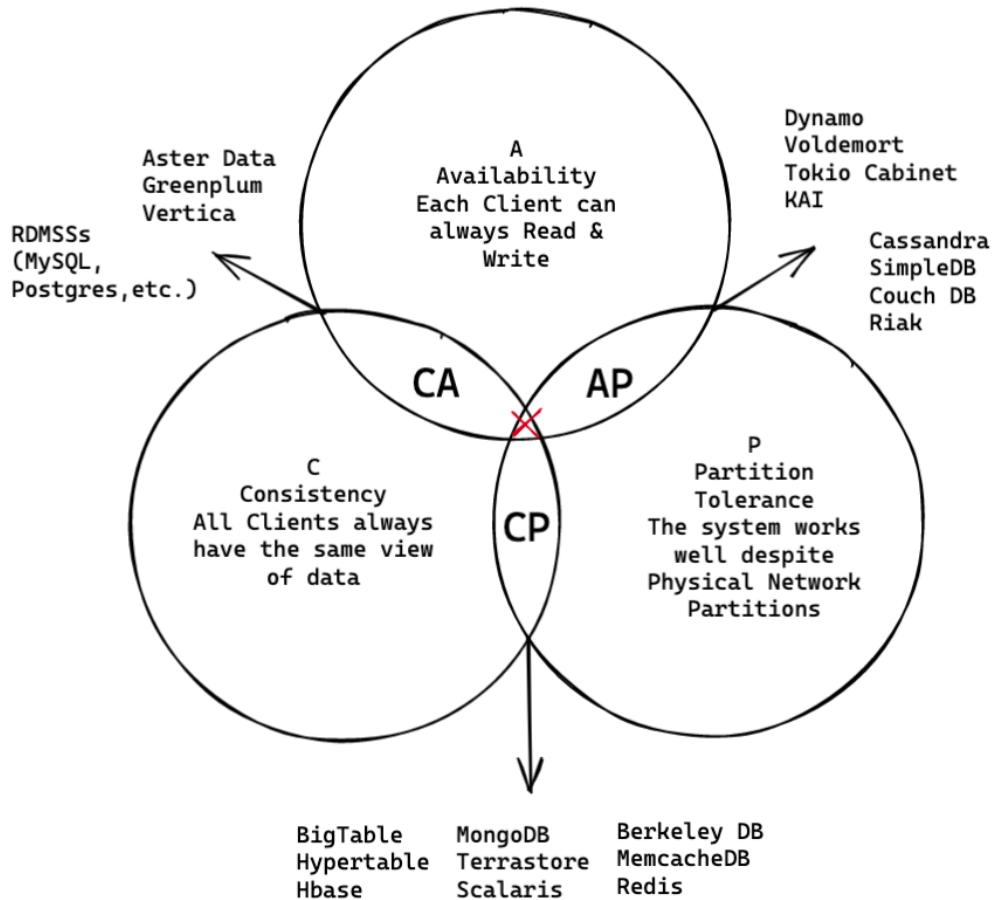
### **Real-World Scenarios and Trade-Offs:**

- **RDBMS (Relational Database Management Systems):** Traditional SQL databases are typically designed for consistency and availability. They sacrifice partition tolerance to ensure data integrity and accuracy. This means that in the event of a network partition, these systems might not be fully operational.
- **NoSQL Databases (Cassandra, Couchbase, MongoDB, Redis):** NoSQL databases often choose to sacrifice consistency under certain conditions to achieve high availability and partition tolerance. They provide eventual consistency, meaning that given enough time, all nodes in the system will converge to a consistent state, but immediate consistency might be compromised.
- **NewSQL Databases:** Some modern databases like Google Spanner and CockroachDB attempt to provide a balance between consistency, availability, and partition tolerance. They use innovative techniques like synchronized clocks and distributed transactions to achieve global consistency while maintaining high availability and partition tolerance.

### **Considerations for System Architects:**

1. **Understanding Requirements:** Architects need to assess the specific needs of their applications. Some applications, like banking systems, require strict consistency, while others, like social media platforms, can tolerate eventual consistency.
2. **Tuning for Trade-Offs:** Architects often need to make trade-offs based on the CAP theorem. For instance, during network partitions, architects might choose to sacrifice consistency temporarily to maintain system availability.
3. **Hybrid Approaches:** In many real-world scenarios, hybrid approaches are adopted. Different parts of the system might prioritize different aspects of the CAP theorem based on their requirements. For example, a system might use a consistent database for financial transactions while employing an eventually consistent database for non-critical data.

In summary, the CAP theorem provides a valuable framework for understanding the inherent trade-offs in distributed systems. Architects and developers must carefully consider these trade-offs based on the specific requirements and use cases of their applications.



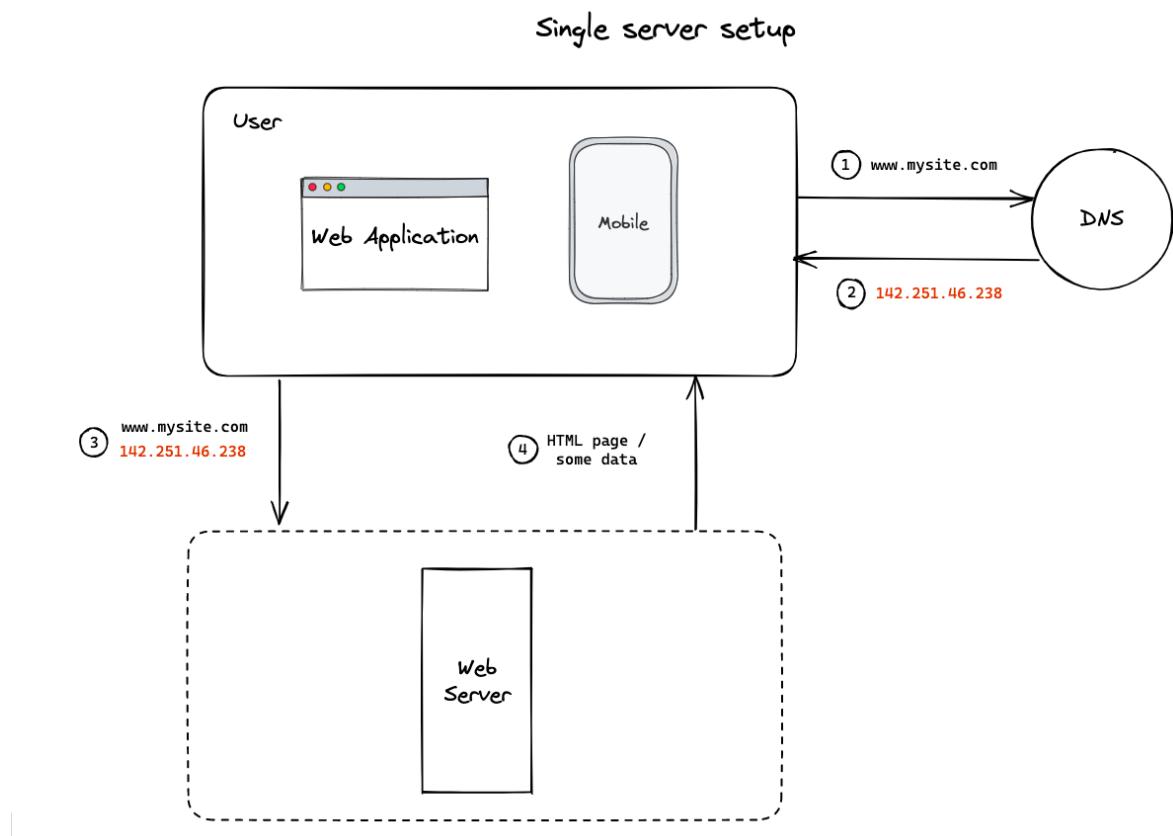
How DNS resolves ip address

How DNS resolves ip address

How dns resolves ip

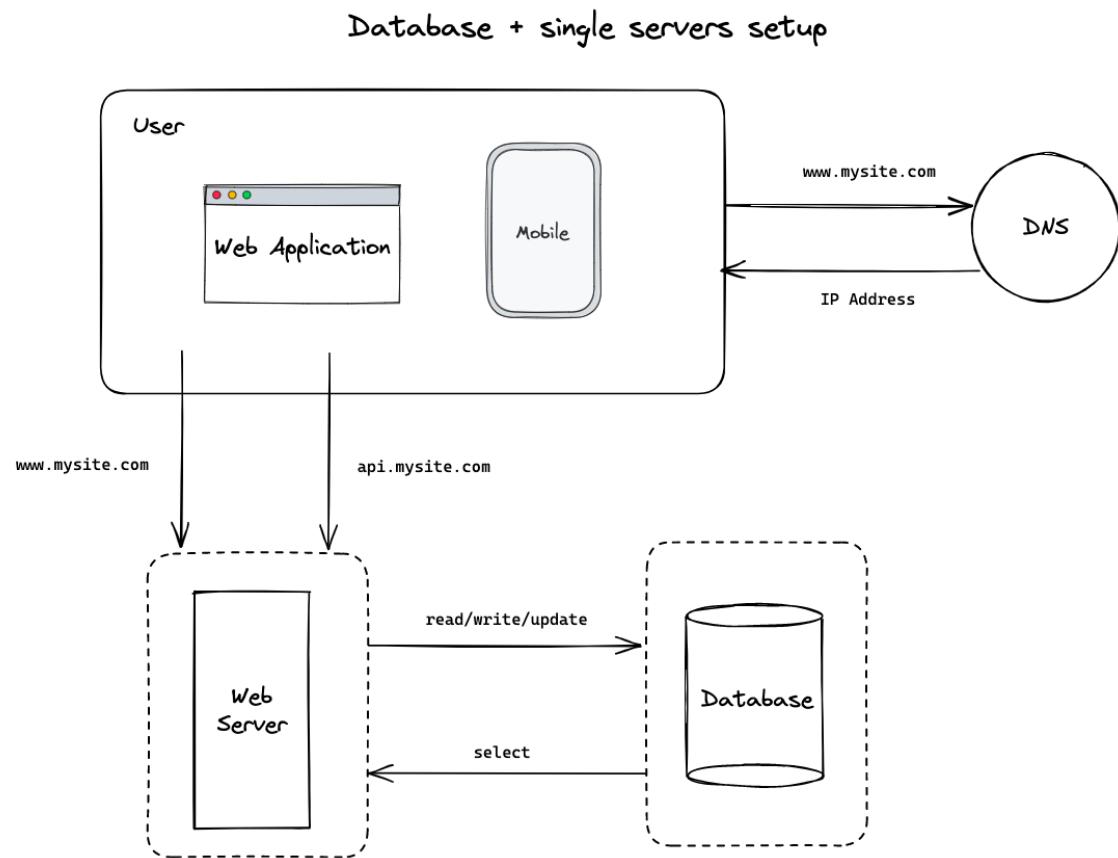
## Single server setup

### Single server setup



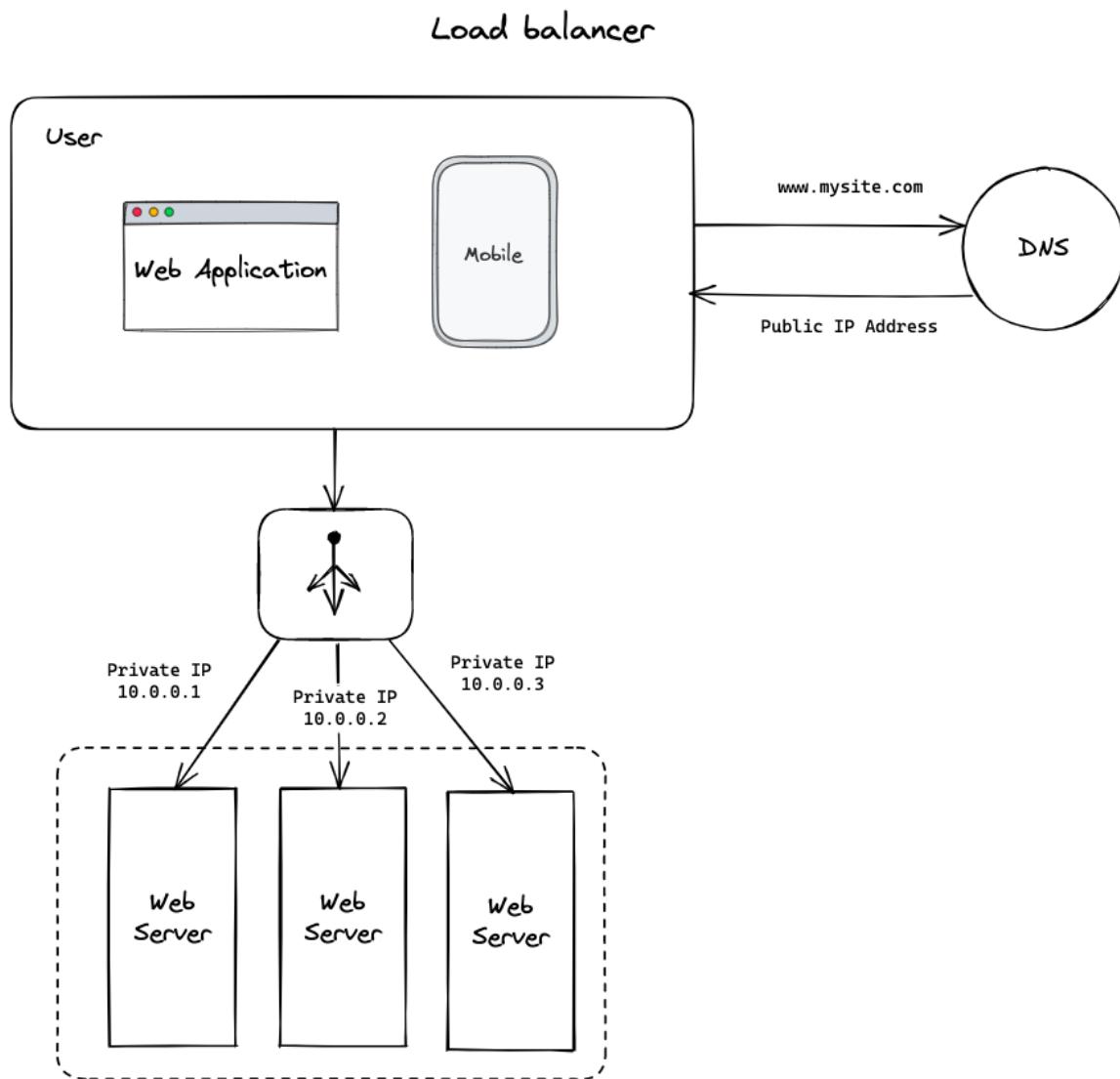
Database + Single servers setup

Database + Single servers setup



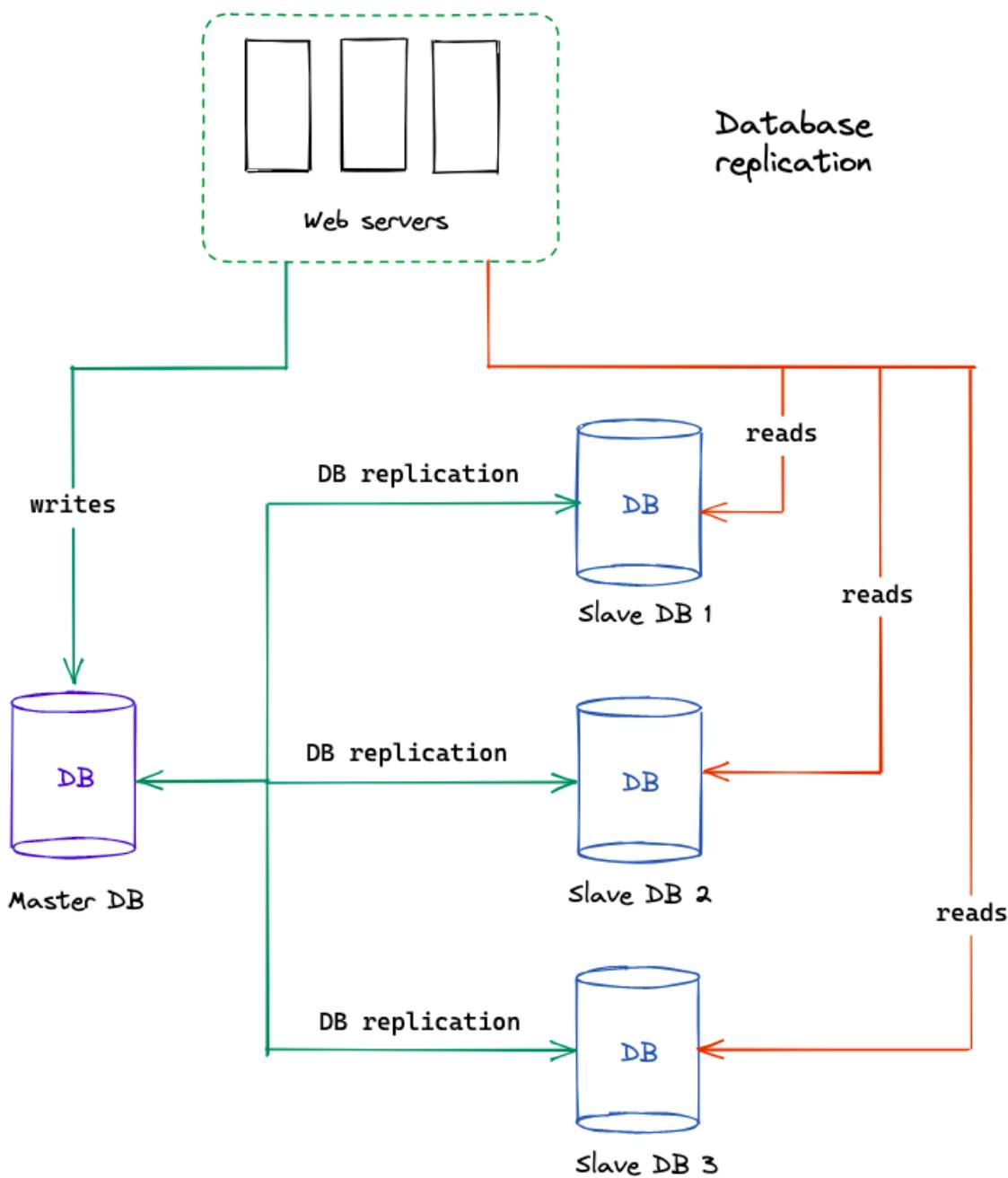
Load balancer

Load balancer



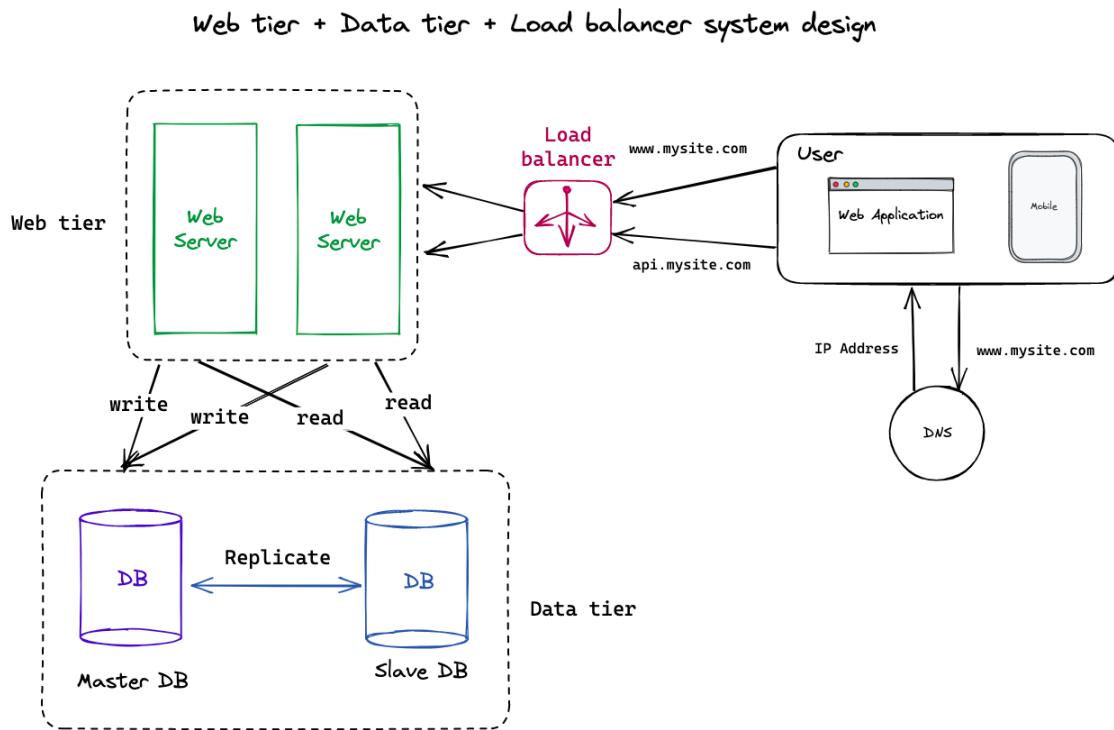
## Database replication

### Database replication



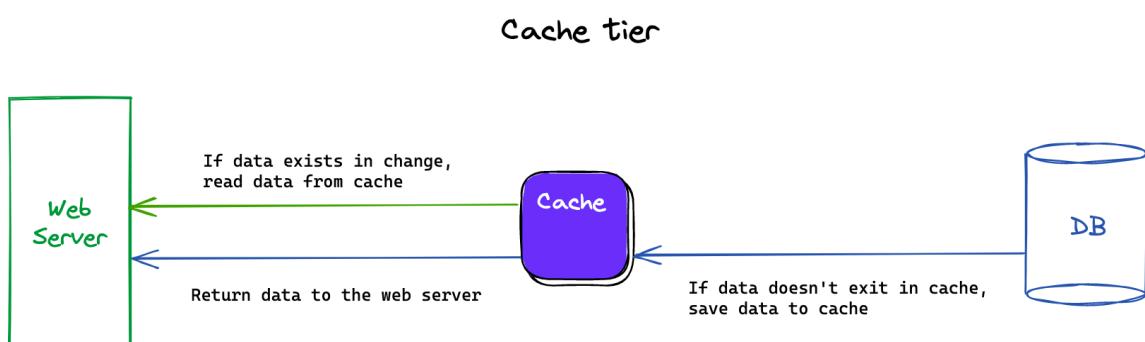
Web tier + Data tier + Load balancer System design

Web tier + Data tier + Load balancer System design



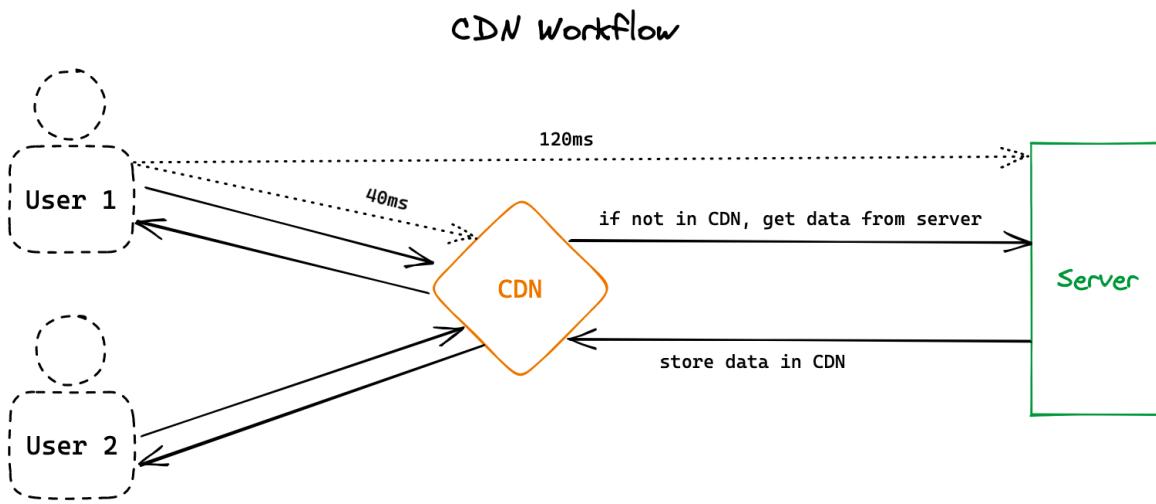
Cache tier

Cache tier



## CDN Workflow

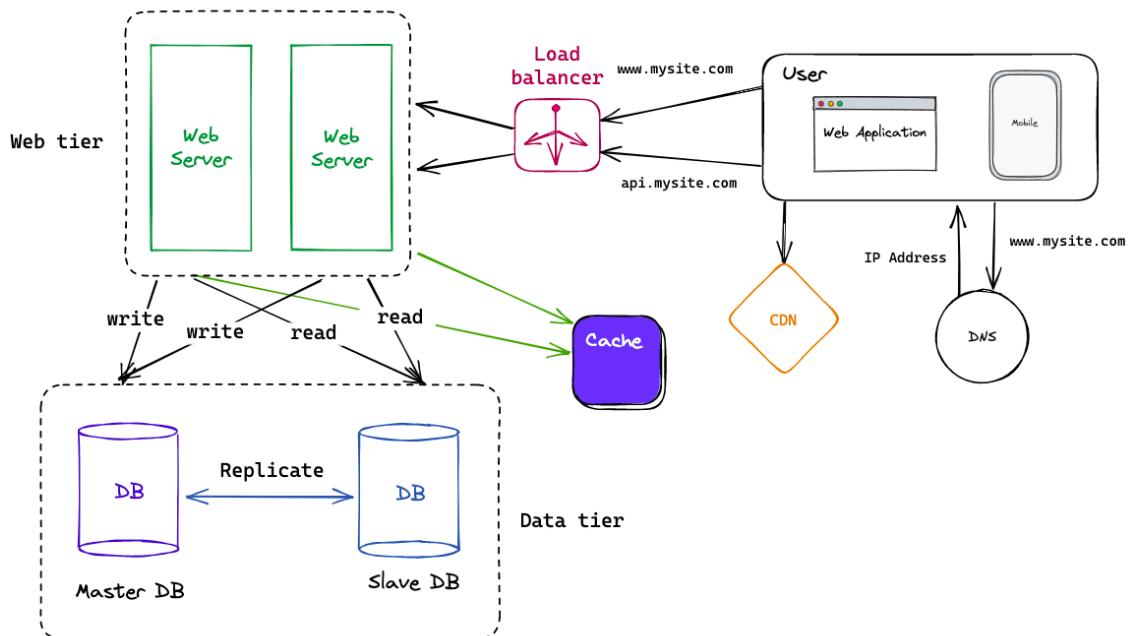
### CDN Workflow



Web tier + Data tier + Load balancer + Cache + CDN System design

Web tier + Data tier + Load balancer + Cache + CDN System design

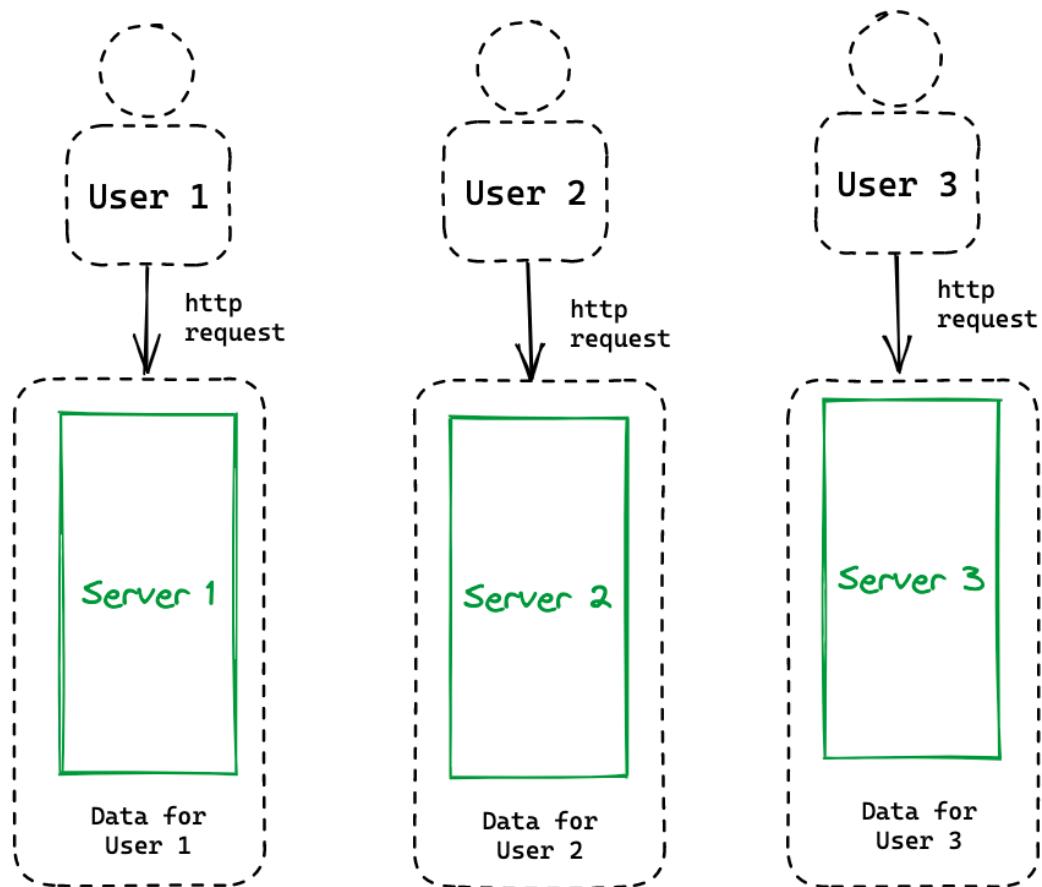
*Web tier + Data tier + Load balancer + Cache + CDN system design*



Stateful architecture

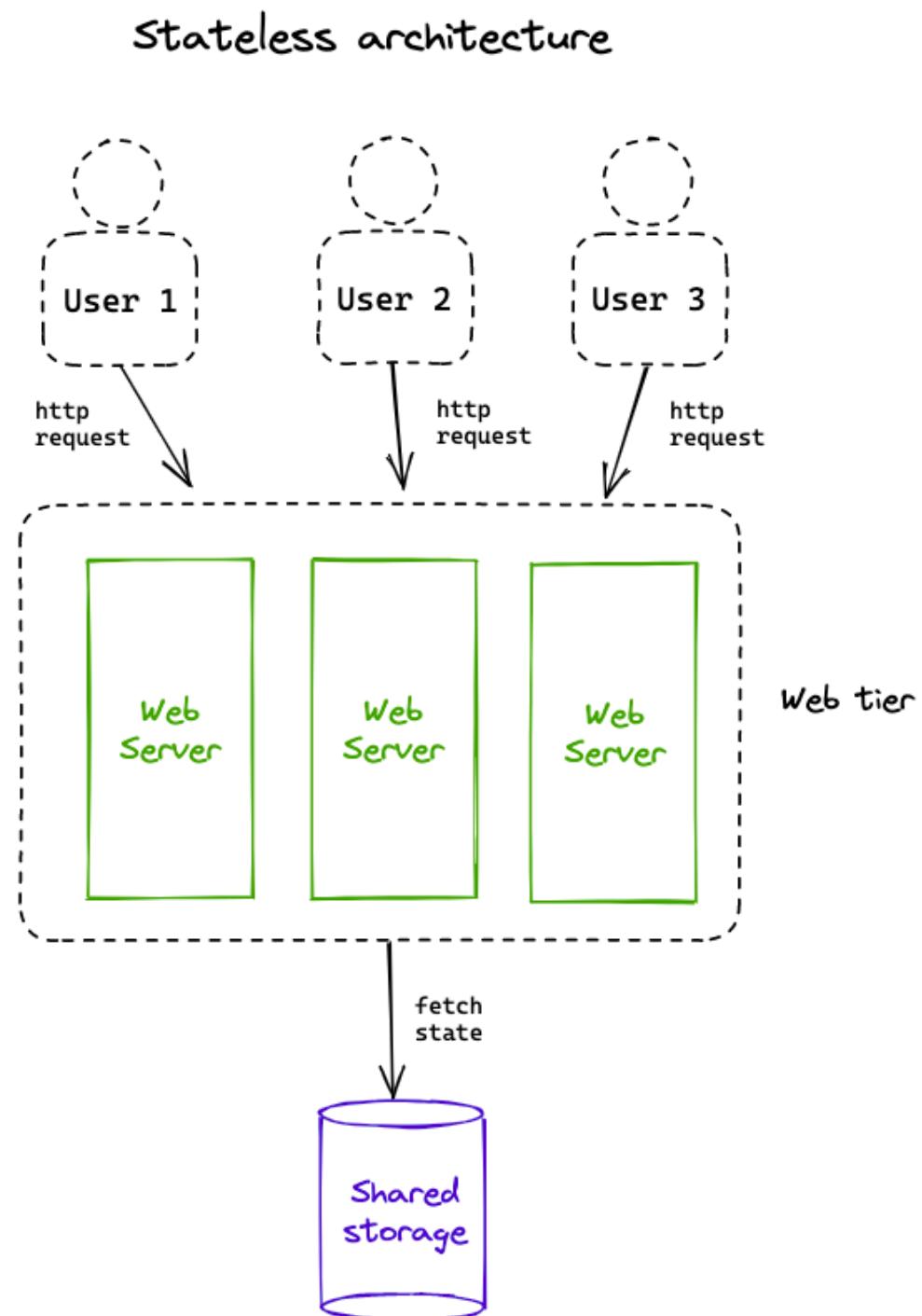
Stateful architecture

## Stateful architecture



Stateless architecture

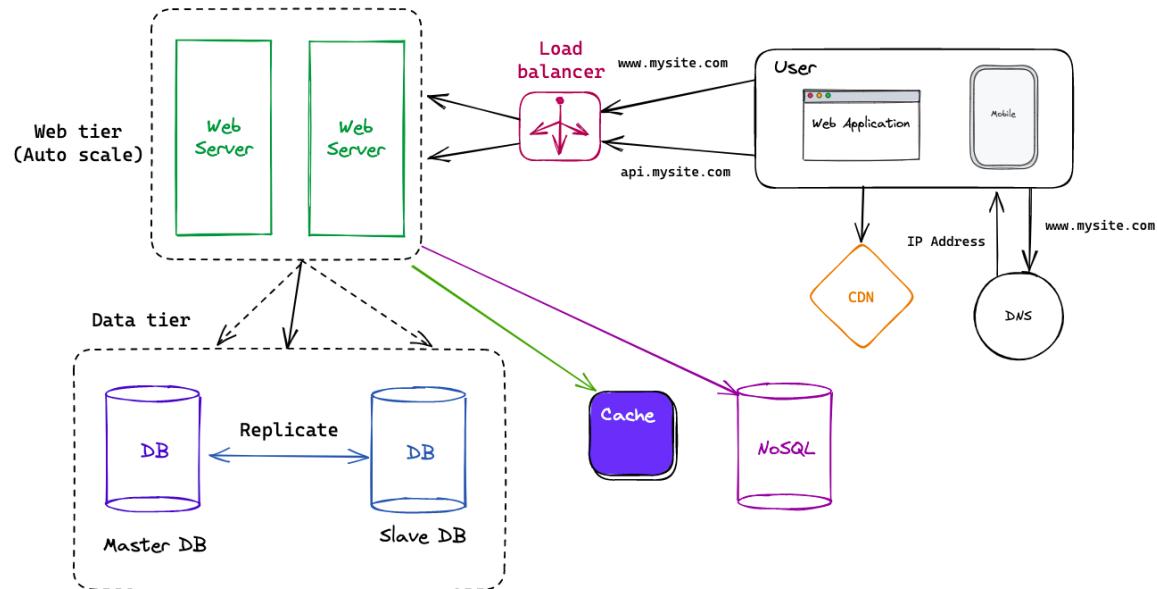
Stateless architecture



Web tier + Data tier + Load balancer + Cache + CDN stateless System design

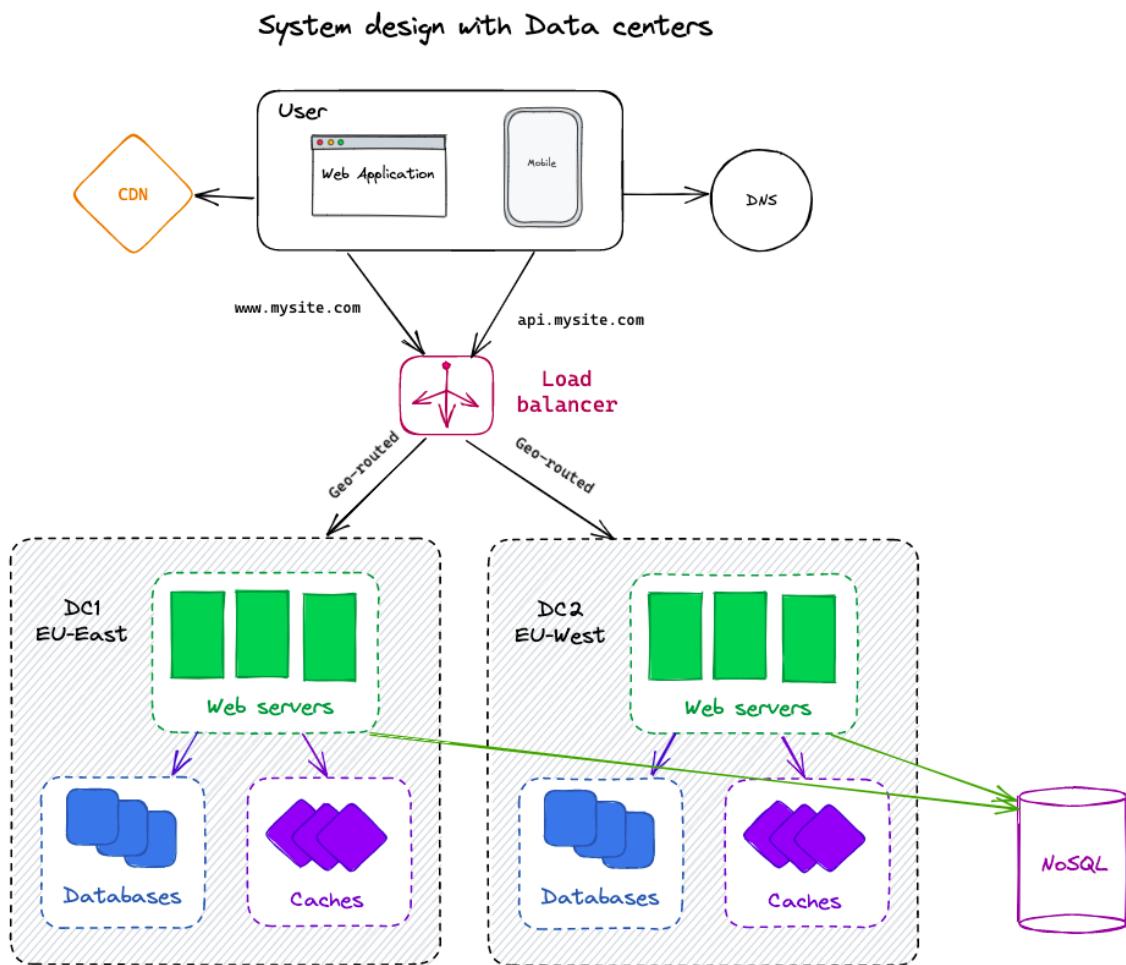
Web tier + Data tier + Load balancer + Cache + CDN stateless System design

Web tier (auto scale) + Data tier + Load balancer + Cache + CDN stateless system design



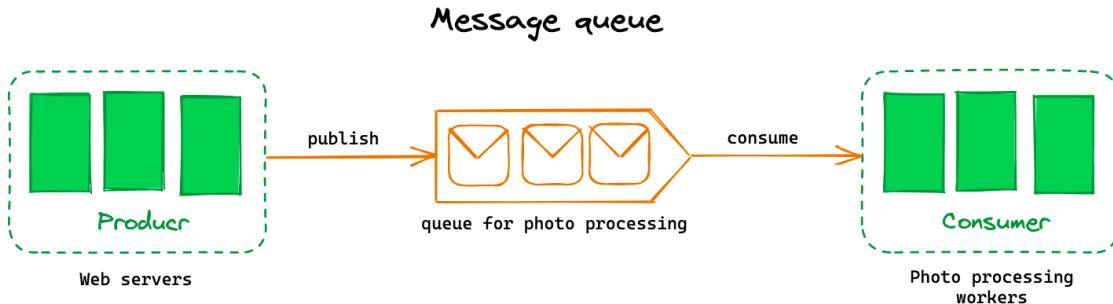
## System design with data centers

### System design with data centers



## Message queue

### Message queue



A message queue is a communication mechanism that allows different parts of a software system to send and receive messages or data asynchronously. It is often used to facilitate communication between different components or services within a distributed system. In the context you mentioned, where there is a server and a consumer, a message queue can be used to enable communication and data exchange between them.

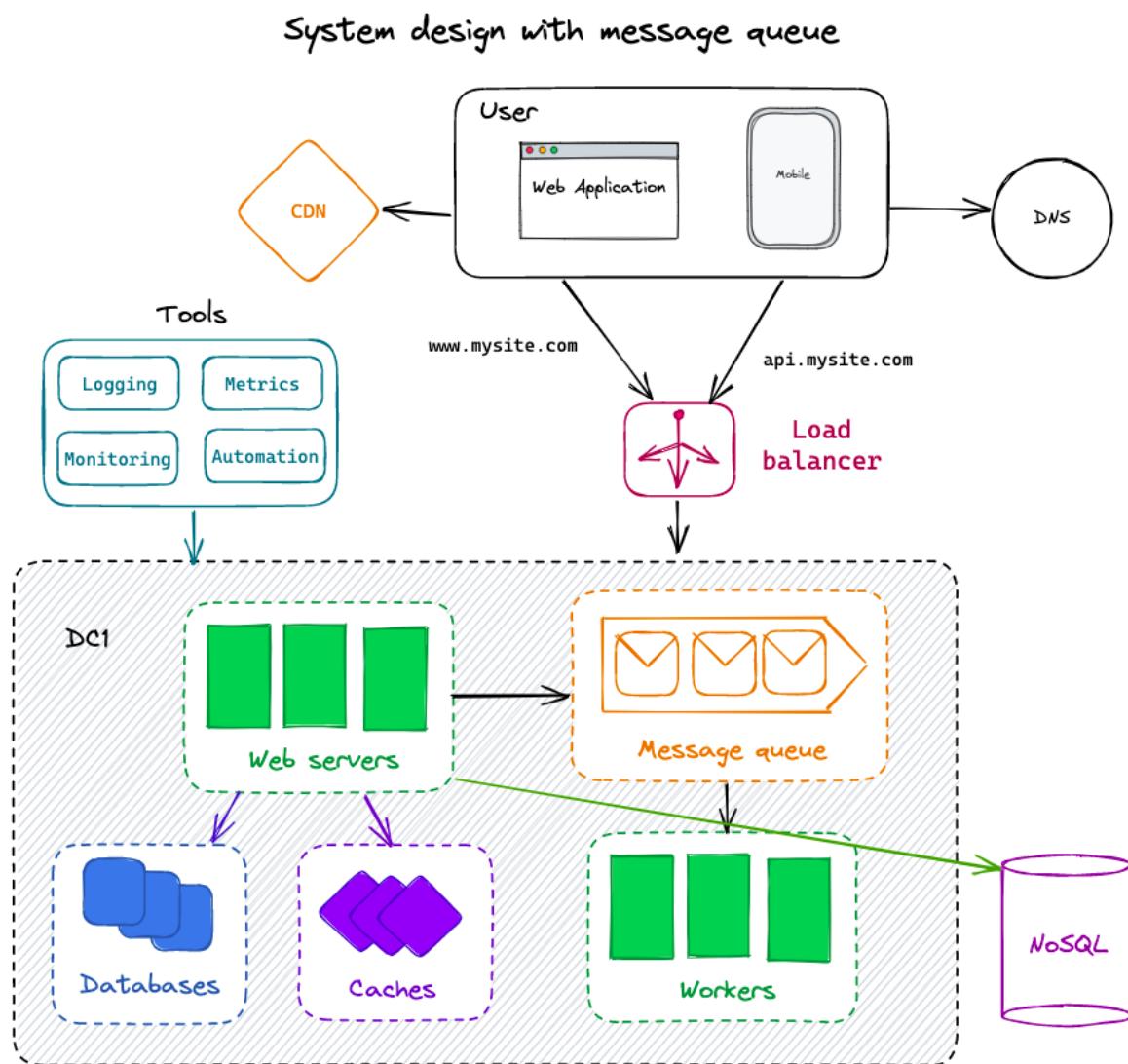
Here's a breakdown of the key components and concepts associated with message queues:

1. **Producer:** The producer is responsible for sending messages to the message queue. In your scenario, the "server" could be acting as the producer, generating messages or data that need to be processed or consumed by another part of the system.
2. **Message Queue:** This is the central component that stores and manages the messages. Messages are typically stored in a queue-like fashion, with new messages being added to the end and consumed from the front. The message queue ensures that messages are processed in the order they are received (FIFO - First In, First Out).
3. **Consumer:** The consumer is responsible for retrieving messages from the message queue and processing them. In your scenario, the "consumer" could be a separate service or component that takes the messages generated by the server and performs some action based on those messages.
4. **Asynchronous Communication:** Message queues enable asynchronous communication, meaning that producers and consumers don't need to interact with each other in real-time. Producers can send messages to the queue at their own pace, and consumers can retrieve and process them at their own pace. This decoupling of components can improve system scalability and robustness.
5. **Message Persistence:** Many message queues offer message persistence, which means that messages are stored even if the system or components fail. This ensures that messages are not lost and can be processed once the system or component recovers.
6. **Message Broker:** In some cases, a message queue system is implemented using a message broker, which acts as an intermediary responsible for routing messages between producers and consumers. Popular message brokers include Apache Kafka, RabbitMQ, and Apache ActiveMQ.
7. **Message Formats:** Messages sent via a message queue can be in various formats, such as JSON, XML, or binary data, depending on the requirements of the system.

Overall, message queues are a fundamental building block in many distributed systems, helping to decouple different components, improve system resilience, and enable efficient communication between services or parts of a system. They are widely used in scenarios like job queues, event-driven architectures, and microservices-based applications.

## System design with message queue

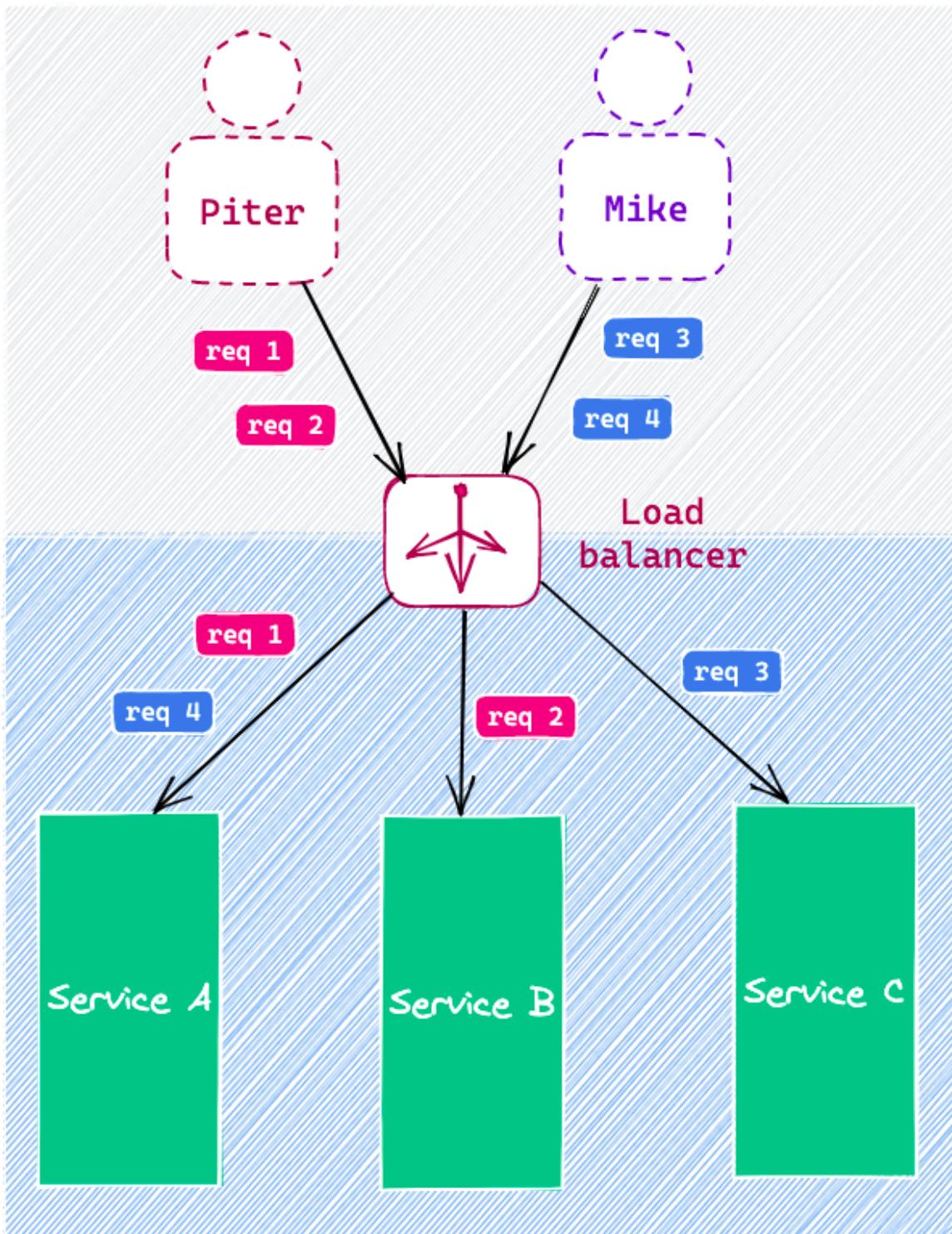
### System design with message queue



Round robin

Round robin

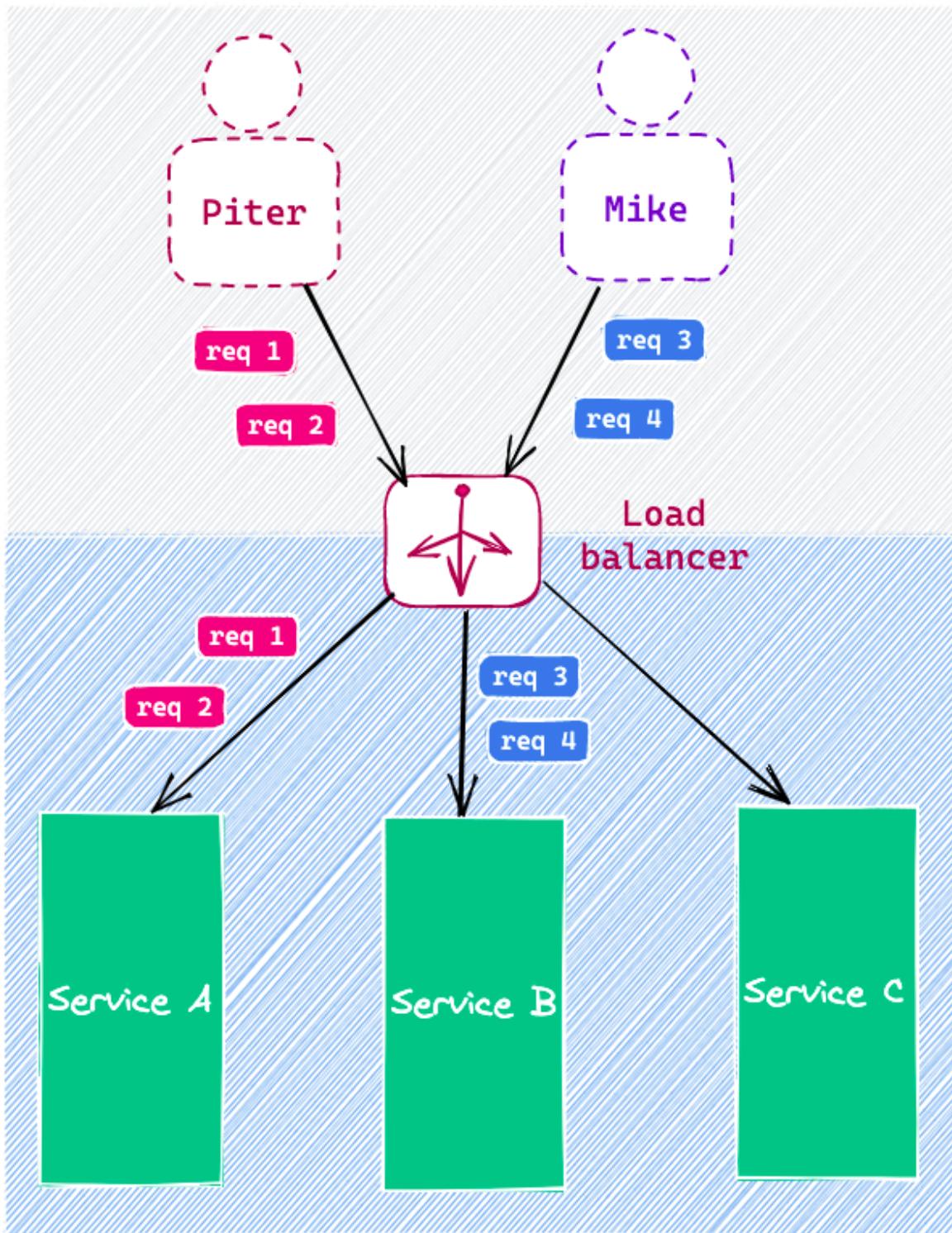
## Round Robin



Sticky round robin

Sticky round robin

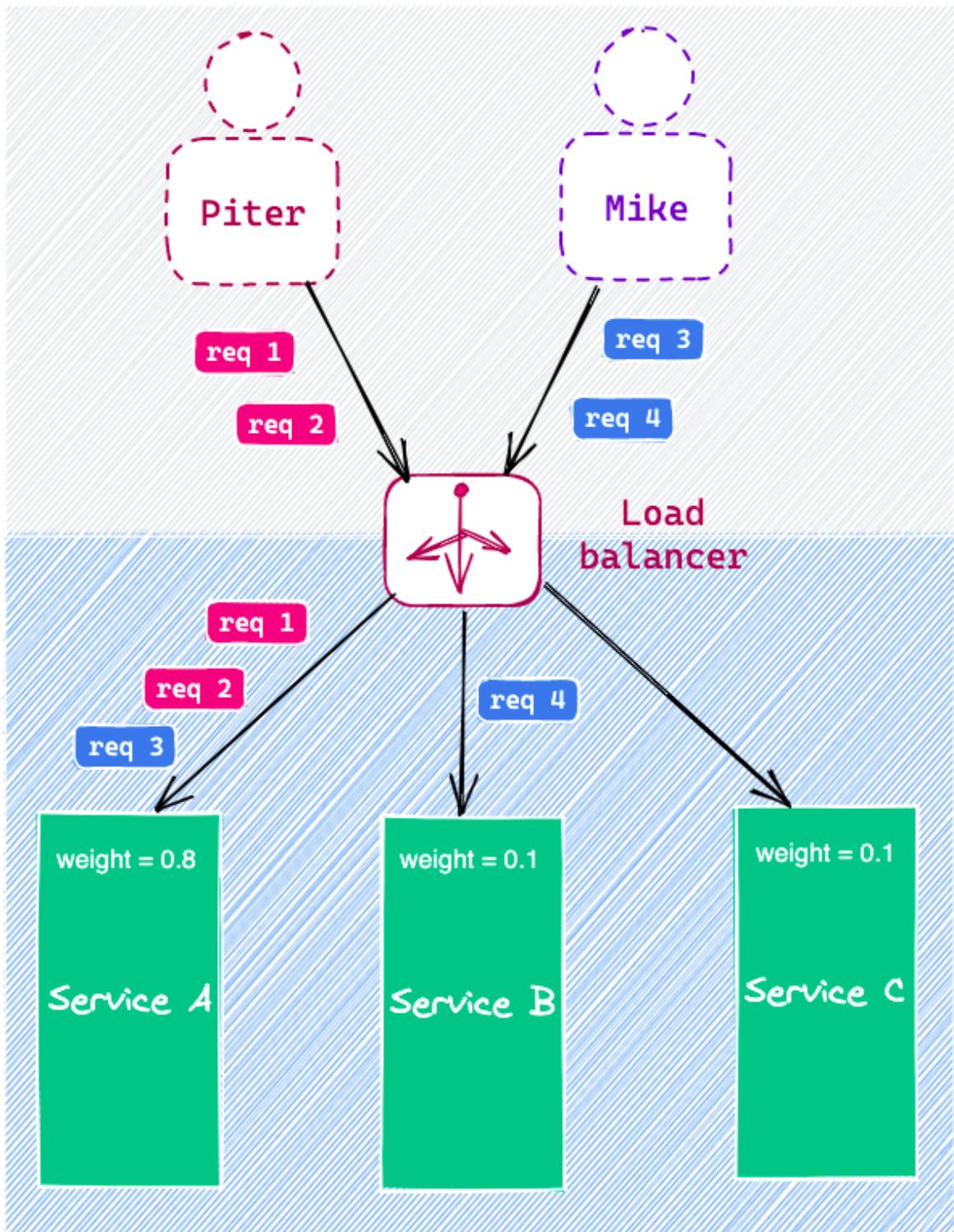
## Sticky Round Robin



Weighted round robin

Weighted round robin

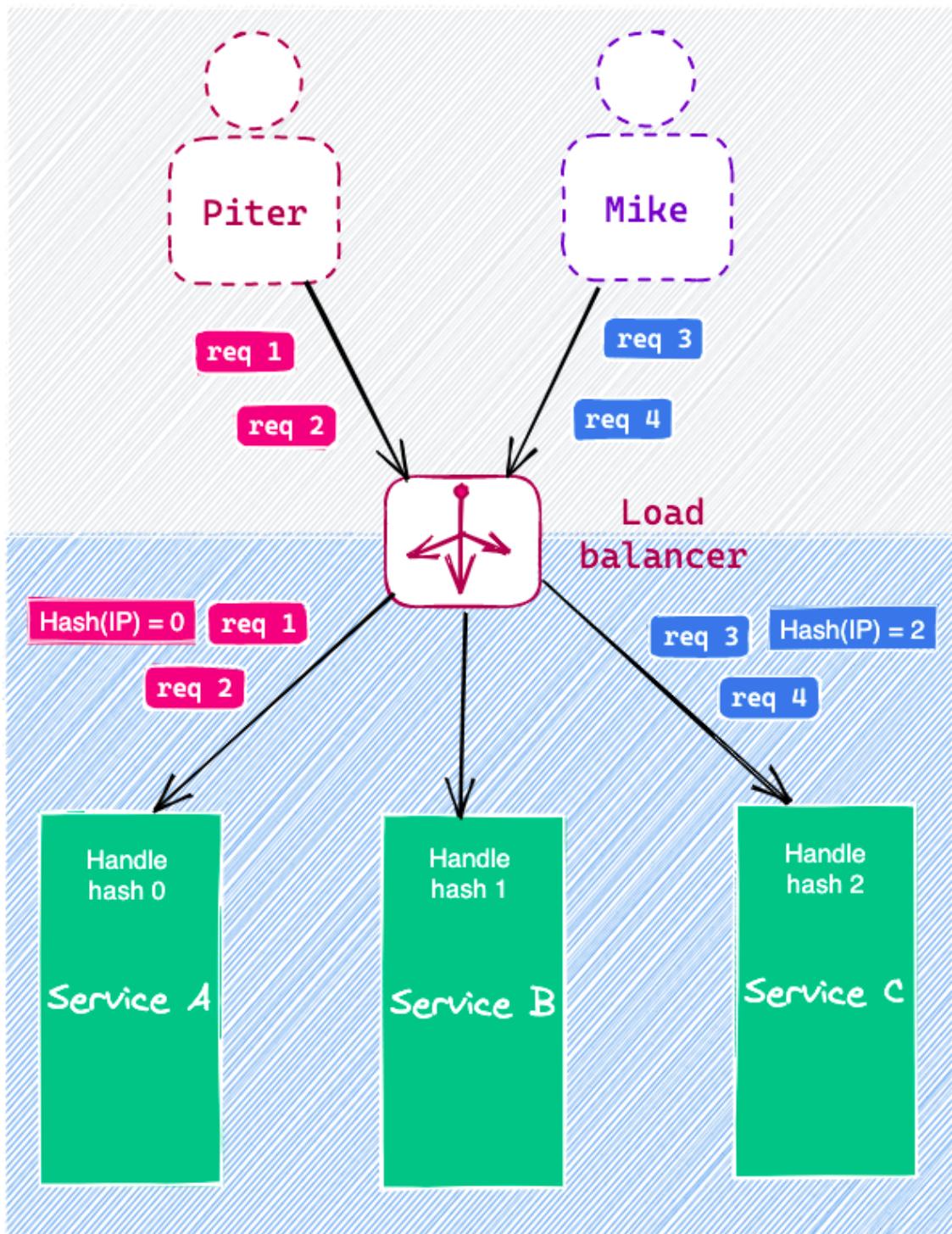
## Weighted Round Robin



IP/URL hash

IP/URL hash

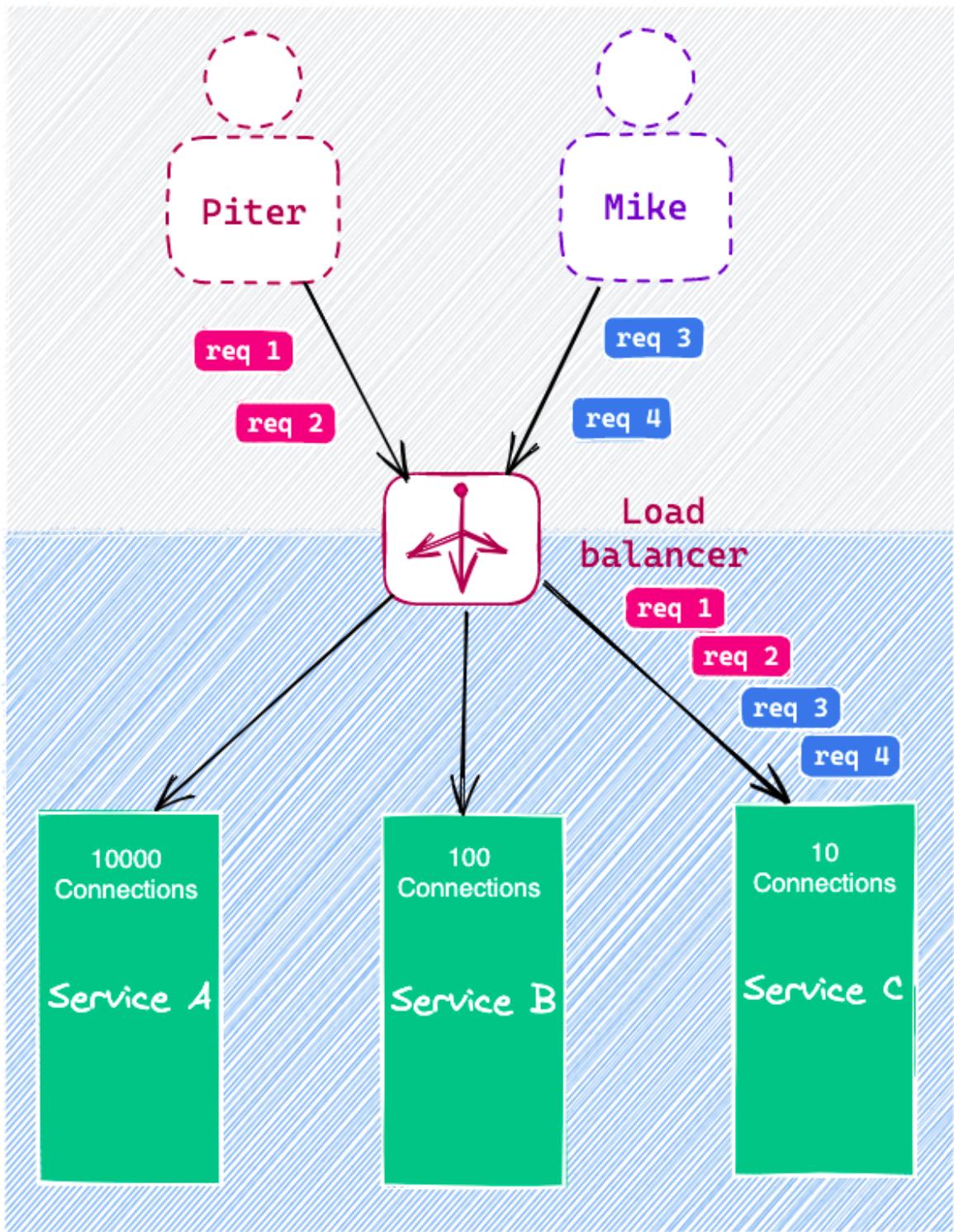
## Ip/Url Hash



Least connections

Least connections

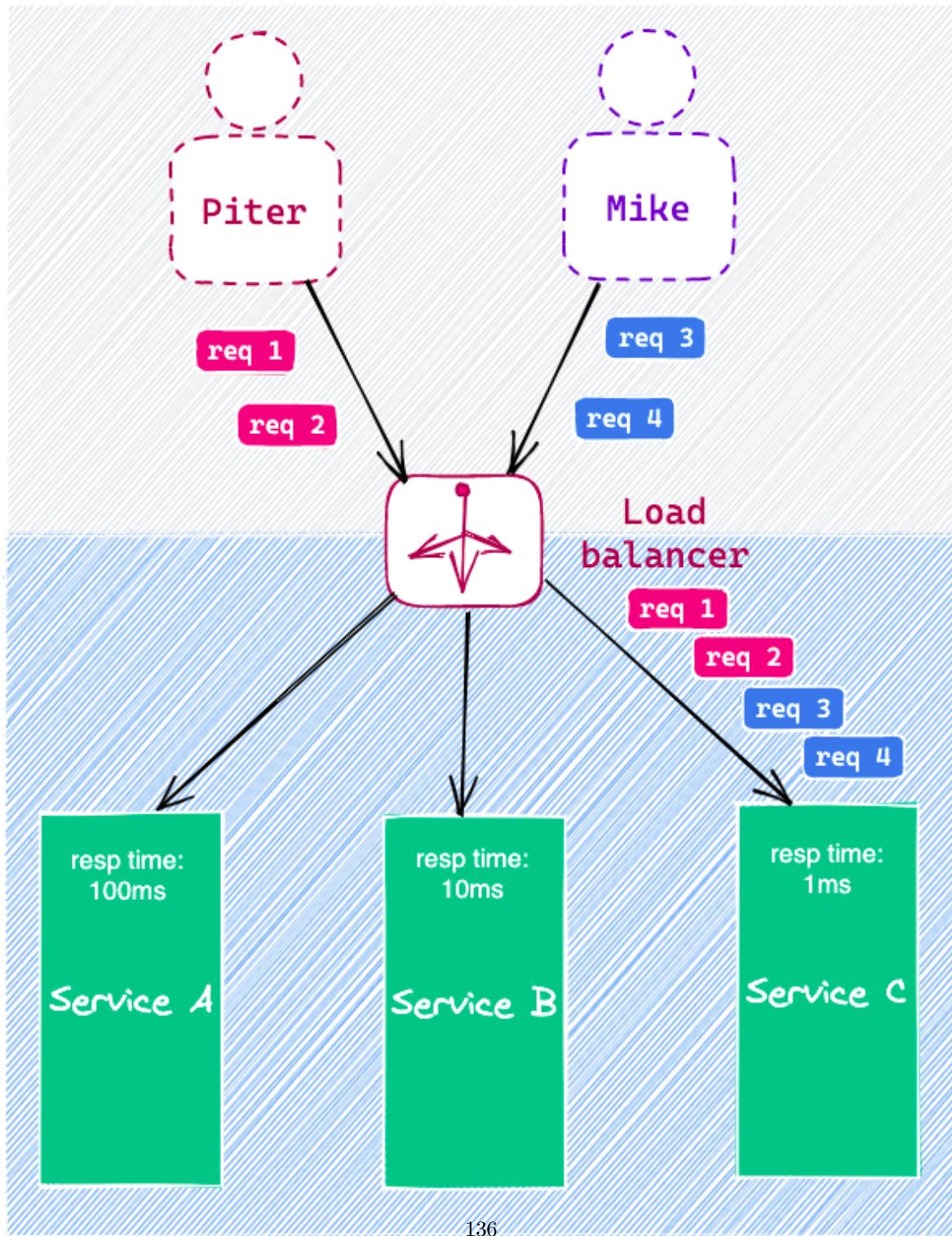
## Least Connections



Least time

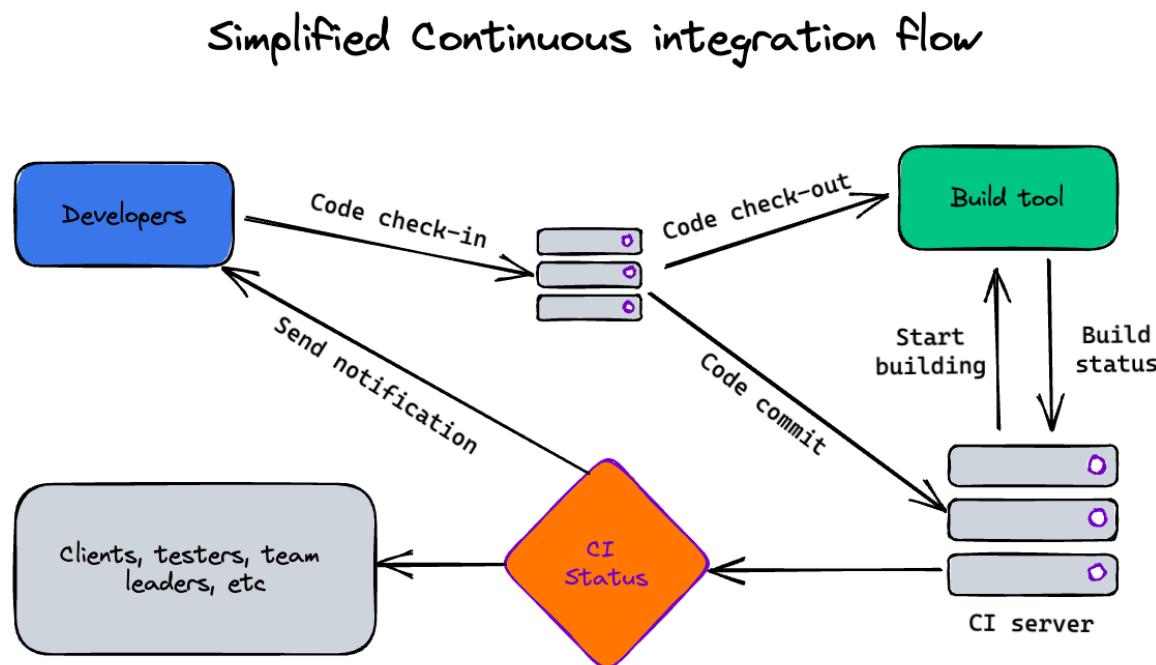
Least time

## Least time



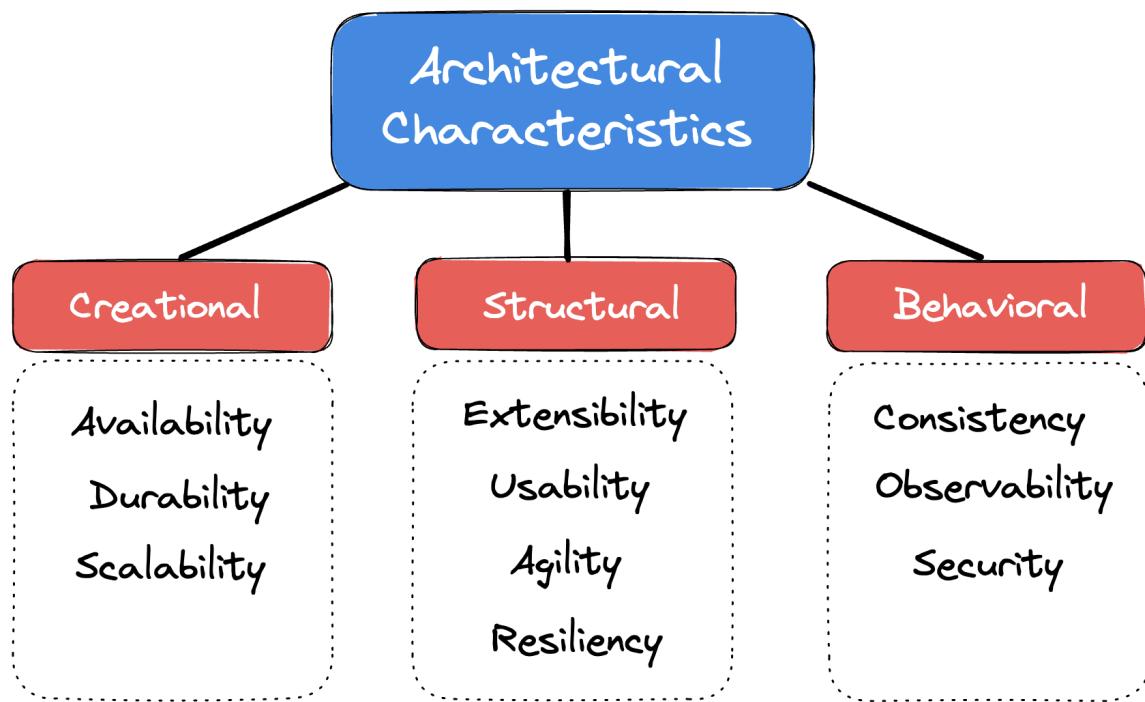
Simplified Continuous integration flow

Simplified Continuous integration flow



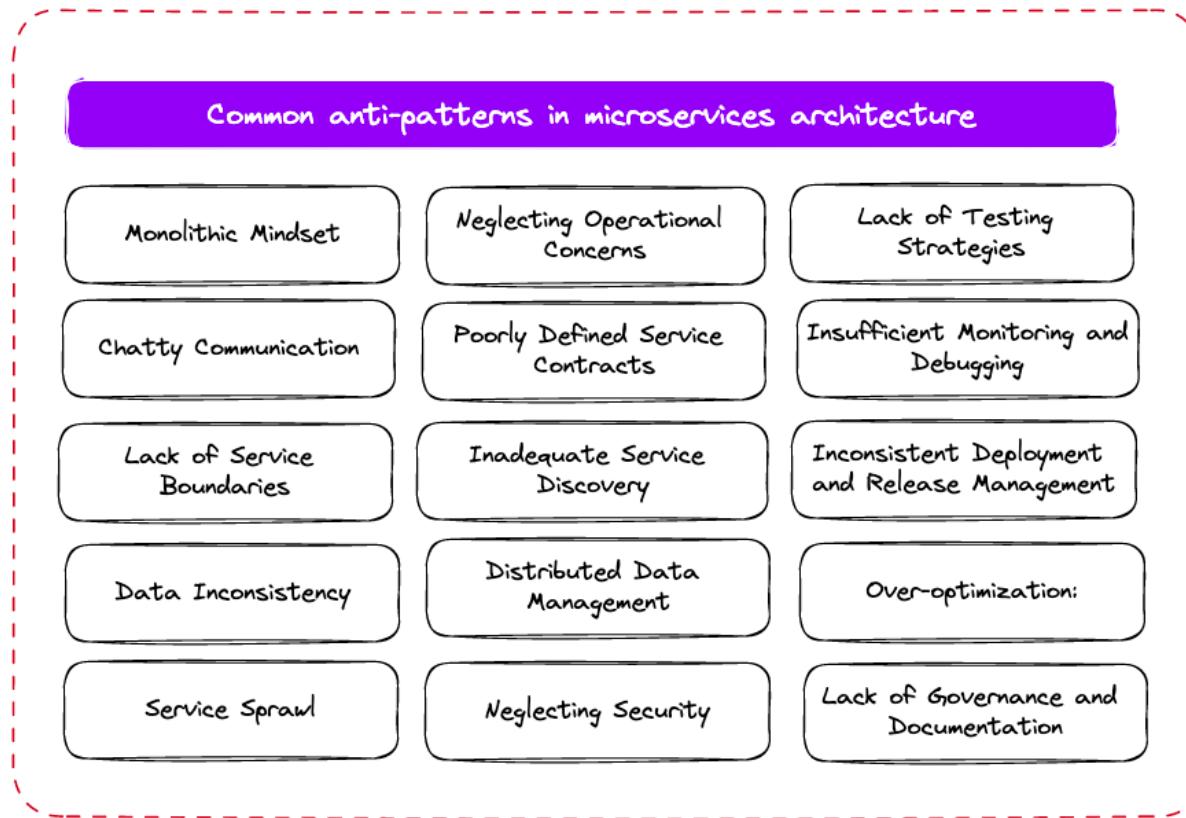
Architectural characteristics

Architectural characteristics



## Common anti-patterns in microservices architecture

### Common anti-patterns in microservices architecture



1. **Monolithic Mindset:** Treating microservices as miniature monoliths, where services are tightly coupled and communicate extensively, negating the benefits of independent, autonomous services.
2. **Chatty Communication:** Excessive inter-service communication, leading to network congestion, increased latency, and reduced performance. Services should communicate judiciously to minimize network overhead.
3. **Lack of Service Boundaries:** Poorly defined or ambiguous service boundaries, resulting in overlapping responsibilities and difficulty in managing and scaling services independently.
4. **Data Inconsistency:** Sharing databases or data stores across multiple services without proper coordination, leading to data inconsistency, integrity issues, and coupling between services.
5. **Service Sprawl:** Creating an excessive number of microservices without a clear reason or justification, resulting in complexity, operational overhead, and difficulty in managing and coordinating the services effectively.
6. **Neglecting Operational Concerns:** Ignoring operational aspects such as monitoring, logging, error handling, and deployment automation, which can hinder observability, maintainability, and reliability of the microservices.
7. **Poorly Defined Service Contracts:** Lack of well-defined and stable service contracts or API specifications, making it difficult for consumers to integrate with the services and increasing the likelihood of breaking changes.

8. **Inadequate Service Discovery:** Relying on hard-coded service endpoints or manual configuration instead of implementing a robust service discovery mechanism, causing challenges in service deployment, scaling, and fault tolerance.
9. **Distributed Data Management:** Attempting to maintain strong consistency across multiple services, leading to complex distributed transactions and increased chances of failures and performance degradation.
10. **Neglecting Security:** Failing to implement proper authentication, authorization, and encryption mechanisms, exposing services to security vulnerabilities and unauthorized access.
11. **Lack of Testing Strategies:** Insufficient testing strategies for end-to-end integration testing, service contracts, fault tolerance, and failure scenarios, resulting in higher chances of undetected issues in the microservices ecosystem.
12. **Insufficient Monitoring and Debugging:** Inadequate monitoring and debugging capabilities across services, making it challenging to identify and troubleshoot issues promptly.
13. **Inconsistent Deployment and Release Management:** Lack of standardized deployment processes and release management practices across services, leading to inconsistencies, deployment errors, and versioning issues.
14. **Over-optimization:** Premature optimization of individual microservices without considering the overall system performance and bottlenecks, leading to suboptimal performance at the system level.
15. **Lack of Governance and Documentation:** Insufficient governance processes, documentation, and guidelines for designing, implementing, and evolving microservices, resulting in inconsistencies and difficulty in maintaining the architecture over time.

**What does API gateway do?**

**What does API gateway do?**

API Gateway serves as a centralized entry point for client applications to access multiple backend APIs. It offers several key functions, including:

- **Routing:** Directs incoming requests to the appropriate backend service/API based on the requested resource path or other routing rules.
- **Authentication:** Validates and authorizes client requests using various authentication mechanisms such as API keys, JWT tokens, or other credentials.
- **Authorization:** Determines if a client has permission to access a specific resource or perform a certain action.
- **Rate Limiting:** Controls the number of requests allowed from a client within a certain timeframe, preventing abuse or excessive usage.
- **Caching:** Stores frequently accessed API responses to reduce backend load and improve response times.
- **Transformation:** Modifies request/response payloads to match the requirements of backend services or client applications.
- **Monitoring:** Tracks and logs API usage and performance metrics to monitor overall health and identify potential issues.

Here are some examples to illustrate these functionalities:

- **Routing:** An API Gateway receives an HTTP request and routes it to the appropriate backend service based on the requested resource path.

- **Authentication:** An API Gateway verifies the identity of a client by validating API keys, JWT tokens, or other authentication credentials.
- **Authorization:** An API Gateway checks if a client has the necessary permissions to access a specific API endpoint or perform a particular action.
- **Rate Limiting:** An API Gateway restricts the number of requests a client can make within a given time period (e.g., 100 requests per minute).
- **Caching:** An API Gateway caches the response from a backend API and serves it directly to future identical requests, reducing backend load.
- **Transformation:** An API Gateway modifies the request or response payloads to match the format required by the backend service or client application.
- **Monitoring:** An API Gateway tracks API usage metrics such as request count, latency, and error rates to monitor performance and health.

### **How is NoSQL database different from SQL databases?**

### **How is NoSQL database different from SQL databases?**

NoSQL (which stands for "Not Only SQL") databases are a class of database management systems that differ from traditional SQL (Structured Query Language) databases in several ways. These differences primarily stem from the fact that NoSQL databases are designed to handle unstructured or semi-structured data and are often used in scenarios where scalability, flexibility, and speed are critical. Here are some key differences between NoSQL and SQL databases:

#### 1. Data Model:

- SQL databases are relational databases, which means they store data in structured tables with predefined schemas. Data is organized into rows and columns.
- NoSQL databases use various data models, such as document-based, key-value, column-family, or graph-based. These models allow for more flexible and dynamic data structures.

#### 2. Schema:

- SQL databases require a rigid schema that defines the structure of the data before it is inserted. Any changes to the schema typically require altering tables and can be time-consuming.
- NoSQL databases are schema-less or have a dynamic schema, meaning you can add new fields to documents or records without altering the entire database structure.

#### 3. Query Language:

- SQL databases use the SQL query language for data retrieval and manipulation. SQL provides a powerful and standardized way to interact with relational data.
- NoSQL databases often have their query languages or APIs specific to their data model. Queries may not be as expressive as SQL but are tailored to the database's particular data structure.

#### 4. Scalability:

- SQL databases are typically scaled vertically by adding more resources (CPU, RAM, etc.) to a single server. This can be expensive and has limitations in terms of scalability.
- NoSQL databases are designed for horizontal scalability. You can distribute data across multiple servers or nodes to handle large amounts of data and high traffic loads more easily.

#### 5. ACID vs. BASE:

- SQL databases follow the ACID (Atomicity, Consistency, Isolation, Durability) properties, ensuring strict data consistency and reliability.

- NoSQL databases follow the BASE (Basically Available, Soft state, Eventually consistent) model, which prioritizes availability and partition tolerance over strict consistency. This makes them suitable for scenarios where eventual consistency is acceptable.

#### 6. Use Cases:

- SQL databases are well-suited for applications with structured and well-defined data, such as financial systems, ERP (Enterprise Resource Planning) software, and traditional relational data storage.
- NoSQL databases are often chosen for applications with large amounts of semi-structured or unstructured data, such as social media, content management systems, real-time analytics, and IoT (Internet of Things) applications.

- Go back

### **How Would You Design an API for Third-Party Developers?**

#### **How Would You Design an API for Third-Party Developers?**

Designing an API for third-party developers requires careful planning and consideration to ensure it is user-friendly, well-documented, and secure. Here are the key steps and principles to follow when designing an API for third-party developers:

##### **1. Define Clear Objectives:**

- Understand the purpose of your API and what problem it solves.
- Identify the target audience and use cases for third-party developers.

##### **2. RESTful Design Principles:**

- Design your API using RESTful principles, which include using HTTP methods (GET, POST, PUT, DELETE) to perform actions on resources.
- Use meaningful resource URIs that are intuitive to developers.
- Implement statelessness, where each API request should contain all the information needed to fulfill it.

##### **3. Authentication and Authorization:**

- Implement robust authentication and authorization mechanisms to ensure only authorized users can access the API.
- Use industry-standard authentication protocols like OAuth 2.0 or API keys.

##### **4. Versioning:**

- Include versioning in your API design to ensure backward compatibility as you make updates in the future. You can include version numbers in the URI (e.g., /v1/resource) or use headers.

##### **5. Use Standard HTTP Status Codes:**

- Respond to API requests with standard HTTP status codes to indicate the success or failure of the request (e.g., 200 for success, 401 for unauthorized).

##### **6. Error Handling:**

- Provide detailed error messages in the response to help developers diagnose issues.

- Use consistent error formats and codes.

## 7. Rate Limiting:

- Implement rate limiting to prevent abuse of your API and to ensure fair usage for all developers.

## 8. Data Formats:

- Use standard data formats like JSON or XML for data exchange.
- Clearly define the data structure and provide examples in your documentation.

## 9. Documentation:

- Create comprehensive and easy-to-understand documentation. Include:
  - API endpoints and methods
  - Request and response examples
  - Authentication instructions
  - Rate-limiting details
  - Error codes and explanations
  - Use cases and sample code
- Consider using tools like Swagger, API Blueprint, or OpenAPI for documenting your API.

## 10. Developer Portal:

- Provide a developer portal or website where third-party developers can access documentation, register for API keys, and find other resources.

## 11. Testing and Sandbox Environment:

- Offer a sandbox environment where developers can test their integrations without affecting production data.

## 12. Security:

- Implement security best practices, including input validation, encryption, and protection against common vulnerabilities like SQL injection and CSRF attacks.

## 13. Monitoring and Analytics:

- Implement monitoring tools to track API usage, performance, and errors.
- Use analytics to gain insights into how developers are using your API and what can be improved.

## 14. Feedback and Support:

- Provide channels for developers to give feedback and request support.
- Actively engage with the developer community to address concerns and improve the API.

## 15. Developer Onboarding:

- Simplify the process of getting started with your API by offering clear onboarding steps and tutorials.

## 16. Change Management:

- Communicate changes to the API, such as deprecations or updates, well in advance.
- Provide migration guides to help developers transition to new versions or features.

## 17. Legal and Terms of Use:

- Clearly outline the terms of use and any legal requirements for using your API.

#### **18. Scalability and Performance:**

- Ensure that your API infrastructure can handle a growing number of requests and maintain good performance under load.

#### **19. Feedback Loop:**

- Continuously gather feedback from third-party developers and iterate on your API based on their needs and suggestions.

#### **20. Compliance:**

- Ensure your API complies with industry regulations and standards, such as GDPR for data protection or HIPAA for healthcare data.

Remember that the key to a successful API for third-party developers is to make it as intuitive, reliable, and well-documented as possible. Regularly update and improve the API based on developer feedback and evolving industry standards.

- Go back

### **How to optimize work of a web application using caching?**

### **How to optimize work of a web application using caching?**

Optimizing a web application using caching is a crucial technique to improve performance, reduce server load, and enhance user experience. Caching involves storing copies of files or data in a location that is closer to the end-users, allowing quicker access without fetching the data from the original source every time. Here are some strategies to optimize your web application using caching:

#### **1. Browser Caching:**

- Leverage browser caching by setting appropriate cache headers. This instructs the user's browser to cache certain resources locally.
- Set an expiration date or a maximum age in the HTTP headers for static resources (images, CSS, JavaScript). Tools like mod\_expires for Apache or ngx\_http\_headers\_module for Nginx can help with this.

#### **2. Content Delivery Network (CDN):**

- Use a CDN service to cache and deliver your static assets. CDNs have servers located worldwide, reducing latency by serving content from a server geographically closer to the user.

#### **3. Server-Side Caching:**

- Implement server-side caching for dynamic content. Popular caching mechanisms include:
  - **Page Caching:** Cache the entire HTML output of a page and serve it to subsequent users without hitting the server. Tools like Varnish can help.
  - **Object Caching:** Cache specific objects or database queries so they can be retrieved quickly without recalculating them.
  - **Opcode Caching:** Use PHP opcode caching extensions like OPCache to store compiled PHP code, reducing the overhead of parsing and compiling on each request.

#### **4. Database Caching:**

- Cache database queries and responses. Use caching mechanisms provided by your database system or an in-memory data store like Redis or Memcached to store frequently accessed data.
- Implement query caching at the database level to store results of frequent queries.

#### **5. API Response Caching:**

- If your application relies on external APIs, cache the responses from these APIs to reduce response time and prevent unnecessary API calls.

#### **6. Fragment Caching:**

- Cache specific parts of a web page (fragments) that are resource-intensive to generate. This is particularly useful for content management systems where certain parts of a page change infrequently.

#### **7. Cache Invalidation:**

- Implement cache invalidation strategies to ensure that cached data is updated when the underlying data changes. Techniques like cache timeouts or versioned URLs can help in this regard.

#### **8. Mobile Caching:**

- Optimize caching for mobile devices, as they often have slower network connections. Serve appropriately sized images and leverage caching for mobile-specific resources.

#### **9. Monitoring and Tuning:**

- Regularly monitor cache performance using tools and logs. Adjust cache expiration times and cache policies based on usage patterns.
- Use tools like Google PageSpeed Insights or GTmetrix to identify caching opportunities and performance bottlenecks.

#### **10. HTTPS and Cache Control:**

- When implementing caching, ensure that your HTTPS configuration is correct. Caching might behave differently for encrypted (HTTPS) and unencrypted (HTTP) connections.

By combining these strategies and tailoring them to your specific web application, you can significantly optimize its performance and improve user experience. Remember that proper testing and monitoring are essential to fine-tune your caching strategies for optimal results.

### **10 steps for system design**

#### **10 steps for system design**

Keep in mind that the specifics can vary based on the type of system you're designing (software, hardware, IT, etc.).

1. **Define the Problem:** Clearly state the problem your system is going to solve. Understand the requirements and constraints.
2. **Gather Requirements:** Collect detailed requirements from stakeholders. Understand both functional (what the system should do) and non-functional requirements (performance, reliability, security, etc.).
3. **Research and Brainstorm:** Investigate existing solutions and brainstorm potential approaches. Consider technologies, frameworks, and methodologies that might be suitable.
4. **Create System Architecture:** Develop a high-level architecture of the system. Identify major components, their relationships, and how data will flow between them.
5. **Detailed Design:** Dive into the details of each component. Design algorithms, data structures, interfaces, and databases. Choose appropriate technologies and tools.
6. **Prototyping:** Create a prototype or a proof of concept to validate critical aspects of the design. It helps in identifying potential issues early on.
7. **Testing:** Develop a testing plan. This includes unit testing for individual components and integration testing to ensure different components work together as intended.
8. **Implementation:** Write code and build the system according to the design. Follow best practices, coding standards, and version control procedures.
9. **Deployment:** Deploy the system in the target environment. This might involve setting up servers, configuring networks, and ensuring all dependencies are met.
10. **Maintenance and Updates:** Once the system is live, there will be a need for ongoing maintenance. This includes bug fixes, updates for new requirements, and possibly scaling the system as user demands grow.

These steps are iterative and often involve revisiting previous stages as new information comes to light or as the project progresses. Effective communication with stakeholders and team members is crucial throughout the process.

## testing

### What is stress test?

In software engineering, a stress test refers to a type of performance testing that evaluates the behavior and stability of a software system under extreme or stressful conditions. The purpose of a stress test is to assess the system's ability to handle high loads, heavy traffic, or resource-intensive scenarios.

During a stress test, the software system is intentionally pushed to its limits to observe how it performs under extreme conditions. The test focuses on identifying performance bottlenecks, uncovering weaknesses, and validating the system's response and recovery capabilities when subjected to stress factors such as high user concurrency, large data volumes, or excessive processing demands.

The stress test simulates conditions that go beyond the typical usage patterns and aims to answer questions like:

1. How does the system behave when a significantly higher number of users access it simultaneously?
2. Does the system respond within acceptable time frames under high load?

3. Does the system handle sudden spikes in traffic without crashing or experiencing performance degradation?
4. Can the system recover gracefully from resource exhaustion or failures?

By conducting stress tests, software engineers can proactively identify and address performance issues, scalability limitations, and architectural flaws before they impact the system's stability, user experience, or overall business operations. The insights gained from stress testing can inform optimization efforts, capacity planning, and infrastructure adjustments to ensure the software system meets performance requirements and can withstand stressful conditions in production environments.

#### What types of software testing do you know?

## What types of software testing do you know?

There are several types of software testing, each serving a specific purpose in the software development lifecycle. Here are some common types of software testing:

1. **Unit Testing:** This type of testing involves testing individual components or modules of a software application to ensure they work as intended.
2. **Integration Testing:** Integration testing is the process of testing the interaction between different components or systems to verify that they work together as expected.
3. **Functional Testing:** Functional testing checks if the software functions as expected and according to the requirements. It focuses on the software's behavior and ensures that it performs its functions correctly.
4. **Regression Testing:** Regression testing involves re-running functional and non-functional tests to ensure that previously developed and tested software still works after a change.
5. **Performance Testing:** Performance testing evaluates the software's performance, responsiveness, and stability under a particular workload. It helps identify and eliminate performance bottlenecks.
6. **Load Testing:** Load testing is a subset of performance testing that assesses the software's ability to handle a specific load, typically focusing on a large number of simultaneous users or transactions.
7. **Stress Testing:** Stress testing evaluates the software's stability and reliability under extreme conditions, such as high loads, limited resources, or unusual data volumes.
8. **Security Testing:** Security testing identifies vulnerabilities in the software to ensure that data and resources are protected from unauthorized access, attacks, and other security threats.
9. **Usability Testing:** Usability testing assesses the software's user interface and user experience to ensure that it is intuitive, easy to use, and meets users' needs.
10. **Compatibility Testing:** Compatibility testing checks if the software is compatible with different operating systems, browsers, devices, and network environments.
11. **Acceptance Testing:** Acceptance testing is performed to determine whether the software meets the acceptance criteria set by the client or end users. It often includes alpha and beta testing phases.
12. **Exploratory Testing:** Exploratory testing involves simultaneous learning, test design, and test execution. Testers explore the software, learn about its behavior, and design and execute tests dynamically.
13. **Ad Hoc Testing:** Ad hoc testing is informal testing without any specific test design or documentation. Testers spontaneously test the software based on their knowledge and experience.

These are some of the most common types of software testing, and they can be further customized or combined to meet the specific needs of a software development project.

## training

### 10001st prime

### 10001st prime

```
function nthPrime(n: number): number {
    const primes: number[] = [2];
    let num: number = 3;

    while (primes.length < n) {
        const sqrtNum: number = Math.sqrt(num);
        let isPrime: boolean = true;

        for (let i: number = 0; i < primes.length && primes[i] <= sqrtNum; i++) {
            if (num % primes[i] === 0) {
                isPrime = false;
                break;
            }
        }

        if (isPrime) {
            primes.push(num);
        }

        num += 2;
    }

    return primes[primes.length - 1];
}
```

#### Solution:

##### 1. Initialization:

- Initialize an array **primes** with the first prime number, 2.
- Initialize a variable **num** to 3. This variable represents the number to be checked for primality.

##### 2. Loop until **primes** array has **n** elements:

- Use a **while** loop that continues until the length of the **primes** array is less than **n**.
- Within the loop:
  - Calculate the square root of **num** and store it in **sqrtNum**.
  - Initialize a boolean variable **isPrime** to **true**.
  - Iterate through the **primes** array up to the square root of **num**:
    - \* If **num** is divisible by any number in the **primes** array, set **isPrime** to **false** and break the loop.
    - If **isPrime** is **true** after the loop, push **num** into the **primes** array.
  - Increment **num** by 2 to check only odd numbers for efficiency, as even numbers greater than 2 are not prime.

##### 3. Return the Nth Prime Number:

- Once the loop finishes and the `primes` array has `n` elements, return the last element of the `primes` array, which represents the Nth prime number.

### 3Sum Closest

## 3Sum Closest

```
function threeSumClosest(nums: number[], target: number): number {
    nums.sort((a, b) => a - b);

    let closestSum = nums[0] + nums[1] + nums[2];

    for (let i = 0; i < nums.length - 2; i++) {
        let left = i + 1;
        let right = nums.length - 1;

        while (left < right) {
            const sum = nums[i] + nums[left] + nums[right];

            if (Math.abs(sum - target) < Math.abs(closestSum - target)) {
                closestSum = sum;
            }

            if (sum < target) {
                left++;
            } else if (sum > target) {
                right--;
            } else {
                return sum;
            }
        }
    }

    return closestSum;
};
```

### Solution:

The `threeSumClosest` function aims to find the sum of three numbers in an array (`nums`) that is closest to a target sum (`target`). Below are the step-by-step explanations of the code:

Step	Description
1	The function <code>threeSumClosest</code> is defined, taking two parameters: <code>nums</code> (an array of numbers) and <code>target</code> (the target sum).
2	The <code>nums</code> array is sorted in ascending order using the <code>sort</code> method. This ensures that the array is arranged numerically.

Step	Description
3	A variable named <code>closestSum</code> is initialized with the sum of the first three numbers in the sorted array ( <code>nums[0] + nums[1] + nums[2]</code> ). This initial sum serves as the closest sum.
4	The code enters a <code>for</code> loop that iterates over the array <code>nums</code> from the first element to the third-to-last element ( <code>nums.length - 2</code> ).
5	Inside the <code>for</code> loop, two pointers named <code>left</code> and <code>right</code> are initialized. <code>left</code> is set to <code>i + 1</code> (the next index after <code>i</code> ), and <code>right</code> is set to the last index of the array ( <code>nums.length - 1</code> ).
6	A <code>while</code> loop is used to iterate as long as the <code>left</code> pointer is less than the <code>right</code> pointer.
7	Inside the <code>while</code> loop, the code calculates the sum of three elements: <code>nums[i]</code> , <code>nums[left]</code> , and <code>nums[right]</code> , and stores it in a variable named <code>sum</code> .
8	The code checks if the absolute difference between <code>sum</code> and <code>target</code> is less than the absolute difference between <code>closestSum</code> and <code>target</code> . If it is, then the <code>closestSum</code> is updated with the new <code>sum</code> . This ensures that we keep track of the sum closest to the target.
9	The code then checks three conditions:
	<ul style="list-style-type: none"> <li>• If <code>sum</code> is less than <code>target</code>, it means we need to increase the sum, so the <code>left</code> pointer is incremented by 1.</li> <li>• If <code>sum</code> is greater than <code>target</code>, it means we need to decrease the sum, so the <code>right</code> pointer is decremented by 1.</li> <li>• If <code>sum</code> is equal to <code>target</code>, it means we have found the exact sum we were looking for, so it is returned immediately.</li> </ul>
10	After the <code>while</code> loop ends, the <code>for</code> loop continues to the next iteration, and the process is repeated with a new <code>i</code> value.
11	Once all iterations of the <code>for</code> loop are complete, the function returns the <code>closestSum</code> , which represents the sum closest to the target.

### Techniques used:

The following techniques are utilized within the code:

1. Sorting an array using the `sort` method.
2. Using two pointers (`left` and `right`) to traverse the array.
3. Calculating the sum of three numbers.
4. Using a `for` loop to iterate over an array.
5. Using a `while` loop for

## 3Sum

### 3Sum

```
function threeSum(nums: number[]): number[][] {
    nums.sort((a, b) => a - b);

    const triplets = [];

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

        if (i > 0 && nums[i] === nums[i - 1]) {
            continue;
        }

        let j = i + 1;
        let k = nums.length - 1;

        while (j < k) {
            if (nums[i] + nums[j] + nums[k] === 0) {
                triplets.push([nums[i], nums[j], nums[k]]);
                j++;
                while (j < k && nums[j] === nums[j - 1]) {
                    j++;
                }
            } else if (nums[i] + nums[j] + nums[k] < 0) {
                j++;
            } else {
                k--;
            }
        }
    }
    return triplets;
};
```

**Solution:** Below are the step-by-step explanations of the code:

Step	Description
1	The function <code>threeSum</code> is defined, taking one parameter: <code>nums</code> (an array of numbers).
2	The <code>nums</code> array is sorted in ascending order using the <code>sort</code> method. This ensures that the array is arranged numerically.
3	An empty array named <code>triplets</code> is initialized to store the found triplets.
4	The code enters a <code>for</code> loop that iterates over the array <code>nums</code> from the first element to the last element.

Step	Description
5	Inside the <code>for</code> loop, the code first checks if the current element is a duplicate by comparing it with the previous element. If they are the same, the loop skips to the next iteration using the <code>continue</code> statement. This ensures that duplicate triplets are avoided.
6	Two pointers, <code>j</code> and <code>k</code> , are initialized. <code>j</code> is set to <code>i + 1</code> (the next index after <code>i</code> ), and <code>k</code> is set to the last index of the array ( <code>nums.length - 1</code> ).
7	A <code>while</code> loop is used to iterate as long as <code>j</code> is less than <code>k</code> .
8	Inside the <code>while</code> loop, the code checks three conditions:
	<ul style="list-style-type: none"> <li>If the sum of the current element (<code>nums[i]</code>), <code>nums[j]</code>, and <code>nums[k]</code> is equal to zero, a triplet is found. The triplet is added to the <code>triplets</code> array using the <code>push</code> method. <code>j</code> is incremented by 1 to move to the next element.</li> <li>If the current sum is less than zero, <code>j</code> is incremented by 1 to increase the sum.</li> <li>If the current sum is greater than zero, <code>k</code> is decremented by 1 to decrease the sum.</li> </ul>
9	After the inner <code>while</code> loop ends, the <code>for</code> loop continues to the next iteration, and the process is repeated with a new <code>i</code> value.
10	Once all iterations of the <code>for</code> loop are complete, the function returns the <code>triplets</code> array, which contains all unique triplets that sum up to zero.

### Techniques used:

The following techniques are utilized within the code:

1. Sorting an array using the `sort` method.
2. Using a `for` loop to iterate over an array.
3. Using conditional statements (`if`, `else if`, `else`) to handle different cases.
4. Using a `while` loop for iterative calculations.
5. Skipping to the next iteration using the `continue` statement.
6. Adding elements to an array using the `push` method.

## 4Sum

### 4Sum

```
function fourSum(nums: number[], target: number): number[][] {
  nums.sort((a, b) => a - b);
  const result: number[][] = [];
  const n = nums.length;
```

```

for (let i = 0; i < n - 3; i++) {
    if (i > 0 && nums[i] === nums[i - 1]) {
        continue;
    }

    for (let j = i + 1; j < n - 2; j++) {
        if (j > i + 1 && nums[j] === nums[j - 1]) {
            continue;
        }

        let left = j + 1;
        let right = n - 1;

        while (left < right) {
            const currSum = nums[i] + nums[j] + nums[left] + nums[right];

            if (currSum === target) {
                result.push([nums[i], nums[j], nums[left], nums[right]]);

                while (left < right && nums[left] === nums[left + 1]) {
                    left++;
                }
                while (left < right && nums[right] === nums[right - 1]) {
                    right--;
                }

                left++;
                right--;
            } else if (currSum < target) {
                left++;
            } else {
                right--;
            }
        }
    }
}

return result;
}

```

### Solution:

#### 1. Sort the Input Array:

- Sort the input array `nums` in ascending order. Sorting is a crucial step for efficiently handling duplicates and finding unique quadruplets.

#### 2. Initialize Result Array:

- Initialize an empty array `result` to store the unique quadruplets that sum up to the target.

#### 3. Loop through Unique Pairs of Indices (*i*, *j*):

- Use two nested loops to iterate through pairs of indices (*i* and *j*).

- The outer loop (*i*) goes from 0 to  $n - 4$ , and the inner loop (*j*) goes from *i + 1* to  $n - 3$ .
- The conditions inside the loops skip duplicates to avoid redundant calculations.

#### 4. Two-Pointer Approach for Remaining Elements:

- Use a two-pointer approach with *left* starting from *j + 1* and *right* starting from  $n - 1$ .
- Calculate the current sum *currSum* using elements at indices *i*, *j*, *left*, and *right*.
- If *currSum* is equal to the target, add the quadruplet to the *result* array.
  - Skip duplicates by incrementing *left* and decrementing *right* until reaching distinct elements.
- If *currSum* is less than the target, increment *left*. If greater, decrement *right*.

#### 5. Return the Result Array:

- After completing all iterations, return the *result* array containing unique quadruplets that sum up to the target.

Add binary

Add binary

```
/** 
 * @param {string} a
 * @param {string} b
 * @return {string}
 */
var addBinary = function(a, b) {
  let result = "";
  let i = a.length - 1;
  let j = b.length - 1;
  let carry = 0;
  while (i >= 0 || j >= 0) {
    let sum = carry;
    if (i >= 0) {
      sum += a[i--] - '0';
    }
    if (j >= 0) {
      sum += b[j--] - '0';
    }
    result = sum % 2 + result;
    carry = parseInt(sum / 2);
  }
  if (carry > 0) {
    result = 1 + result;
  }
  return result;
};
```

Time complexity: O(n) Space complexity: O(n)

Apologies for the oversight. Here's the revised explanation of the `addBinary` code, including a Markdown table for better clarity:

### Solution:

The provided code solves the task by performing binary addition on the input strings **a** and **b**. Here's a step-by-step breakdown:

Step	Description
1	The <code>addBinary</code> function is defined, taking two parameters: <b>a</b> (a binary string) and <b>b</b> (another binary string).
2	Several variables are initialized: <ul style="list-style-type: none"><li><code>result</code> is initialized as an empty string to store the result of the binary addition.</li><li><code>i</code> is set to the index of the last character in string <b>a</b>.</li><li><code>j</code> is set to the index of the last character in string <b>b</b>.</li><li><code>carry</code> is initialized as 0 to hold the carry value during addition.</li></ul>
3	A <code>while</code> loop is used to iterate as long as either <code>i</code> or <code>j</code> is greater than or equal to 0.
4	Inside the <code>while</code> loop, the variable <code>sum</code> is initialized with the value of <code>carry</code> . This represents the current sum at each iteration.
5	Conditional statements are used to add the respective binary digits of strings <b>a</b> and <b>b</b> to the <code>sum</code> : <ul style="list-style-type: none"><li>If <code>i</code> is greater than or equal to 0, the current binary digit from <b>a</b> is added to <code>sum</code> by converting it from a character to a number using the expression <code>a[i--] - '0'</code>.</li><li>If <code>j</code> is greater than or equal to 0, the current binary digit from <b>b</b> is added to <code>sum</code> using the same conversion.</li></ul>
6	The current binary digit of the sum, obtained by taking the modulo 2 ( <code>sum % 2</code> ), is concatenated to the left side of the <code>result</code> string.
7	The carry value for the next iteration is updated by dividing the sum by 2 and parsing the integer value using <code>parseInt</code> .
8	If there is a remaining carry after the <code>while</code> loop ends (i.e., <code>carry &gt; 0</code> ), a '1' is concatenated to the left side of the <code>result</code> string.
9	Finally, the <code>result</code> string, which represents the sum of the binary strings <b>a</b> and <b>b</b> , is returned.

### Techniques used:

The following techniques are utilized within the code:

1. Iterating over a string using a `while` loop.
2. Accessing characters in a string using indices.
3. Converting a character to a number using subtraction ('0').

4. Concatenating strings using the `+` operator.
5. Performing arithmetic operations (addition, modulo, division) on numbers.
6. Handling carry values during addition.
7. Checking conditions using `if` statements.
8. Updating variables with post-increment (`i--`) and assignment (`=`) operations.

## Add Strings

### Add Strings

```
function addStrings(num1: string, num2: string): string {

    let carry = 0;
    let firstP = num1.length - 1;
    let secondP = num2.length - 1;
    let res = '';

    while (firstP >= 0 || secondP >= 0) {
        let sum = 0;
        let first = firstP >= 0 ? Number(num1[firstP--]) : 0
        let second = secondP >= 0 ? Number(num2[secondP--]) : 0

        sum += first + second + carry;
        carry = 0;

        if (sum > 9) {
            sum %= 10;
            carry++;
        }
        res = sum + res;
    };

    if (carry > 0) {
        res = carry + res;
    }

    return res;
};
```

**Solution:** Below are the step-by-step explanations of the code:

1. The function `addStrings` is defined, taking two parameters: `num1` (a number string) and `num2` (another number string).
2. Several variables are initialized:
  - `carry` is initialized as 0 to hold the carry value during addition.
  - `firstP` is set to the index of the last character in string `num1`.
  - `secondP` is set to the index of the last character in string `num2`.
  - `res` is initialized as an empty string to store the result of the addition.
3. A `while` loop is used to iterate as long as either `firstP` or `secondP` is greater than or equal to 0.

4. Inside the `while` loop, the variable `sum` is initialized with the sum of the corresponding digits from `num1` and `num2`, along with the carry value.
5. Conditional statements are used to handle cases where either `firstP` or `secondP` is less than 0 (reached the beginning of the string). In such cases, the corresponding digit is considered as 0.
6. The sum is computed by adding `first`, `second`, and `carry`. The `carry` value is reset to 0.
7. If the sum is greater than 9, indicating a carry, the sum is updated to the remainder of dividing by 10, and the `carry` value is incremented.
8. The current digit of the sum, obtained after carry adjustment, is concatenated to the left side of the `res` string.
9. After the `while` loop ends, if there is a remaining carry (i.e., `carry > 0`), it is concatenated to the left side of the `res` string.
10. Finally, the `res` string, which represents the sum of the number strings `num1` and `num2`, is returned.

#### Techniques used:

The following techniques are utilized within the code:

1. Iterating over strings using a `while` loop.
2. Accessing characters in a string using indices.
3. Converting characters to numbers using `Number`.
4. Performing arithmetic operations (addition, modulo) on numbers.
5. Handling carry values during addition.
6. Concatenating strings using the `+` operator.

## Add Two Numbers

### Add Two Numbers

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     val: number
 *     next: ListNode | null
 *     constructor(val?: number, next?: ListNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *     }
 * }
 */

function addTwoNumbers(l1: ListNode | null, l2: ListNode | null): ListNode | null {

    let head = null

    let temp = null;

    let carry = 0;
```

```

while(l1 !== null || l2 !== null) {
    let sum = carry;
    if(l1 !== null) {
        sum += l1.val;
        l1 = l1.next;
    }
    if(l2 !== null) {
        sum += l2.val;
        l2 = l2.next;
    }

    const node = new ListNode(Math.floor(sum) % 10);
    carry = Math.floor(sum / 10);
    if(temp === null) {
        temp = node;
        head = node;
    } else {
        temp.next = node;
        temp = temp.next;
    }
}

if(carry > 0) {
    temp.next = new ListNode(carry);
}

return head;
};

```

**Solution:** Below are the step-by-step explanations of the code:

1. The code begins with the definition of the `ListNode` class, representing a node in the singly-linked list. It has a `val` property to store the node value and a `next` property to reference the next node in the list.
2. The `addTwoNumbers` function is defined, taking two parameters: `l1` (a `ListNode` representing the first linked list) and `l2` (a `ListNode` representing the second linked list). It returns a `ListNode` representing the sum of the two linked lists.
3. Several variables are initialized:
  - `head` is initially set to `null` and will be used to keep track of the head node of the resulting linked list.
  - `temp` is initially set to `null` and will be used to traverse the linked list and add new nodes.
  - `carry` is initially set to 0 and will hold the carry value during addition.
4. A `while` loop is used to iterate as long as either `l1` or `l2` is not `null`.
5. Inside the `while` loop, the variable `sum` is initialized with the current carry value.
6. Conditional statements are used to handle cases where either `l1` or `l2` is not `null`. If `l1` is not `null`, the value of `l1` is added to `sum`, and `l1` is moved to the next node. Similarly, if `l2` is not `null`, the value of `l2` is added to `sum`, and `l2` is moved to the next node.

7. A new `ListNode` is created with the value of `Math.floor(sum) % 10` (to handle carry) and assigned to the `node` variable.
8. The carry value is updated by calculating `Math.floor(sum / 10)`.
9. Depending on whether `temp` is `null` or not, the `node` is either assigned to `temp` and `head` (in the case of the first node), or it is assigned to `temp.next` (for subsequent nodes), and `temp` is moved to the next node.
10. After the `while` loop ends, if there is a remaining carry (i.e., `carry > 0`), a new `ListNode` with the carry value is appended to the linked list by assigning it to `temp.next`.
11. Finally, the `head` node of the resulting linked list is returned.

#### Techniques used:

The following techniques are utilized within the code:

1. Definition and usage of a `ListNode` class for representing a node in a singly-linked list.
2. Iterating over linked lists using a `while` loop.
3. Accessing properties (`val` and `next`) of a linked list node.
4. Performing arithmetic operations (addition, modulo, division) on numbers.
5. Handling carry values during addition.
6. Creating new linked list nodes using the `ListNode` constructor.
7. Updating references between linked list nodes to form the resulting linked list.

#### Adjacency list to adjacency matrix

### Adjacency list to adjacency matrix

```
class Solution {
    int[][] adjListToMatrix(int n, ArrayList<Integer>[] adjList) {
        int [][] matrix = new int[n][n];

        for(int i = 0; i < n; i++) {
            for(int j = 0; j < adjList[i].size(); j++) {
                matrix[i][adjList[i].get(j)] = 1;
            }
        }

        return matrix;
    }
}
```

**Solution:** Below are the step-by-step explanations of the code:

1. The `Solution` class is defined, containing the `adjListToMatrix` method.
2. The `adjListToMatrix` method takes two parameters:
  - `n`: an integer representing the number of nodes in the graph.
  - `adjList`: an array of `ArrayList<Integer>` representing the adjacency list of the graph.

3. A 2D array, `matrix`, is initialized with dimensions  $n \times n$ . It will hold the adjacency matrix representation of the graph.
4. Two nested `for` loops are used to iterate over each node in the adjacency list and its corresponding neighbors.
5. The outer loop iterates from 0 to  $n-1$ , representing each node in the graph.
6. The inner loop iterates over the neighbors of the current node, accessed through `adjList[i]`, where `i` is the index of the current node.
7. For each neighbor, the corresponding entry in the `matrix` is set to 1 to indicate an edge between the current node and its neighbor.
8. After the nested loops finish, the completed `matrix` representing the adjacency matrix is returned.

#### **Techniques used:**

The following techniques are utilized within the code:

1. Iterating over arrays using `for` loops.
2. Accessing elements of an array using indices.
3. Accessing elements of an `ArrayList` using the `get` method.
4. Constructing a 2D array with specified dimensions.
5. Assigning values to elements of a 2D array.
6. Converting an adjacency list representation to an adjacency matrix representation.

#### **Adjacency matrix to adjacency list**

### **Adjacency matrix to adjacency list**

```
class Solution {
    ArrayList<Integer>[] matrixToAdjList(int n, int[][] matrix) {
        ArrayList<Integer>[] adjList = new ArrayList[n];
        for(int i = 0; i < n; i++) {
            adjList[i] = new ArrayList<Integer>();
        }

        for(int i = 0; i < n; i++) {
            for(int j = 0; j < n; j++) {
                if(matrix[i][j] == 1) {
                    adjList[i].add(j);
                }
            }
        }
        return adjList;
    }
}
```

**Solution:** Below are the step-by-step explanations of the code:

1. The `Solution` class is defined, containing the `adjListToMatrix` method.

2. The `adjListToMatrix` method takes two parameters:
  - `n`: an integer representing the number of nodes in the graph.
  - `adjList`: an array of `ArrayList<Integer>` representing the adjacency list of the graph.
3. A 2D array, `matrix`, is initialized with dimensions `n × n`. It will hold the adjacency matrix representation of the graph.
4. Two nested `for` loops are used to iterate over each node in the adjacency list and its corresponding neighbors.
5. The outer loop iterates from 0 to `n-1`, representing each node in the graph.
6. The inner loop iterates over the neighbors of the current node, accessed through `adjList[i]`, where `i` is the index of the current node.
7. For each neighbor, the corresponding entry in the `matrix` is set to 1 to indicate an edge between the current node and its neighbor.
8. After the nested loops finish, the completed `matrix` representing the adjacency matrix is returned.

#### Techniques used:

The following techniques are utilized within the code:

1. Iterating over arrays using `for` loops.
2. Accessing elements of an array using indices.
3. Accessing elements of an `ArrayList` using the `get` method.
4. Constructing a 2D array with specified dimensions.
5. Assigning values to elements of a 2D array.
6. Converting an adjacency list representation to an adjacency matrix representation.

## Arithmetic sequence

### Arithmetic sequence

```
class Solution {
    boolean isArithmetSequence (int[] arr) {
        int length = arr.length;
        if(length == 1) {
            return true;
        }

        Arrays.sort(arr);
        int diff = arr[1] - arr[0];
        for(int i = 1; i < length; i++) {
            if(arr[i] - arr[i-1] != diff) {
                return false;
            }
        }
        return true;
    }
}
```

**Solution:** Below are the step-by-step explanations of the code:

1. The `Solution` class is defined, containing the `isArithmeticSequence` method.
2. The `isArithmeticSequence` method takes one parameter:
  - `arr`: an array of integers to be checked for being an arithmetic sequence.
3. The length of the array, `length`, is assigned to the `length` variable.
4. An `if` statement is used to check if the length of the array is equal to 1. If so, it means there is only one element in the array, and the method immediately returns `true` since a single element can be considered an arithmetic sequence.
5. The `Arrays.sort()` method is called to sort the array in ascending order. This step is necessary to ensure that the elements are in increasing order for checking the difference.
6. The difference between the second element (`arr[1]`) and the first element (`arr[0]`) is computed and assigned to the `diff` variable. This represents the expected constant difference between consecutive elements in an arithmetic sequence.
7. A `for` loop is used to iterate over the elements of the array starting from the second element (`i = 1`) up to the last element (`i < length`).
8. Inside the loop, an `if` statement is used to check if the difference between the current element and the previous element (`arr[i] - arr[i-1]`) is not equal to the expected difference (`diff`). If the difference is not equal, it means the array does not form an arithmetic sequence, and the method immediately returns `false`.
9. If the loop completes without encountering any mismatch in differences, it means all elements satisfy the condition for an arithmetic sequence, and the method returns `true`.

#### Techniques used:

The following techniques are utilized within the code:

1. Performing operations on array lengths and accessing elements of an array.
2. Sorting an array using `Arrays.sort()`.
3. Computing the difference between two consecutive elements in an array.
4. Using a `for` loop to iterate over array elements.
5. Performing equality checks and conditional branching using `if` statements.
6. Returning boolean values based on the outcome of the checks.

## Arranging Coins

### Arranging Coins

```
function arrangeCoins(n: number): number {
    let i = 1;
    let coins = 0;
    let cnt = 0;

    while((n-coins) >= i) {
        coins += i;
        i++;
        cnt++;
    }
}
```

```

    }

    return cnt;
};

```

**Solution:** Below are the step-by-step explanations of the code:

1. The `arrangeCoins` function is defined, which takes one parameter:
  - `n`: a number representing the total number of coins.
2. Three variables are initialized:
  - `i` is set to 1 and represents the number of coins in the current row.
  - `coins` is set to 0 and will keep track of the total number of coins arranged.
  - `cnt` is set to 0 and will be used to count the number of complete rows.
3. A `while` loop is used to iterate while the number of remaining coins (`n-coins`) is greater than or equal to the number of coins in the current row (`i`).
4. Inside the loop, the number of coins in the current row (`i`) is added to the `coins` variable, representing the accumulation of coins as each row is added.
5. The `i` variable is incremented by 1 to move to the next row.
6. The `cnt` variable is incremented by 1 to count the completion of a full row.
7. The loop continues until the number of remaining coins is no longer sufficient to form a complete row.
8. Once the loop exits, the final value of `cnt` represents the maximum number of complete rows that can be arranged.
9. The `cnt` value is returned as the result.

#### Techniques used:

The following techniques are utilized within the code:

1. Variable initialization and assignment.
2. Incrementing a variable using the `++` operator.
3. Comparison and subtraction operations to control the `while` loop.
4. Accumulation of values using the `+=` operator.
5. Returning a calculated result.

## Array Partition

# Array Partition

```

function arrayPairSum(nums: number[]): number {
  nums.sort((a, b) => a - b);
  let sum = 0;

  for (let i = 0; i < nums.length; i += 2) {

```

```

        sum += nums[i];
    }

    return sum;
};

```

**Solution:** Below are the step-by-step explanations of the code:

1. The `arrayPairSum` function is defined, which takes one parameter:
  - `nums`: an array of numbers.
2. The `sort` method is called on the `nums` array with a comparator function `(a, b) => a - b`. This sorts the array in ascending order.
3. A variable `sum` is initialized to 0, which will store the sum of the pairs.
4. A `for` loop is used to iterate over the elements of the `nums` array with a step size of 2. This means it will process elements at indices 0, 2, 4, and so on.
5. Inside the loop, the current element at index `i` is added to the `sum`.
6. The loop continues until `i` reaches the end of the `nums` array.
7. Once the loop exits, the final value of `sum` represents the maximum possible sum of the pairs formed from the array.
8. The `sum` value is returned as the result.

#### Techniques used:

The following techniques are utilized within the code:

1. Sorting an array using the `sort` method with a comparator function.
2. Iterating over array elements using a `for` loop with a step size.
3. Accumulating values using the `+=` operator.
4. Returning a calculated result.

## Assign Cookies

### Assign Cookies

```

function findContentChildren(g: number[], s: number[]): number {
    const sortedG: number[] = g.sort((a,b) => a-b);
    const sortedS: number[] = s.sort((a,b) => a-b);
    let gCount: number = 0;
    let sCount: number = 0;
    while(gCount < sortedG.length && sCount < sortedS.length) {
        if(sortedG[gCount] <= sortedS[sCount]) {
            gCount++;
            sCount++;
        } else {
            sCount++;
        }
    }
    return sCount;
}

```

```

        }
    }
    return gCount;
};

```

**Solution:** Below are the step-by-step explanations of the code:

1. The `findContentChildren` function is defined, which takes two parameters:
  - `g`: an array of numbers representing the greed factor of children.
  - `s`: an array of numbers representing the size of cookies.
2. Two new arrays, `sortedG` and `sortedS`, are initialized to store the sorted versions of arrays `g` and `s` respectively. The sorting is done in ascending order using the `sort` method with a comparator function `(a, b) => a - b`.
3. Two variables, `gCount` and `sCount`, are initialized to 0. They will keep track of the indices being compared in the sorted arrays.
4. A `while` loop is used to iterate while there are elements remaining in both `sortedG` and `sortedS`.
5. Inside the loop, an `if` statement checks if the greed factor of the current child (`sortedG[gCount]`) is less than or equal to the size of the current cookie (`sortedS[sCount]`).
6. If the condition is true, it means the child can be content with the cookie. Therefore, both `gCount` and `sCount` are incremented to move on to the next child and cookie respectively.
7. If the condition is false, it means the current cookie is not enough to satisfy the child's greed. In this case, only `sCount` is incremented to move on to the next cookie.
8. The loop continues until either `gCount` reaches the end of `sortedG` or `sCount` reaches the end of `sortedS`.
9. Once the loop exits, the value of `gCount` represents the maximum number of children that can be content with the available cookies.
10. The value of `gCount` is returned as the result.

#### Techniques used:

The following techniques are utilized within the code:

1. Sorting arrays using the `sort` method with a comparator function.
2. Iterating over arrays using a `while` loop and comparing array indices.
3. Conditional branching using an `if` statement.
4. Incrementing variables to move to the next element in the array.
5. Returning a calculated result.

## Balanced Binary Tree

### Balanced Binary Tree

```

/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */

function isBalanced(root: TreeNode | null): boolean {
    if(root === null) return true;
    if(height(root) === -1) return false;
    return true;
};

function height(node: TreeNode): number {
    if(node == null) return 0;

    const left = height(node.left)
    const right = height(node.right);

    if(left === -1 || right === -1) return -1;
    if(Math.abs(left - right) > 1) return -1;

    return Math.max(left, right) + 1;
}

```

**Solution:** Below are the step-by-step explanations of the code:

1. A binary tree node is defined using the `TreeNode` class, which has properties:
  - `val`: a number representing the value of the node.
  - `left`: a reference to the left child node.
  - `right`: a reference to the right child node.
2. The `isBalanced` function is defined, which takes one parameter:
  - `root`: a reference to the root node of the binary tree.
3. In the `isBalanced` function, the first condition checks if the `root` is `null`. If it is `null`, it means the tree is empty, and an empty tree is considered balanced. In that case, the function returns `true`.
4. The second condition calls the `height` function to calculate the height of the `root` node. If the height is `-1`, it indicates that the subtree rooted at `root` is not balanced, so the function returns `false`.
5. If both conditions are passed, it means the tree is balanced, and the function returns `true`.
6. The `height` function is defined, which takes one parameter:
  - `node`: a reference to the current node being processed.

7. Inside the `height` function, the first condition checks if the `node` is `null`. If it is `null`, it means the current subtree has no nodes, so the height is 0. In that case, the function returns 0.
8. The `height` function recursively calls itself to calculate the height of the left subtree and assigns the result to the variable `left`.
9. Similarly, the function recursively calls itself to calculate the height of the right subtree and assigns the result to the variable `right`.
10. The next condition checks if either `left` or `right` is `-1`. If any of them is `-1`, it means the respective subtree is not balanced, so the function returns `-1`.
11. The next condition checks if the absolute difference between `left` and `right` is greater than 1. If it is, it means the current node's subtree is not balanced, so the function returns `-1`.
12. If none of the above conditions are met, it means the current node's subtree is balanced. The function returns the maximum of `left` and `right` (the height of the taller subtree) plus 1, representing the height of the current node's subtree.

#### **Techniques used:**

The following techniques are utilized within the code:

1. Defining a binary tree node using a class with properties.
2. Recursive function calls to calculate heights and check subtree balance.
3. Conditional branching using `if` statements.
4. Returning results based on conditions.
5. Performing mathematical calculations using `Math.abs` and `Math.max`.

#### **Balanced brackets**

### **Balanced brackets**

```
const tokens = {
    '{': '}',
    '[': ']',
    '(': ')'
}

function isOpenTerm(s) {
    return tokens[s];
}

function matches(first, second) {
    for(let item in tokens) {
        if(item === first) {
            return tokens[item] === second
        }
    }
    return false;
}

function isBalanced(s) {
    // Write your code here
}
```

```

const stack = [];
for(let i = 0; i < s.length; i++) {
    if(isOpenTerm(s[i])) {
        stack.push(s[i]);
    } else {
        if(stack.length === 0 || !matches(stack.pop(), s[i])) {
            return 'NO';
        }
    }
}
return stack.length === 0 ? 'YES' : 'NO'
}

```

**Solution:** Below are the step-by-step explanations of the code:

1. The `tokens` object is defined, which maps opening parentheses, braces, and brackets to their corresponding closing counterparts.
2. The `isOpenTerm` function is defined, which takes one parameter:
  - `s`: a character representing an opening term.
3. The `isOpenTerm` function returns the corresponding closing term from the `tokens` object by accessing `tokens[s]`. If the opening term exists in the `tokens` object, its corresponding closing term is returned; otherwise, `undefined` is returned.
4. The `matches` function is defined, which takes two parameters:
  - `first`: a character representing an opening term.
  - `second`: a character representing a closing term.
5. The `matches` function iterates over the properties of the `tokens` object using a `for...in` loop. For each property, it checks if `item` (the opening term) is equal to `first`. If there is a match, it compares `tokens[item]` (the corresponding closing term) with `second` and returns `true` if they match; otherwise, it returns `false`.
6. If no match is found in the `tokens` object, the `matches` function returns `false`.
7. The `isBalanced` function is defined, which takes one parameter:
  - `s`: a string of parentheses, braces, and brackets.
8. A stack data structure, `stack`, is initialized as an empty array. It will be used to track opening terms.
9. The function iterates over each character of the input string using a `for` loop.
10. For each character, it checks if it is an opening term by calling the `isOpenTerm` function. If it is an opening term, it is pushed onto the stack.
11. If the character is not an opening term, it means it is a closing term. In this case, it checks if the stack is empty or if the closing term matches the last opening term on the stack using the `matches` function. If either condition fails, indicating unbalanced terms, it returns 'NO'.
12. After processing all characters, the function checks if the stack is empty. If it is empty, it means all opening terms have been matched with their corresponding closing terms, and it returns 'YES'. Otherwise, it returns 'NO'.

### Techniques used:

The following techniques are utilized within the code:

1. Defining an object to store mappings between opening and closing terms.
2. Accessing object properties using dot notation and square bracket notation.
3. Iterating over object properties using a `for...in` loop.
4. Conditional branching using `if` statements.
5. Manipulating an array-based stack by pushing and popping elements.
6. Returning results based on conditions.

\*\* Source: <http://hackerrank.com>\*\*

## Base 7

### Base 7

```
function convertToBase7(num: number): string {
    if (num === 0) {
        return "0";
    }

    let isNegative = num < 0;
    num = Math.abs(num);
    let base7 = "";

    while (num > 0) {
        base7 = (num % 7).toString() + base7;
        num = Math.floor(num / 7);
    }

    if (isNegative) {
        base7 = "-" + base7;
    }

    return base7;
};
```

**Solution:** Below are the step-by-step explanations of the code:

1. The `convertToBase7` function is defined, which takes one parameter:
  - `num`: a number to be converted to base-7.
2. The first condition checks if `num` is equal to 0. If it is, it means the input number is already 0, so the function returns the string "0" as the base-7 representation of 0.
3. A variable `isNegative` is initialized to `true` if `num` is less than 0, indicating a negative number. This is used to determine the sign of the base-7 representation.
4. The `num` is converted to its absolute value using `Math.abs(num)` to ensure positive values for further calculations.

5. A variable `base7` is initialized as an empty string. It will store the base-7 representation of `num`.
6. A `while` loop is used to iterate as long as `num` is greater than 0.
7. Inside the loop, the remainder of `num` divided by 7 (`num % 7`) is computed and converted to a string using `.toString()`. This represents the next digit in the base-7 representation.
8. The obtained digit is appended to the front of the `base7` string using `base7 = (num % 7).toString() + base7`.
9. The `num` is updated by performing integer division (`Math.floor(num / 7)`), effectively removing the last digit in the base-7 representation.
10. Steps 7-9 are repeated until `num` becomes 0.
11. After the loop ends, the base-7 representation, stored in the `base7` string, is ready.
12. If the original `num` was negative (`isNegative` is `true`), a `"+"` sign is prefixed to the `base7` string to indicate the negative sign.
13. The final `base7` string is returned as the result.

#### Techniques used:

The following techniques are utilized within the code:

1. Conditional branching using `if` statements.
2. Manipulating variables to track the sign and absolute value of a number.
3. Performing mathematical operations, such as modulus and integer division.
4. Concatenating strings to build the base-7 representation.
5. Returning a calculated result.

Source: <https://leetcode.com>

## Best Time to Buy and Sell Stock

### Best Time to Buy and Sell Stock

```
function maxProfit(prices: number[]): number {
    let min = 10000;

    let maxDiff = 0;
    for (let i = 0; i < prices.length; i++) {
        min = Math.min(prices[i], min);
        maxDiff = Math.max(prices[i] - min, maxDiff);
    }
    return maxDiff;
};
```

**Solution:** Below are the step-by-step explanations of the code:

1. The `maxProfit` function is defined, which takes one parameter:

- **prices**: an array of numbers representing the stock prices.
2. A variable **min** is initialized to a high value, 10000, to track the minimum price encountered during the iteration. It will be updated as a new minimum price is found.
  3. A variable **maxDiff** is initialized to 0, which will store the maximum difference (profit) encountered during the iteration. It will be updated if a higher difference is found.
  4. A **for** loop is used to iterate over the elements of the **prices** array.
  5. Inside the loop, the minimum value between the current price (**prices[i]**) and the current minimum (**min**) is calculated using **Math.min(prices[i], min)**. This updates the **min** variable to track the new minimum price encountered.
  6. The maximum difference (profit) between the current price and the minimum price (**prices[i] - min**) is calculated using **Math.max(prices[i] - min, maxDiff)**. This updates the **maxDiff** variable if a higher difference is found.
  7. The loop continues until all prices in the array are processed.
  8. Once the loop exits, the final value of **maxDiff** represents the maximum possible profit that can be obtained by buying and selling the stock.
  9. The **maxDiff** value is returned as the result.

#### **Techniques used:**

The following techniques are utilized within the code:

1. Initializing variables to track minimum prices and maximum differences.
2. Iterating over array elements using a **for** loop.
3. Updating variables based on conditions using **Math.min** and **Math.max**.
4. Returning a calculated result.

Source: <https://leetcode.com>

## **Big countries**

**Big countries**

```
SELECT name, population, area FROM World WHERE population >= 25000000 OR area >= 3000000;
```

The given code is a SQL query. It selects the columns **name**, **population**, and **area** from the **World** table and applies a filter using the **WHERE** clause. The filter specifies that only rows satisfying either the condition **population >= 25000000** or the condition **area >= 3000000** should be returned. Here's the breakdown of the query:

- **SELECT**: Specifies the columns to be retrieved from the table.
- **name, population, area**: The columns to be selected.
- **FROM World**: Specifies the table **World** from which the data will be retrieved.
- **WHERE**: Specifies the condition for filtering the rows.
- **population >= 25000000 OR area >= 3000000**: The filtering condition. It checks if either the population is greater than or equal to 25,000,000 or the area is greater than or equal to 3,000,000.

In summary, the query selects the **name**, **population**, and **area** columns from the **World** table, and only retrieves the rows where either the population is greater than or equal to 25,000,000 or the area is greater than or equal to 3,000,000.

Binary search path you took

## Binary search path you took

```
function binarySearch(searchList: number[], value: number): (number[] | string) {
  let path: number[] = [];

  let start: number = 0;
  let end: number = searchList.length - 1;

  while (start <= end) {
    let middle: number = Math.floor((start + end) / 2);
    path.push(searchList[middle]);

    if (searchList[middle] === value) {
      return path;
    } else if (searchList[middle] < value) {
      start = middle + 1;
    } else {
      end = middle - 1;
    }
  }

  return ["Value Not Found"];
}

// Test with the provided array
const testArray: number[] = [
  0, 1, 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
  23, 49, 70
];

const result: (number[] | string) = binarySearch(testArray, 5);
console.log(result); // Output: [4, 6, 5]
```

Solution:

### 1. Function Definition:

- The function `binarySearch` takes two parameters: `searchList` (an array of numbers) and `value` (the target value to search for).
- It returns either an array of numbers (representing the path followed during the binary search) or a string indicating that the value was not found.

### 2. Initialize Variables:

- Create an empty array called `path` to store the path followed during the binary search.
- Initialize `start` to 0, representing the start index of the search list.
- Initialize `end` to the last index of the search list.

### 3. Binary Search Loop:

- Enter a while loop that continues as long as `start` is less than or equal to `end`.

- Calculate the middle index (`middle`) as the floor of the average of `start` and `end`.
- Push the value at the middle index onto the `path` array.

#### 4. Check if Value is Found:

- If the value at the middle index is equal to the target value, return the `path` array.

#### 5. Adjust Search Range:

- If the value at the middle index is less than the target value, update `start` to `middle + 1` (narrowing the search to the right half).
- If the value at the middle index is greater than the target value, update `end` to `middle - 1` (narrowing the search to the left half).

#### 6. Repeat Until Value is Found or Search Range is Empty:

- Continue the loop until either the target value is found or the search range becomes empty.

#### 7. Return "Value Not Found" if Necessary:

- If the loop exits without finding the target value, return an array containing the string "Value Not Found".

The algorithm performs a binary search on a sorted array, updating the search range based on whether the middle element is equal to, less than, or greater than the target value. The path followed during the search is recorded in the `path` array, and this path is returned if the value is found. If the value is not found, the function returns an array with the string "Value Not Found".

## Binary Tree Inorder Traversal

### Binary Tree Inorder Traversal

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */

function inorderTraversal(root: TreeNode | null): number[] {
    const list = [];
    const stack = [];
    let node = root;

    while(node !== null || stack.length > 0) {
        while(node !== null) {
            stack.push(node);
            node = node.left;
        }
        node = stack.pop();
        list.push(node.val);
        node = node.right;
    }
    return list;
}
```

```

        stack.push(node);
        node = node.left
    }
    list.push(stack[stack.length -1].val);
    let el = stack.pop();
    node = el.right;
}
return list;
};

```

**Solution:** Below are the step-by-step explanations of the code:

1. A binary tree node is defined using the `TreeNode` class, which has properties:
  - `val`: a number representing the value of the node.
  - `left`: a reference to the left child node.
  - `right`: a reference to the right child node.
2. The `inorderTraversal` function is defined, which takes one parameter:
  - `root`: a reference to the root node of the binary tree.
3. A variable `list` is initialized as an empty array. It will store the inorder traversal values.
4. A stack, `stack`, is initialized as an empty array. It will be used to keep track of the nodes during traversal.
5. A variable `node` is initialized with the value of the `root` node.
6. The outer `while` loop runs as long as `node` is not `null` or the `stack` is not empty.
7. Inside the outer loop, an inner `while` loop is used to traverse to the leftmost node of the current subtree. It keeps pushing the nodes onto the `stack` until reaching a `null` left child.
8. After the inner loop, the value of the rightmost node in the current subtree is retrieved from the `stack` and added to the `list` using `list.push(stack[stack.length - 1].val)`.
9. The rightmost node is popped from the `stack` and stored in a temporary variable `el`.
10. The `node` is updated to the right child of the popped node (`node = el.right`), which will be processed in the next iteration of the outer loop.
11. The outer loop continues until all nodes are processed.
12. Once the loop exits, the `list` array contains the inorder traversal values of the binary tree.
13. The `list` array is returned as the result.

#### Techniques used:

The following techniques are utilized within the code:

1. Defining a binary tree node using a class with properties.
2. Initializing variables and arrays to store values and references.
3. Iterating using `while` loops.
4. Traversing a binary tree in inorder fashion using a stack.
5. Pushing and popping elements from a stack.
6. Accessing properties and values of nodes.
7. Returning a calculated result.

## Binary Tree Paths

# Binary Tree Paths

```
/**  
 * Definition for a binary tree node.  
 * class TreeNode {  
 *     val: number  
 *     left: TreeNode | null  
 *     right: TreeNode | null  
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {  
 *         this.val = (val===undefined ? 0 : val)  
 *         this.left = (left===undefined ? null : left)  
 *         this.right = (right===undefined ? null : right)  
 *     }  
 * }  
 */  
  
function binaryTreePaths(root: TreeNode | null): string[] {  
    const response: string[] = [];  
    helper(root, "", response);  
    return response;  
};  
  
function helper(root: TreeNode, path: string, response: string[]) {  
    if (root == null) { return; }  
  
    if (root.left == null && root.right == null) {  
        response.push(path + root.val);  
        return;  
    }  
  
    helper(root.left, path + root.val + "->", response);  
    helper(root.right, path + root.val + "->", response);  
}
```

**Solution:** Below are the step-by-step explanations of the code:

1. A binary tree node is defined using the `TreeNode` class, which has properties:
  - `val`: a number representing the value of the node.
  - `left`: a reference to the left child node.
  - `right`: a reference to the right child node.
2. The `binaryTreePaths` function is defined, which takes one parameter:
  - `root`: a reference to the root node of the binary tree.
3. A variable `response` is initialized as an empty array of strings. It will store the paths from the root to the leaf nodes.
4. The `helper` function is called with the `root`, an empty string "", and the `response` array.

5. The `helper` function is defined, which takes three parameters:
  - `root`: a reference to the current node being processed.
  - `path`: a string representing the current path from the root to the current node.
  - `response`: a reference to the array storing the paths.
6. The first condition checks if the `root` is `null`. If it is, it means the current subtree is empty, so the function returns without any further processing.
7. The second condition checks if the `root` is a leaf node, i.e., it has no left or right child. If it is a leaf node, the current path (`path + root.val`) is appended to the `response` array using `response.push(path + root.val)`.
8. If neither of the above conditions is true, it means the current node is not a leaf node. The `helper` function is recursively called with the left child node, updating the `path` by appending the current node's value and a "`->`" separator. This explores the left subtree.
9. Similarly, the `helper` function is recursively called with the right child node, updating the `path` in the same way. This explores the right subtree.
10. The recursive calls continue until all paths from the root to the leaf nodes are traversed.
11. Once all recursive calls are completed, the `response` array contains all the paths from the root to the leaf nodes.
12. The `response` array is returned as the result.

#### Techniques used:

The following techniques are utilized within the code:

1. Defining a binary tree node using a class with properties.
2. Initializing variables and arrays to store values and references.
3. Recursive function calls to explore the tree.
4. Conditional branching using `if` statements.
5. Appending strings and values to an array.
6. Returning a calculated result.

#### Binary Tree Postorder Traversal

### Binary Tree Postorder Traversal

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *   val: number
 *   left: TreeNode | null
 *   right: TreeNode | null
 *   constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 *   }
 * }
```

```

/*
function postorderTraversal(root: TreeNode | null): number[] {
  var res = [];
  helper(root, res);
  return res;
};

const helper = function (root, res) {
  if (!root) return;
  helper(root.left, res);
  helper(root.right, res);
  res.push(root.val);
};

```

**Solution:** Below are the step-by-step explanations of the code:

1. A binary tree node is defined using the `TreeNode` class, which has properties:
  - `val`: a number representing the value of the node.
  - `left`: a reference to the left child node.
  - `right`: a reference to the right child node.
2. The `postorderTraversal` function is defined, which takes one parameter:
  - `root`: a reference to the root node of the binary tree.
3. A variable `res` is initialized as an empty array. It will store the postorder traversal values.
4. The `helper` function is called with the `root` and the `res` array.
5. The `helper` function is defined, which takes two parameters:
  - `root`: a reference to the current node being processed.
  - `res`: a reference to the array storing the traversal values.
6. The first condition checks if the `root` is `null`. If it is, it means the current subtree is empty, so the function returns without any further processing.
7. The `helper` function is recursively called with the left child node, which explores the left subtree.
8. Similarly, the `helper` function is recursively called with the right child node, which explores the right subtree.
9. The recursive calls continue until all nodes are traversed in a postorder manner (left subtree, right subtree, current node).
10. After the recursive calls, the function pushes the value of the current node (`root.val`) to the `res` array using `res.push(root.val)`. This appends the value in the postorder traversal order.
11. Once all recursive calls are completed, the `res` array contains the postorder traversal values of the binary tree.
12. The `res` array is returned as the result.

#### Techniques used:

The following techniques are utilized within the code:

1. Defining a binary tree node using a class with properties.
2. Initializing variables and arrays to store values and references.
3. Recursive function calls to explore the tree.
4. Conditional branching using `if` statements.
5. Appending values to an array.
6. Returning a calculated result.

## Binary Tree Preorder Traversal

# Binary Tree Preorder Traversal

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */

function preorderTraversal(root: TreeNode | null): number[] {
    const output: any[] = [];
    preorder(root, output);
    return output;
};

function preorder(root: TreeNode | null, output: any[]) {
    if (root == null) return;

    output.push(root.val);
    preorder(root.left, output);
    preorder(root.right, output);
}
```

**Solution:** Below are the step-by-step explanations of the code:

1. A binary tree node is defined using the `TreeNode` class, which has properties:
  - `val`: a number representing the value of the node.
  - `left`: a reference to the left child node.
  - `right`: a reference to the right child node.
2. The `preorderTraversal` function is defined, which takes one parameter:
  - `root`: a reference to the root node of the binary tree.
3. A variable `output` is initialized as an empty array. It will store the preorder traversal values.

4. The `preorder` function is called with the `root` and the `output` array.
5. The `preorder` function is defined, which takes two parameters:
  - `root`: a reference to the current node being processed.
  - `output`: a reference to the array storing the traversal values.
6. The first condition checks if the `root` is `null`. If it is, it means the current subtree is empty, so the function returns without any further processing.
7. The function pushes the value of the current node (`root.val`) to the `output` array using `output.push(root.val)`. This appends the value in the preorder traversal order.
8. The `preorder` function is recursively called with the left child node, which explores the left subtree.
9. Similarly, the `preorder` function is recursively called with the right child node, which explores the right subtree.
10. The recursive calls continue until all nodes are traversed in a preorder manner (current node, left subtree, right subtree).
11. Once all recursive calls are completed, the `output` array contains the preorder traversal values of the binary tree.
12. The `output` array is returned as the result.

#### **Techniques used:**

The following techniques are utilized within the code:

1. Defining a binary tree node using a class with properties.
2. Initializing variables and arrays to store values and references.
3. Recursive function calls to explore the tree.
4. Conditional branching using `if` statements.
5. Appending values to an array.
6. Returning a calculated result.

#### **Binary Tree Tilt**

### **Binary Tree Tilt**

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */
```

```

function findTilt(root: TreeNode | null): number {
    let tilt = 0;

    function calculateTiltAndSum(node: TreeNode | null): number {
        if (node === null) {
            return 0;
        }

        const leftSum = calculateTiltAndSum(node.left);
        const rightSum = calculateTiltAndSum(node.right);

        tilt += Math.abs(leftSum - rightSum);

        return node.val + leftSum + rightSum;
    }

    calculateTiltAndSum(root);

    return tilt;
};

```

**Solution:** Below are the step-by-step explanations of the code:

1. A binary tree node is defined using the `TreeNode` class, which has properties:
  - `val`: a number representing the value of the node.
  - `left`: a reference to the left child node.
  - `right`: a reference to the right child node.
2. The `findTilt` function is defined, which takes one parameter:
  - `root`: a reference to the root node of the binary tree.
3. A variable `tilt` is initialized to 0. It will store the total tilt of the binary tree.
4. The `calculateTiltAndSum` function is defined inside the `findTilt` function. It takes one parameter:
  - `node`: a reference to the current node being processed.
5. The `calculateTiltAndSum` function calculates the sum of the node's value, left subtree values, and right subtree values recursively. It also updates the `tilt` variable by adding the absolute difference between the sum of left subtree values and right subtree values.
6. The first condition checks if the `node` is `null`. If it is, it means the current subtree is empty, so the function returns 0.
7. The function recursively calls `calculateTiltAndSum` with the left child node and assigns the returned value to the variable `leftSum`.
8. Similarly, the function recursively calls `calculateTiltAndSum` with the right child node and assigns the returned value to the variable `rightSum`.
9. The absolute difference between `leftSum` and `rightSum` is calculated using `Math.abs(leftSum - rightSum)`, and the result is added to the `tilt` variable.

10. The sum of the current node's value, `leftSum`, and `rightSum` is calculated and returned (`node.val + leftSum + rightSum`).
11. The `calculateTiltAndSum` function is initially called with the `root` node.
12. The `tilt` value now contains the total tilt of the binary tree.
13. The `tilt` value is returned as the result.

#### Techniques used:

The following techniques are utilized within the code:

1. Defining a binary tree node using a class with properties.
2. Initializing variables to store values.
3. Recursive function calls to explore the tree.
4. Conditional branching using `if` statements.
5. Calculating absolute differences using `Math.abs`.
6. Updating variables with calculated values.
7. Returning a calculated result.

## Binary Watch

### Binary Watch

```
class Solution {
    public List<String> readBinaryWatch(int turnedOn) {
        List<String> res = new ArrayList<>();
        for(int h = 0; h < 12; h++){
            for(int m = 0; m < 60; m++){
                if(Integer.bitCount(h) + Integer.bitCount(m) == turnedOn){
                    res.add(String.format("%d:%02d", h, m));
                }
            }
        }
        return res;
    }
}
```

**Solution:** Below is the step-by-step explanation of the code:

1. The `readBinaryWatch` method is defined as a part of the `Solution` class. It takes an integer `turnedOn` as the input.
2. A new `ArrayList` called `res` is created to store the resulting times on the binary watch.
3. The outer `for` loop iterates over the possible values of the hours (0 to 11).
4. The inner `for` loop iterates over the possible values of the minutes (0 to 59).
5. For each combination of hours and minutes, the condition `Integer.bitCount(h) + Integer.bitCount(m) == turnedOn` is checked. This condition ensures that the total number of turned-on LEDs in the binary representation of hours and minutes combined is equal to `turnedOn`.

6. If the condition is satisfied, the current time is added to the `res` list using `String.format("%d:%02d", h, m)`. The "%d:%02d" format is used to represent the time in the HH:MM format with leading zeros for minutes.
7. The loops continue until all possible combinations of hours and minutes are checked.
8. Once the loops exit, the `res` list contains all the valid times where the number of turned-on LEDs matches `turnedOn`.
9. The `res` list is returned as the result.

#### **Techniques used:**

The following techniques are utilized within the code:

1. Using nested `for` loops to iterate over all possible combinations of hours and minutes.
2. Checking the number of turned-on LEDs in the binary representation using `Integer.bitCount`.
3. Formatting the time string using `String.format`.
4. Adding elements to an `ArrayList`.
5. Returning a calculated result.

## **Can Place Flowers**

### **Can Place Flowers**

```
function canPlaceFlowers(flowerbed: number[], n: number): boolean {
    let count = 0;

    for (let i = 0; i < flowerbed.length; i++) {
        if (flowerbed[i] === 0) {
            const prevEmpty = i === 0 || flowerbed[i - 1] === 0;
            const nextEmpty = i === flowerbed.length - 1 || flowerbed[i + 1] === 0;

            if (prevEmpty && nextEmpty) {
                flowerbed[i] = 1;
                count++;
            }
        }
    }

    return count >= n;
};
```

#### **1. Initialize Counter:**

- Initialize a counter variable `count` to keep track of the number of flowers placed.

#### **2. Iterate Through Flowerbed:**

- Use a `for` loop to iterate through the `flowerbed` array.

#### **3. Check Current Position:**

- Check if the current position in the flowerbed is empty (`flowerbed[i] === 0`).

#### 4. Check Previous and Next Positions:

- Check if the previous position is empty (`i === 0 || flowerbed[i - 1] === 0`) and if the next position is empty (`i === flowerbed.length - 1 || flowerbed[i + 1] === 0`).

#### 5. Place Flower:

- If both the previous and next positions are empty, place a flower at the current position (`flowerbed[i] = 1`) and increment the count.

#### 6. Repeat Until End:

- Continue iterating through the flowerbed, checking and placing flowers in every valid position.

#### 7. Check if Enough Flowers Placed:

- After the iteration, check if the number of placed flowers (`count`) is greater than or equal to the required number of flowers (`n`).

#### 8. Return Result:

- Return `true` if enough flowers are placed; otherwise, return `false`.

## Chain adding functions

## Chain adding functions

Multiple calls:

```
add(1)(2)(3); // 6 add(1)(2)(3)(4); // 10 add(1)(2)(3)(4)(5); // 15
```

Single call: `add(1); // 1`

Mixed thing: `let addTwo = add(2); addTwo; // 2 addTwo + 5; // 7 addTwo(3); // 5 addTwo(3)(5); // 10`

```
function add(x: number): any {
  let sum = x;
  function f(y: number) {
    sum += y;
    return f;
  }
  f.toString = function() {
    return sum;
  };
  return f;
}
```

The provided code defines a function `add` that returns a special function `f` with a cumulative behavior. Each time `f` is called with a parameter `y`, it adds `y` to the accumulated sum `sum` and returns `f` itself, allowing for chaining multiple function calls. The `toString` method is overridden to return the current value of `sum` when `f` is coerced to a string.

Here's a step-by-step breakdown of the code:

1. The `add` function is defined, which takes a parameter `x` of type number.
2. Inside the `add` function, a local variable `sum` is initialized with the value of `x`. This variable will hold the accumulated sum.
3. The function `f` is defined inside the `add` function. It takes a parameter `y` of type number.
4. Within `f`, the `sum` variable is updated by adding `y` to it (`sum += y`).
5. The `f` function returns itself, allowing for chaining multiple function calls.
6. The `toString` method is defined for the `f` function. It overrides the default `toString` method and returns the value of `sum` when `f` is coerced to a string.
7. Finally, the `f` function is returned from the `add` function.

This code implements a concept called "currying" where functions are transformed to take multiple arguments by returning a series of functions that each take one argument. The returned function `f` allows for a fluent syntax where multiple numbers can be added together by calling `add(x)(y)(z)...` and the final value can be obtained by coercing `add(x)(y)(z)...` to a string.

Example usage:

```
const result = add(1)(2)(3)(4).toString();
console.log(result); // Output: 10
```

In this example, the `add` function is called with the initial value of 1, and then `f` is repeatedly called with subsequent values 2, 3, and 4. The cumulative sum is  $1 + 2 + 3 + 4 = 10$ . When `toString()` is called on the returned function, it returns the string representation of the accumulated sum, which is "10".

## Chain calculator

### Chain calculator

```
function ChainCalculator(given) {
  this.num = given || 0;
  const actions = {
    half: ['this.num = this.num/2; return this;'],
    quarter: ['this.num = this.num/4; return this;'],
    third: ['this.num = this.num/3; return this;'],
    pow: ['given', 'this.num = Math.pow(this.num, given); return this;'],
    sqrt: ['this.num = Math.sqrt(this.num); return this;'],
    log: ['console.log(this.num); return this;'],
    sum: ['given', `this.num += given; return this;`],
    minus: ['given', `this.num -= given; return this;`],
    multiply: ['given', `this.num *= given; return this;`],
    divide: ['given', `this.num /= given; return this;`],
    finish: ['return this.num;']
  }
  for(const key in actions) {
    this[key] = new Function(...actions[key])
  }
}
```

```

}

const calculator = new ChainCalculator();

calculator.log().sum(15).log().sum(10).log().finish().log()

```

The code above defines a `ChainCalculator` constructor function that creates calculator objects with chainable operations. The calculator can perform various mathematical operations on a number value and supports method chaining. The operations are defined as properties of the calculator object, and each operation returns the calculator object itself, allowing for sequential method calls.

Here's a step-by-step breakdown of the code:

1. The `ChainCalculator` constructor function is defined, which takes an optional parameter `given` (defaults to 0). This parameter represents the initial value of the calculator.
2. Inside the constructor function, the `num` property is assigned the value of `given`. This property holds the current number value.
3. The `actions` object is defined, which maps operation names to their corresponding code strings. Each code string is an array of statements that modify the `num` property and return the calculator object.
4. A loop is used to iterate over each key (operation name) in the `actions` object.
5. For each key, a new function is created using the `Function` constructor. The code string for the corresponding operation is passed as arguments to the `Function` constructor to create the function.
6. The newly created function is assigned as a property of the calculator object using the current key (operation name).
7. The `...actions[key]` syntax is used to spread the code string array elements as separate arguments to the `Function` constructor. This allows the function to be defined with the statements from the code string.
8. After the loop, an instance of the `ChainCalculator` is created using the `new` keyword, without passing any arguments. This creates a calculator object with an initial value of 0.
9. The `calculator` object is used to chain method calls, performing various operations on the number value.
10. The `log()` method is called on the `calculator` object, which logs the current number value to the console. It returns the `calculator` object itself.
11. The `sum(15)` method is called on the `calculator` object, which adds 15 to the current number value. It returns the `calculator` object itself.
12. The `log()` method is called again, logging the updated number value to the console.
13. The `sum(10)` method is called, adding 10 to the current number value.
14. The `log()` method is called once more, logging the final number value to the console.
15. The `finish()` method is called, which returns the final number value.
16. The `log()` method is called again, but since it doesn't return the `calculator` object, it doesn't have any effect.

Please note that the code provided uses the `Function` constructor to dynamically create functions from code strings. While this approach offers flexibility, it can also be potentially unsafe if the code strings come from untrusted sources. It's important to ensure that the code strings used in this manner are secure and not susceptible to code injection vulnerabilities.

## Classes More Than 5 Students

# Classes More Than 5 Students

```
SELECT class
FROM Courses
GROUP BY class
HAVING COUNT(student) >= 5;
```

### 1. Select Statement:

- Use the `SELECT` statement to retrieve data from the database.

### 2. Column Selection:

- Specify the column to be selected as `class`.

### 3. Table Specification:

- Specify the table from which to retrieve data as `Courses`.

### 4. Grouping:

- Use the `GROUP BY` clause to group the results based on the values in the "class" column.

### 5. Counting Distinct Students:

- Use the `COUNT(student)` function within the `HAVING` clause to filter groups where the count of distinct values in the "student" column is greater than or equal to 5.

### 6. Filtering with HAVING Clause:

- Apply the `HAVING` clause to filter groups based on the count condition.

### 7. Final Result:

- Retrieve and display the distinct values in the "class" column that meet the specified criteria.

The result is a list of distinct class values from the "Courses" table where there are at least 5 distinct students enrolled in each class.

## Climbing stairs

# Climbing stairs

```
function climbStairs(n: number): number {
  if(n <= 3) {
    return n;
  }

  let a = 3;
```

```

let b = 2;

for(let i = 0; i < n -3; i++) {
    a = a + b;
    b = a - b;
}
return a;
};

```

**Solution:** Below is the step-by-step breakdown of the code:

1. The `climbStairs` function takes an integer `n` as input.
2. If `n` is less than or equal to 3, there are `n` distinct ways to climb the stairs. In this case, `n` is directly returned.
3. If `n` is greater than 3, two variables `a` and `b` are initialized. `a` represents the number of distinct ways to reach the current step, and `b` represents the number of distinct ways to reach the previous step.
4. A loop is started from 0 up to `n - 3` (since the cases for `n <= 3` are handled separately).
5. Inside the loop, `a` is updated to the sum of its current value and `b`. This represents the number of distinct ways to reach the current step based on the number of ways to reach the previous step and the step before that.
6. The value of `b` is updated to `a - b`, which becomes the number of distinct ways to reach the previous step for the next iteration.
7. Once the loop completes, the value of `a` represents the total number of distinct ways to climb the stairs with `n` steps.
8. The final value of `a` is returned as the result.

#### Techniques used:

The following techniques are utilized within the code:

1. Conditional branching using `if` statement.
2. Iterative loop using `for` loop.
3. Updating variables to store intermediate results.
4. Mathematical computation to calculate the number of distinct ways.
5. Efficient variable swapping using `a = a + b;`; `b = a - b;`.

This approach uses a dynamic programming-like approach to solve the problem, avoiding the need for recursion or maintaining an array. By iteratively updating the number of ways to reach each step based on the previous steps, the code calculates the total number of distinct ways to climb the stairs.

The time complexity of this solution is  $O(n)$  since the loop iterates  $n - 3$  times. It has a constant space complexity since it only requires two variables to store the intermediate results.

**Source:** <https://leetcode.com>

## Combine Two Tables

### Combine Two Tables

```
select p.FirstName, p.LastName, a.City, a.State from person p
left join address a on p.personid = a.personid
```

The SQL query retrieves the `FirstName` and `LastName` columns from the `person` table, along with the `City` and `State` columns from the `address` table. It performs a `LEFT JOIN` operation between the `person` and `address` tables using the `personid` column as the join condition.

Here's a breakdown of the query:

1. The `SELECT` statement specifies the columns to be retrieved from the tables.
2. `p.FirstName` retrieves the `FirstName` column from the `person` table.
3. `p.LastName` retrieves the `LastName` column from the `person` table.
4. `a.City` retrieves the `City` column from the `address` table.
5. `a.State` retrieves the `State` column from the `address` table.
6. The `FROM` clause specifies the source tables for the query.
7. `person p` specifies the `person` table with an alias of `p`.
8. The `LEFT JOIN` keyword performs a left outer join operation between the `person` and `address` tables.
9. `address a` specifies the `address` table with an alias of `a`.
10. The `ON` keyword specifies the join condition between the `person` and `address` tables. In this case, the join is based on matching values in the `personid` column.

The result of the query will contain rows with the `FirstName` and `LastName` values from the `person` table, along with the corresponding `City` and `State` values from the `address` table. If there is no matching address for a person, the `City` and `State` values will be `NULL` for that person.

- Go back

## Construct the Rectangle

```
function constructRectangle(area: number): number[] {
    let width = Math.floor(Math.sqrt(area));

    while (area % width !== 0) {
        width--;
    }

    const length = area / width;
    return [length, width];
};
```

**Solution:** Below is the step-by-step breakdown of the code:

1. The `constructRectangle` function takes a number `area` as input.
2. The variable `width` is initialized to the square root of the `area` using the `Math.sqrt` function and rounded down to the nearest integer using the `Math.floor` function. This provides an initial estimate for the width of the rectangle.
3. A `while` loop is used to find the exact width of the rectangle. The loop continues as long as the remainder of dividing `area` by `width` is not equal to 0.
4. Inside the loop, `width` is decremented by 1 in each iteration until a width is found that evenly divides the area without any remainder.
5. Once the loop exits, the value of `width` represents the exact width of the rectangle.
6. The variable `length` is calculated by dividing the `area` by the `width`, which gives the corresponding length of the rectangle.
7. The function returns an array `[length, width]` containing the calculated dimensions of the rectangle.

#### Techniques used:

The following techniques are utilized within the code:

1. Mathematical computation using `Math.sqrt` to calculate the square root of a number and `Math.floor` to round down a decimal number to the nearest integer.
2. Looping using a `while` loop to perform iterative computations until a certain condition is met.
3. Conditional branching using the `while` loop condition to check for the remainder of the division.
4. Variable manipulation by decrementing the `width` variable inside the loop.
5. Division and arithmetic computation to calculate the length of the rectangle based on the given area.
6. Array creation and return using `[length, width]` to return the calculated dimensions.

The code efficiently calculates the dimensions of a rectangle given its area by using the square root of the area as an initial estimate for the width and then iteratively adjusting the width until an exact value is found. The calculated width and length guarantee that their product equals the given area.

The time complexity of this solution is  $O(\sqrt{\text{area}})$  since it uses `Math.sqrt` to calculate the initial width, and the loop has a maximum number of iterations equal to the square root of the area.

**Source:** <https://leetcode.com>

## Container With Most Water

```
function maxArea(height: number[]): number {
    let maximumArea = Number.MIN_SAFE_INTEGER;
    let left = 0;
    let right = height.length - 1;
    while (left < right) {
        let shorterLine = Math.min(height[left], height[right]);
        maximumArea = Math.max(maximumArea, shorterLine * (right - left));
        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }
    return maximumArea;
}
```

```

        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }
    return maximumArea;
};

}

```

**Solution:** Below is the step-by-step breakdown of the code:

1. The `maxArea` function takes an array `height` as input.
2. The variable `maximumArea` is initialized to the minimum safe integer value using `Number.MIN_SAFE_INTEGER`. This variable will be used to track the maximum area encountered.
3. The variables `left` and `right` are initialized to 0 and `height.length - 1` respectively. These variables represent the left and right pointers that will traverse the array.
4. A `while` loop is used to iterate until the `left` pointer is less than the `right` pointer. This ensures that all possible container configurations are explored.
5. Inside the loop, the variable `shorterLine` is calculated as the minimum value between the heights at `height[left]` and `height[right]`. This represents the height of the shorter line in the container.
6. The maximum area is updated by taking the maximum value between the current `maximumArea` and the product of `shorterLine` and the distance between the `right` and `left` pointers. This accounts for the width of the container.
7. Depending on the comparison between `height[left]` and `height[right]`, either the `left` pointer is incremented (if `height[left]` is smaller) or the `right` pointer is decremented (if `height[right]` is smaller). This allows the pointers to move towards the center, exploring potentially higher lines.
8. Once the loop exits, the value of `maximumArea` represents the maximum area of water that can be contained in the container formed by the vertical lines.
9. The maximum area is returned as the result.

#### Techniques used:

The following techniques are utilized within the code:

1. Two-pointer technique using `left` and `right` pointers to traverse the array from both ends towards the center.
2. Conditional branching using an `if` statement to compare the heights of the lines and determine which pointer to move.
3. Updating variables to track the maximum area encountered and the height of the shorter line.
4. Mathematical computation to calculate the area based on the shorter line's height and the width of the container.
5. Looping using a `while` loop to iterate until a certain condition is met.
6. Efficient variable initialization using `Number.MIN_SAFE_INTEGER` to initialize `maximumArea`.

This approach efficiently explores all possible configurations of the container by utilizing the two-pointer technique and calculating the area based on the shorter line and the width. By gradually moving the pointers towards the center, the algorithm examines potentially higher lines and keeps track of the maximum area encountered.

The time complexity of this solution is  $O(n)$  since it uses a single loop to traverse the `height` array. The space complexity is  $O(1)$  as the algorithm uses a constant amount of additional space to store variables.

**Source:** <https://leetcode.com>

## Contains Duplicates 2

### Contains Duplicates 2

```
function containsNearbyDuplicate(nums: number[], k: number): boolean {
    const seen: any = new Map();
    for (let i = 0; i < nums.length; ++i) {
        if (i - seen.get(nums[i]) <= k) {
            return true;
        }
        seen.set(nums[i], i);
    }
    return false;
};
```

**Solution:** Below is the step-by-step breakdown of the code:

1. The `containsNearbyDuplicate` function takes an array `nums` and a distance `k` as input.
2. A `Map` object named `seen` is initialized to keep track of previously seen elements and their indices.
3. A `for` loop is used to iterate over the elements of the `nums` array.
4. For each element at index `i`, the code checks if there exists a previous occurrence of the same element within the distance `k` from the current index. This is done by retrieving the index of the previous occurrence from the `seen` Map using `seen.get(nums[i])` and comparing it with the current index `i`.
5. If the condition `i - seen.get(nums[i]) <= k` is satisfied, it means that a duplicate element within the given distance `k` has been found. In such a case, the function returns `true` to indicate the presence of nearby duplicates.
6. If no duplicate element is found within the distance `k`, the current element and its index `i` are added to the `seen` Map using `seen.set(nums[i], i)`. This allows tracking of the latest occurrence of each element.
7. Once the loop completes without finding any duplicates, the function returns `false` to indicate the absence of nearby duplicates.

#### Techniques used:

The following techniques are utilized within the code:

1. Map data structure: A `Map` object named `seen` is used to store previously seen elements and their indices, allowing efficient lookup and retrieval.

2. Iteration: The `for` loop is used to iterate over the elements of the `nums` array.
3. Conditional branching: An `if` statement is used to check the condition `i - seen.get(nums[i]) <= k` and determine whether a duplicate element within the specified distance `k` has been found.
4. Index retrieval and comparison: The code retrieves the index of a previously seen element using `seen.get(nums[i])` and compares it with the current index `i` to check the distance between occurrences.
5. Element and index tracking: The code updates the `seen` Map with the latest occurrence of each element and its index using `seen.set(nums[i], i)`.

The provided solution efficiently checks for nearby duplicates by utilizing a `Map` data structure to store previously seen elements and their indices. By iterating over the array and comparing indices, the algorithm determines whether there are any duplicate elements within the specified distance `k`.

The time complexity of this solution is  $O(n)$ , where  $n$  is the length of the `nums` array, since it requires iterating over the entire array. The space complexity is also  $O(n)$  due to the usage of the `seen` Map to store elements and their indices.

**Source:** <https://leetcode.com>

## Contains Duplicate

## Contains Duplicate

```
function containsDuplicate(nums: number[]): boolean {
    nums.sort((a: any, b: any) => a-b);
    for(let i = 0; i < nums.length -1; i++) {
        if(nums[i] === nums[i+1]) {
            return true;
        }
    }
    return false;
};
```

**Solution:** Below is the step-by-step breakdown of the code:

1. The `containsDuplicate` function takes an array `nums` as input.
2. The `sort` method is called on the `nums` array using a comparison function `(a: any, b: any) => a - b`. This sorts the array in ascending order.
3. A `for` loop is used to iterate over the elements of the `nums` array from index 0 to `nums.length - 1`.
4. Inside the loop, the code compares each element `nums[i]` with its adjacent element `nums[i+1]`.
5. If a duplicate element is found (i.e., `nums[i] === nums[i+1]`), the function immediately returns `true` to indicate the presence of duplicates.
6. If no duplicates are found after the loop completes, the function returns `false` to indicate the absence of duplicates.

### Techniques used:

The following techniques are utilized within the code:

1. Sorting: The `sort` method is used to sort the `nums` array in ascending order. This brings duplicate elements next to each other, simplifying the duplicate check.
2. Iteration: The `for` loop is used to iterate over the elements of the `nums` array.
3. Conditional branching: An `if` statement is used to compare adjacent elements and check for duplicates.

The provided solution sorts the array to bring duplicate elements together, allowing for a simple comparison of adjacent elements to determine duplicates. This approach has a time complexity of  $O(n \log n)$  due to the sorting operation and a space complexity of  $O(1)$  since no additional data structures are used.

**Source:** <https://leetcode.com>

## Convert a Number to Hexadecimal

### Convert a Number to Hexadecimal

```
class Solution {
    public String toHex(int num) {
        if(num == 0) return "0";
        char[] map = {'0','1','2','3','4','5','6','7','8','9','a','b','c','d','e','f'};
        StringBuilder sb = new StringBuilder();
        while(num != 0){
            sb.insert(0, map[num & 15]);
            num = num >>> 4;
        }
        return sb.toString();
    }
}
```

**Solution:** Below is the step-by-step breakdown of the code:

1. The `toHex` method takes an integer `num` as input and returns its hexadecimal representation as a string.
2. The code checks if `num` is equal to 0. If it is, the method immediately returns the string "0" since the hexadecimal representation of 0 is "0".
3. An array `map` is initialized with characters representing the hexadecimal digits from 0 to 15. This array allows easy mapping between decimal values and their corresponding hexadecimal characters.
4. A `StringBuilder` object named `sb` is created to build the hexadecimal string.
5. The code enters a `while` loop that continues until `num` becomes 0.
6. Inside the loop, the code performs the following operations:
  - It extracts the last 4 bits of `num` using the bitwise AND operation `num & 15`. The number 15 is represented in binary as 0000 1111, so the AND operation with 15 extracts the rightmost 4 bits.
  - It inserts the hexadecimal character corresponding to the extracted value at the beginning of the `sb` using `sb.insert(0, map[num & 15])`.
  - It right shifts `num` by 4 bits using the logical right shift operator `num >>> 4` to discard the processed bits.
7. After the loop completes, the `sb` contains the hexadecimal representation of the input `num`. It is converted to a string using `sb.toString()` and returned as the result.

### Techniques used:

The following techniques are utilized within the code:

1. Bitwise operations: The code uses the bitwise AND operation `num & 15` to extract the last 4 bits of `num` and convert them to a hexadecimal digit. It also uses the logical right shift operator `num >>> 4` to discard processed bits during each iteration.
2. StringBuilder: The `StringBuilder` object `sb` is used to efficiently build the hexadecimal string by inserting characters at the beginning.

The provided solution efficiently converts an integer to its hexadecimal representation by extracting the bits and mapping them to their corresponding hexadecimal characters. It utilizes bitwise operations and a `StringBuilder` object to perform the conversion.

### Convert Sorted Array to Binary Search Tree

## Convert Sorted Array to Binary Search Tree

```
/**  
 * Definition for a binary tree node.  
 * class TreeNode {  
 *     val: number  
 *     left: TreeNode | null  
 *     right: TreeNode | null  
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {  
 *         this.val = (val===undefined ? 0 : val)  
 *         this.left = (left===undefined ? null : left)  
 *         this.right = (right===undefined ? null : right)  
 *     }  
 * }  
 */  
  
function sortedArrayToBST(nums: number[]): TreeNode | null {  
    return helper(nums, 0, nums.length - 1);  
};  
  
function helper(nums: number[], start:number, end: number) {  
    if(start > end) return null;  
    let mid = Math.trunc((start + end) / 2);  
    let node = new TreeNode(nums[mid]);  
    node.left = helper(nums, start, mid -1);  
    node.right = helper(nums, mid + 1, end);  
    return node  
}
```

**Solution:** Below is the step-by-step breakdown of the code:

1. The `sortedArrayToBST` function takes a sorted array `nums` as input and returns the root node of the constructed BST.

2. The function calls a helper function `helper` with the array `nums`, start index 0, and end index `nums.length - 1` as arguments.
3. The `helper` function is a recursive function that constructs a BST from a subarray of `nums`.
4. If the start index `start` is greater than the end index `end`, the `helper` function immediately returns `null` to indicate an empty subarray.
5. If the subarray is not empty, the `helper` function calculates the middle index `mid` as the truncation of the average of `start` and `end`.
6. A new `TreeNode` object is created with the value at the middle index `nums[mid]`.
7. Recursive calls to the `helper` function are made to construct the left subtree and right subtree of the current node:
  - The left subtree is constructed by calling `helper` with `start` and `mid - 1` as the new start and end indices.
  - The right subtree is constructed by calling `helper` with `mid + 1` and `end` as the new start and end indices.
8. The left and right subtrees are assigned to the `left` and `right` properties of the current node, respectively.
9. Finally, the constructed node is returned.

#### **Techniques used:**

The following techniques are utilized within the code:

1. Recursion: The `helper` function uses recursion to construct the BST. It recursively divides the input array into smaller subarrays to create balanced subtrees.
2. Divide and Conquer: The code follows a divide-and-conquer strategy by splitting the sorted array into halves and constructing balanced subtrees for each half.
3. Binary Search Tree (BST): The code constructs a BST by assigning values from the sorted array to the tree nodes. The left subtree contains smaller values, and the right subtree contains larger values.

The provided solution effectively constructs a height-balanced BST from a sorted array using recursive divide-and-conquer techniques.

#### **Count and Say**

### **Count and Say**

```
function countAndSay(n: number): string {
  if (n === 1) {
    return "1";
  }

  const prevResult = countAndSay(n - 1);
  let result = "";
  let count = 1;
```

```

        for (let i = 0; i < prevResult.length; i++) {
            if (prevResult[i] === prevResult[i + 1]) {
                count++;
            } else {
                result += count + prevResult[i];
                count = 1;
            }
        }

        return result;
    };

```

**Solution:** Let's break down the provided TypeScript function `countAndSay` step by step:

```

function countAndSay(n: number): string {
    // Step 1: Base case - when n is 1, return "1"
    if (n === 1) {
        return "1";
    }

    // Step 2: Recursively call countAndSay for n-1
    const prevResult = countAndSay(n - 1);

    // Step 3: Initialize variables for the current result and count
    let result = "";
    let count = 1;

    // Step 4: Iterate through the previous result to generate the current result
    for (let i = 0; i < prevResult.length; i++) {
        // Step 5: Check if the current character is the same as the next one
        if (prevResult[i] === prevResult[i + 1]) {
            // If yes, increment the count
            count++;
        } else {
            // If no, append the count and the character to the result
            result += count + prevResult[i];
            // Reset the count for the new character
            count = 1;
        }
    }

    // Step 6: Return the generated result for the current n
    return result;
};

```

## Solution

### 1. Base Case:

- Check if `n` is 1. If so, return the base case "1".

### 2. Recursive Call:

- Recursively call `countAndSay` for  $n - 1$  to get the previous result (`prevResult`).

### 3. Initialize Variables:

- Initialize variables `result` to store the current result and `count` to count the occurrences of a character.

### 4. Iterate Through Previous Result:

- Use a `for` loop to iterate through the characters of the previous result (`prevResult`).

### 5. Count Consecutive Characters:

- Check if the current character is the same as the next one.
- If true, increment the `count`.

### 6. Generate Current Result:

- If the current character is different from the next one:
  - Append the count and the current character to the result (`result`).
  - Reset the count for the new character.

### 7. Return Result:

- Return the generated result for the current  $n$ . This process continues recursively until the base case is reached.

## Count Complete Tree Nodes

### Count Complete Tree Nodes

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */

function countNodes(root: any | null): number {
  if (!root) {
    return 0;
  }

  const leftDepth = getLeftDepth(root);
  const rightDepth = getRightDepth(root);
```

```

if (leftDepth === rightDepth) {
    // If the left and right subtrees have the same depth, the tree is complete.
    return Math.pow(2, leftDepth) - 1;
}

// If the left and right subtrees have different depths, recursively count nodes.
return 1 + countNodes(root.left) + countNodes(root.right);
}

function getLeftDepth(node: any | null): number {
    let depth = 0;
    while (node) {
        depth++;
        node = node.left;
    }
    return depth;
}

function getRightDepth(node: any | null): number {
    let depth = 0;
    while (node) {
        depth++;
        node = node.right;
    }
    return depth;
}

```

**Solution:** This code defines a function `countNodes` to count the total number of nodes in a binary tree. The binary tree is represented using the `TreeNode` class, which has a `val` property for the node value and `left` and `right` properties for the left and right child nodes.

Breakdown:

#### 1. `countNodes` Function:

- The function takes a binary tree's root node as an argument (`root: TreeNode | null`).
- If the tree is empty (i.e., `root` is `null`), it returns 0.
- It then calculates the depth of the left and right subtrees using the helper functions `getLeftDepth` and `getRightDepth`.
- If the left and right subtrees have the same depth, the tree is complete, and it returns the total number of nodes using the formula  $(2^{\text{leftDepth}} - 1)$ .
- If the left and right subtrees have different depths, it recursively counts nodes in the left and right subtrees and adds 1 for the current node.

#### 2. `getLeftDepth` and `getRightDepth` Helper Functions:

- These functions take a node as an argument and return the depth of the left or right subtree, respectively.
- They use a while loop to traverse the tree and count the depth by moving to the left or right child nodes until reaching the deepest node.

The overall logic of the code is based on the observation that in a complete binary tree, if the left and right subtrees have the same depth, then the tree is complete, and the total number of nodes can be calculated directly. Otherwise, it recursively counts nodes in the left and right subtrees.

## Counting Bits

# Counting Bits

```
function countBits(n: number): number[] {  
    let result: number[] = [0];  
    let pow = 1;  
    let offset = 1;  
    for (let i = 1; i <= n; i++) {  
        if (i === pow) {  
            result[i] = 1;  
            pow = pow << 1;  
            offset = 1;  
        } else {  
            result[i] = result[offset] + 1;  
            offset++;  
        }  
    }  
    return result;  
};
```

### Solution:

Step-by-step breakdown of the code:

1. The `countBits` function takes an integer `n` as input and returns an array where each element represents the number of 1-bits in the binary representation of the corresponding index.
2. Initialize an array `result` with the first element set to 0. This array will store the count of 1-bits for each index.
3. Initialize variables `pow` and `offset` to keep track of the power of 2 and the current offset, respectively. Start with `pow` set to 1 and `offset` set to 1.
4. Iterate from `i = 1` to `n`.
5. Check if `i` is equal to `pow`. If it is, it means that `i` is a power of 2. In this case:
  - Set `result[i]` to 1, as a power of 2 always has a single 1-bit in its binary representation.
  - Update `pow` by left-shifting it by 1 to get the next power of 2.
  - Reset `offset` to 1.
6. If `i` is not a power of 2:
  - Set `result[i]` to `result[offset] + 1`. This means that the number of 1-bits in `i` is equal to the number of 1-bits in the number at the corresponding offset in the array, plus 1.
  - Increment `offset` by 1 to move to the next offset.
7. Return the `result` array containing the counts of 1-bits for each index.

Techniques used:

1. Bitwise Operations: The code utilizes the bitwise left shift operator (`<<`) to update the `pow` variable by shifting it to the left, effectively calculating the next power of 2.

2. Binary Representation: The solution leverages the binary representation of numbers to count the number of 1-bits at each index.

Summary:

The provided solution efficiently counts the number of 1-bits in the binary representation of each number from 0 to  $n$  and stores the counts in an array. It achieves this by identifying powers of 2 and updating the count based on the previously calculated counts using bitwise operations.

## Cumulative Sum

# Cumulative Sum

```
class Solution {  
    int[] getCumulativeSum (int[] arr) {  
        int[] out = new int[arr.length];  
        int total = 0;  
        for(int i = 0; i < arr.length; i++) {  
            total += arr[i];  
            out[i] = total;  
        }  
        return out;  
    }  
}
```

## Solution:

Step-by-step breakdown of the code:

1. The `getCumulativeSum` function takes an integer array `arr` as input and returns a new array `out` where each element represents the cumulative sum of elements up to that index.
2. Initialize a new integer array `out` with the same length as the input array `arr`.
3. Initialize a variable `total` to keep track of the running total. Set it to 0.
4. Iterate through each element of the input array `arr` using a for loop.
5. Inside the loop:
  - Add the current element `arr[i]` to the `total` variable, updating the running sum.
  - Assign the value of `total` to the corresponding index `out[i]` in the output array.
6. Repeat steps 5 for each element in the input array until the loop completes.
7. Return the resulting output array `out`, which contains the cumulative sum of elements up to each index.

Techniques used:

1. Array Manipulation: The code initializes and updates the elements of the `out` array to store the cumulative sum values.

2. Looping: The code utilizes a for loop to iterate through each element of the input array and perform the necessary calculations.
3. Accumulation: The code calculates the cumulative sum by continuously adding the current element to the running total.

Summary:

The provided solution efficiently calculates the cumulative sum of the input array by iteratively summing up the elements and storing the cumulative sum at each index in a new array. This approach allows for quick access to the cumulative sum at any given index.

## Curry function

# Curry function

```
function curry(func: any) {
  const curried = (...args: any) => {
    if(args.length >= func.length) {
      return func.apply(this, args);
    } else {
      return (...args2: any) => {
        return curried.apply(this, args.concat(args2))
      }
    }
  };
  return curried;
}
```

## Solution:

Step-by-step breakdown of the code:

1. The `curry` function takes a function `func` as input and returns a curried version of the function.
2. Inside the `curry` function, a new function `curried` is defined using arrow function syntax. This function will be the curried version of the original function.
3. The `curried` function accepts any number of arguments using the rest parameter syntax `...args`. These arguments will be passed to the original function `func`.
4. Inside the `curried` function, a check is performed to determine if the number of arguments passed (`args.length`) is greater than or equal to the expected number of arguments of the original function (`func.length`).
5. If the condition is true:
  - The original function `func` is called using the `apply` method, passing `this` as the context and `args` as the arguments.
  - The result of the function call is returned.
6. If the condition is false:
  - A new function is returned using arrow function syntax. This new function accepts additional arguments `args2`.

- The `curried` function is recursively called with the combined arguments of `args` and `args2` using the `apply` method.
7. The process of checking the number of arguments and either returning the result or returning a new function is repeated until the required number of arguments is met.
  8. The curried function `curried` is returned.

Techniques used:

1. Higher-Order Functions: The `curry` function takes a function `func` as input and returns a new function `curried` as the result.
2. Rest Parameter: The rest parameter syntax `...args` allows the `curried` function to accept any number of arguments and collect them into an array.
3. Arrow Function: The arrow function syntax is used to define the `curried` function, which simplifies the syntax and provides lexical scoping of `this`.
4. Conditional Statements: The code uses a conditional statement to check the number of arguments and determine whether to return the result or a new curried function.

Summary:

The provided solution implements the concept of currying by creating a curried version of a function. It allows the function to be called with a partial set of arguments, returning a new function that accepts the remaining arguments until the required number of arguments is provided.

## Customer Placing the Largest Number of Order

# Customer Placing the Largest Number of Order

```
SELECT customer_number
FROM Orders
GROUP BY customer_number
HAVING COUNT(order_number) = (
    SELECT COUNT(order_number)
    FROM Orders
    GROUP BY customer_number
    ORDER BY COUNT(order_number) DESC
    LIMIT 1
);
```

**Solution:** This SQL query retrieves the `customer_number` from the `Orders` table for customers who have placed the maximum number of orders. Here's a breakdown of the query:

### 1. Main Query:

- The main query selects the `customer_number` from the `Orders` table.
- It uses the `GROUP BY` clause to group the orders based on the `customer_number`.
- The `HAVING` clause filters the results to include only those groups where the count of `order_number` is equal to the maximum count of `order_number` in the entire table.

## 2. Subquery:

- The subquery calculates the count of `order_number` for each `customer_number`.
- It uses the `GROUP BY` clause to group the orders based on the `customer_number`.
- The `ORDER BY` clause orders the results by the count of `order_number` in descending order.
- The `LIMIT 1` clause ensures that only the row with the maximum count is selected.

The main idea is to compare the count of orders for each customer with the maximum count across all customers, and select only those customers whose order count matches the maximum.

### Customers Who Never Order

## Customers Who Never Order

```
SELECT Name AS Customers FROM Customers WHERE Id NOT IN( SELECT CustomerId FROM Orders);
```

### Solution:

Step-by-step breakdown of the SQL query:

1. The query selects the `Name` column from the `Customers` table and aliases it as `Customers`.
2. The `FROM` clause specifies the table `Customers` from which to retrieve the data.
3. The `WHERE` clause filters the results based on a condition.
4. The condition uses the `NOT IN` operator and a subquery to check if the `Id` column of the `Customers` table is not present in the `CustomerId` column of the `Orders` table.
5. The subquery selects the `CustomerId` column from the `Orders` table.
6. The `NOT IN` operator negates the condition, selecting only the rows where the `Id` column of `Customers` does not exist in the `CustomerId` column of `Orders`.
7. The query returns the result set with the selected `Name` column aliased as `Customers`.

Techniques used:

1. Column Selection: The query uses the `SELECT` statement to specify the `Name` column from the `Customers` table.
2. Table Specification: The `FROM` clause identifies the table `Customers` from which to retrieve the data.
3. Conditional Filtering: The `WHERE` clause filters the results based on a condition involving the `NOT IN` operator and a subquery.
4. Subquery: The subquery is used to retrieve the `CustomerId` column from the `Orders` table for comparison in the outer query.
5. Alias: The `AS` keyword is used to alias the selected `Name` column as `Customers` in the result set.

Summary:

The provided SQL query retrieves the names of customers from the `Customers` table who do not have any corresponding orders in the `Orders` table. It achieves this by using a subquery with the `NOT IN` operator to exclude customers whose `Id` values exist in the `CustomerId` column of the `Orders` table.

## Delete Duplicate Emails

```
delete A from Person A, Person B where A.id > B.id and A.email=B.email;
```

### Solution:

Step-by-step breakdown of the SQL query:

1. The query uses the `DELETE` statement to remove rows from the table `Person`.
2. The tables `Person A` and `Person B` are specified in the `FROM` clause, representing two instances of the `Person` table.
3. The `WHERE` clause is used to specify the condition for deleting rows.
4. The condition `A.id > B.id` compares the `id` column values between the two instances of the `Person` table.
5. The condition `A.email = B.email` compares the `email` column values between the two instances of the `Person` table.
6. The `DELETE` statement removes the rows from the `Person` table where both conditions are satisfied.

Techniques used:

1. Table Specification: The `FROM` clause identifies the tables `Person A` and `Person B` from which to delete rows.
2. Conditional Filtering: The `WHERE` clause filters the rows based on the specified conditions.
3. Comparison Operators: The `>` operator is used to compare the `id` column values, and the `=` operator is used to compare the `email` column values.
4. Delete Statement: The `DELETE` statement is used to remove the selected rows from the table.

### Summary:

The provided SQL query deletes rows from the `Person` table where the `id` value of one row is greater than the `id` value of another row, and their `email` values are the same. This ensures that only one row with the duplicate email address and the higher `id` value remains in the table.

## Detect Capital Use

## Detect Capital Use

```
function detectCapitalUse(word: string): boolean {
  const isUpperCase = (ch: string): boolean => ch === ch.toUpperCase();

  if (word === word.toUpperCase()) {
    return true;
```

```

    }

    if (word === word.toLowerCase()) {
        return true;
    }

    if (isUpperCase(word[0]) && word.slice(1) === word.slice(1).toLowerCase()) {
        return true;
    }

    return false;
};

```

### Solution:

Step-by-step breakdown of the code:

1. The function `detectCapitalUse` takes a `word` as input and returns a boolean value indicating if the capitalization of the word is correct.
2. The function defines an inner helper function `isUpperCase` which takes a character `ch` as input and returns `true` if the character is uppercase, and `false` otherwise.
3. The first condition checks if the entire `word` is uppercase. It uses the `word.toUpperCase()` method to convert the `word` to uppercase and compares it with the original `word`. If they are equal, it means that all characters in the word are uppercase, so the function returns `true`.
4. The second condition checks if the entire `word` is lowercase. It uses the `word.toLowerCase()` method to convert the `word` to lowercase and compares it with the original `word`. If they are equal, it means that all characters in the word are lowercase, so the function returns `true`.
5. The third condition checks if the first character of the `word` is uppercase (`isUpperCase(word[0])`) and the remaining characters (excluding the first character) are all lowercase (`word.slice(1) === word.slice(1).toLowerCase()`). If both conditions are true, it means that the capitalization of the word follows the correct pattern, so the function returns `true`.
6. If none of the above conditions are satisfied, it means that the capitalization of the word is incorrect, so the function returns `false`.

Techniques used:

1. Function Declaration: The code defines the function `detectCapitalUse` to encapsulate the capitalization checking logic.
2. Helper Function: The code defines an inner helper function `isUpperCase` to check if a character is uppercase.
3. Conditional Statements: The code uses conditional statements (`if` statements) to check different conditions and return the appropriate boolean value.
4. String Methods: The code uses the `toUpperCase()`, `toLowerCase()`, `slice()`, and `==` methods to manipulate and compare strings.

### Summary:

The provided JavaScript code checks the capitalization of a given word and returns `true` if the capitalization is correct (e.g., all uppercase, all lowercase, or starting with an uppercase followed by lowercase), and `false` otherwise.

## Diameter of Binary Tree

```
/**  
 * Definition for a binary tree node.  
 * class TreeNode {  
 *     val: number  
 *     left: TreeNode | null  
 *     right: TreeNode | null  
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {  
 *         this.val = (val===undefined ? 0 : val)  
 *         this.left = (left===undefined ? null : left)  
 *         this.right = (right===undefined ? null : right)  
 *     }  
 * }  
 */  
  
function diameterOfBinaryTree(root: TreeNode | null): number {  
    let diameter = 0;  
  
    function depth(node: TreeNode | null): number {  
        if (node === null) {  
            return 0;  
        }  
  
        const leftDepth = depth(node.left);  
        const rightDepth = depth(node.right);  
  
        diameter = Math.max(diameter, leftDepth + rightDepth);  
  
        return Math.max(leftDepth, rightDepth) + 1;  
    }  
  
    depth(root);  
  
    return diameter;  
}
```

### Solution:

Step-by-step breakdown of the code:

1. The function `diameterOfBinaryTree` takes a `root` node of the binary tree as input and returns the diameter of the tree.
2. The function initializes a variable `diameter` to 0. This variable will store the maximum diameter found during the depth traversal of the tree.
3. The function defines an inner helper function `depth` which calculates the depth of a given node in the binary tree.
4. The `depth` function takes a `node` as input and returns the depth (height) of the node.

5. If the `node` is `null`, indicating an empty subtree, the function returns 0.
6. The `depth` function recursively calculates the depth of the left and right subtrees of the current node using the `depth` function.
7. The `leftDepth` and `rightDepth` variables store the depths of the left and right subtrees, respectively.
8. The variable `diameter` is updated by comparing its current value with the sum of `leftDepth` and `rightDepth`. This step finds the maximum diameter among all nodes of the binary tree.
9. The function returns the maximum depth of the current node by adding 1 to the maximum of `leftDepth` and `rightDepth`. This represents the depth of the current subtree rooted at the current node.
10. The function `depth(root)` is called to start the recursive depth traversal from the root of the binary tree.
11. The function returns the calculated `diameter`, which represents the diameter of the binary tree.

Techniques used:

1. Function Declaration: The code defines the function `diameterOfBinaryTree` to calculate the diameter of the binary tree.
2. Helper Function: The code defines an inner helper function `depth` to calculate the depth (height) of a node in the binary tree.
3. Recursive Approach: The code uses a recursive approach to calculate the depth of the binary tree nodes and to find the maximum diameter.
4. Tree Traversal: The code traverses the binary tree to calculate the depth of each node and update the maximum diameter.

Summary:

The provided TypeScript code calculates the diameter of a binary tree by finding the maximum depth of each node and updating the maximum diameter accordingly. The function returns the calculated diameter value.

- Go back

## Distribute Candies

# Distribute Candies

```
function distributeCandies(candyType: number[]): number {
  const uniqueCandies = new Set(candyType);
  const maxCandies = candyType.length / 2;

  return Math.min(uniqueCandies.size, maxCandies);
};
```

**Solution:**

Step-by-step breakdown of the code:

1. The function `distributeCandies` takes an array `candyType` representing the types of candies available and returns the maximum number of unique candies the sister can receive.
2. The function creates a `Set` called `uniqueCandies` to store the unique types of candies available in the `candyType` array. The `Set` automatically eliminates duplicates, so it contains only unique candy types.
3. The variable `maxCandies` is assigned the value of half the length of the `candyType` array (`candyType.length / 2`). This represents the maximum number of candies the sister can receive if the candies are distributed equally between her and her brother.
4. The function returns the minimum value between the size of the `uniqueCandies` set and `maxCandies`. This step ensures that the sister receives the maximum possible unique candies while adhering to the constraint that she cannot receive more than half of the total candies.
5. The function returns the calculated maximum number of unique candies that the sister can receive.

Techniques used:

1. Data Structure - Set: The code uses a `Set` to store the unique types of candies available in the `candyType` array. The `Set` ensures that only unique candy types are present in the collection.
2. Mathematical Calculation: The code calculates the maximum number of candies the sister can receive (half of the total candies) and compares it with the number of unique candies available. The minimum value of these two is returned as the result.

Summary:

The provided TypeScript code calculates the maximum number of unique candies that the sister can receive when the candies are distributed equally between her and her brother. It ensures that she receives the maximum possible unique candies without exceeding half of the total candies.

## Divide Two Integers

### Divide Two Integers

```
function divide(dividend: number, divisor: number): number {
    let did = Math.abs(dividend);
    let dis = Math.abs(divisor);
    let sign = (divisor > 0 && dividend > 0) || (divisor < 0 && dividend < 0);
    let res = 0;
    let arr = [dis];

    if (dividend === 0 || did < dis) return 0;
    if (divisor === -1 && dividend === -2147483648) return 2147483647;
    if (dis === 1) return divisor > 0 ? dividend : -dividend;

    while (arr[arr.length - 1] < did) arr.push(arr[arr.length - 1] + arr[arr.length - 1]);

    for (var i = arr.length - 1; i >= 0; i--) {
        if (did >= arr[i]) {
            did -= arr[i];
            res += i === 0 ? 1 : 2 << (i - 1);
        }
    }
}
```

```
}

return sign ? res : -res;
};
```

### Solution:

#### 1. Get Absolute Values:

- Calculate the absolute values of `dividend` and `divisor` using `Math.abs()`.

#### 2. Determine Sign:

- Determine the sign of the result based on the signs of `dividend` and `divisor`.

#### 3. Initialize Result and Array:

- Initialize the result (`res`) to 0.
- Create an array `arr` containing the divisor.

#### 4. Handle Special Cases:

- Check if the dividend is 0 or if `did` (absolute value of dividend) is less than `dis` (absolute value of divisor). In these cases, return 0.
- Handle a special case where the divisor is -1 and the dividend is -2147483648, returning 2147483647.
- If the divisor is 1, return the dividend or its negation based on the sign of the divisor.

#### 5. Calculate Powers of Divisor:

- Use a while loop to calculate powers of the divisor and store them in the array until the last element is greater than or equal to `did`.

#### 6. Iterate Through Array to Calculate Result:

- Iterate through the array in reverse order.
- For each element in the array, subtract it from `did` if `did` is greater than or equal to the element.
- Update the result accordingly.

#### 7. Apply Sign and Return Result:

- Apply the determined sign to the result and return the final result.

### Duplicate Emails

## Duplicate Emails

```
select `Email` from `Person` group by `Email` having count(*) > 1
```

### Solution:

Step-by-step breakdown of the query:

1. **SELECT Email:** This part of the query selects the **Email** column from the **Person** table.
2. **FROM Person:** This part of the query specifies the source table, which is **Person** in this case.
3. **GROUP BY Email:** This part of the query groups the results based on the values in the **Email** column. It combines rows with the same email into a single group.
4. **HAVING COUNT(\*) > 1:** This part of the query filters the groups after the grouping has been done. It retains only those groups that have a count (number of occurrences) greater than 1. In other words, it selects only those email addresses that appear more than once in the **Person** table.
5. The query execution returns the **Email** values that satisfy the condition of having a count greater than 1.

Techniques used:

1. **SQL Query Language:** The code uses SQL (Structured Query Language) to interact with the database. SQL is a standard language used for querying and manipulating relational databases.
2. **SELECT Statement:** The query uses the **SELECT** statement to specify the columns to be retrieved from the database.
3. **GROUP BY Clause:** The query uses the **GROUP BY** clause to group the rows based on the values in the **Email** column.
4. **HAVING Clause:** The query uses the **HAVING** clause to filter the groups based on the condition that the count of rows in each group (**COUNT(\*)**) is greater than 1.

Summary:

The provided SQL query retrieves the **Email** values from the **Person** table, groups them by **Email**, and then selects only those groups where the **Email** occurs more than once in the table.

## **Employee Bonus**

# **Employee Bonus**

```
SELECT Employee.name, Bonus.bonus
FROM Employee
LEFT JOIN Bonus ON Employee.empId = Bonus.empId
WHERE Bonus.bonus < 1000 OR Bonus.bonus IS NULL;
```

**Solution:**

Step-by-step breakdown of the query:

1. **SELECT Employee.name, Bonus.bonus:** This part of the query selects the **name** column from the **Employee** table and the **bonus** column from the **Bonus** table.
2. **FROM Employee:** This part of the query specifies the source table, which is **Employee** in this case.
3. **LEFT JOIN Bonus ON Employee.empId = Bonus.empId:** This part of the query performs a **LEFT JOIN** between the **Employee** and **Bonus** tables based on the common column **empId**. The **LEFT JOIN** ensures that all rows from the left table (**Employee**) are included in the result, and matching rows from the right table (**Bonus**) are joined based on the specified condition.

4. WHERE Bonus.bonus < 1000 OR Bonus.bonus IS NULL: This part of the query filters the joined results. It includes only those rows where the **bonus** value from the **Bonus** table is less than 1000 or where the **bonus** value is **NULL**. This means that the query includes employees who have a bonus less than 1000 or employees who do not have any bonus (bonus value is **NULL**).
5. The query execution returns the **name** and **bonus** values that satisfy the condition of having a bonus less than 1000 or no bonus (**NULL**).

Techniques used:

1. SQL Query Language: The code uses SQL (Structured Query Language) to interact with the database. SQL is a standard language used for querying and manipulating relational databases.
2. SELECT Statement: The query uses the **SELECT** statement to specify the columns to be retrieved from the database.
3. LEFT JOIN: The query uses a **LEFT JOIN** to combine rows from two tables based on a common column (**empId**), including all rows from the left table (**Employee**) and matching rows from the right table (**Bonus**).
4. WHERE Clause: The query uses the **WHERE** clause to filter the rows based on the specified condition (bonus less than 1000 or **NULL**).

Summary:

The provided SQL query retrieves the **name** and **bonus** values from the **Employee** and **Bonus** tables, respectively, using a **LEFT JOIN**. It filters the results to include only employees with a bonus amount less than 1000 or employees who do not have any bonus (**NULL** value in the **bonus** column).

### **Employees Earning More Than Their Managers**

## **Employees Earning More Than Their Managers**

```
SELECT a.Name AS Employee
FROM Employee a, Employee b
WHERE a.ManagerId = b.Id AND a.Salary > b.Salary
```

### **Solution:**

Step-by-step breakdown of the query:

1. **SELECT a.Name AS Employee**: This part of the query selects the **Name** column from the **Employee** table and aliases it as **Employee**.
2. **FROM Employee a, Employee b**: This part of the query specifies the source table, which is **Employee**, and aliases it as **a** and **b**. The query performs a self-join on the **Employee** table by referencing it twice with different aliases (**a** and **b**).
3. **WHERE a.ManagerId = b.Id AND a.Salary > b.Salary**: This part of the query specifies the condition for the self-join. It selects only those rows where the **ManagerId** of an employee (**a**) matches the **Id** of another employee (**b**), and the salary of the employee (**a**) is greater than the salary of their respective manager (**b**). This effectively retrieves employees who have higher salaries than their managers.

4. The query execution returns the **Name** values of employees who meet the condition of having higher salaries than their managers.

Techniques used:

1. SQL Query Language: The code uses SQL (Structured Query Language) to interact with the database. SQL is a standard language used for querying and manipulating relational databases.
2. SELECT Statement: The query uses the **SELECT** statement to specify the column to be retrieved from the database and aliases the result column as **Employee**.
3. Self-Join: The query performs a self-join on the **Employee** table by referencing it twice with different aliases (**a** and **b**). This allows the query to compare employees with their respective managers.
4. WHERE Clause: The query uses the **WHERE** clause to specify the condition for the self-join, filtering the rows based on the specified criteria (salary comparison between employees and their managers).

Summary:

The provided SQL query retrieves the **Name** values of employees who have higher salaries than their respective managers by performing a self-join on the **Employee** table based on the **ManagerId** and **Salary** columns.

## Even Fibonacci Numbers

### Even Fibonacci Numbers

```
function fiboEvenSum(n: number): number {
  if (n <= 1) {
    return 0;
  } else {
    let evenSum: number = 0;
    let prevFibNum: number = 1;
    let fibNum: number = 2;

    while (fibNum <= n) {
      if (fibNum % 2 === 0) {
        evenSum += fibNum;
      }

      const nextFibNum: number = prevFibNum + fibNum;
      prevFibNum = fibNum;
      fibNum = nextFibNum;
    }

    return evenSum;
  }
}
```

#### Step-by-Step Breakdown:

1. Function Declaration:

- The function `fiboEvenSum` takes a single parameter `n` of type `number`.
- The function returns a value of type `number`.

## 2. Base Case Check:

- The function checks if `n` is less than or equal to 1. If true, it returns 0 since there are no even Fibonacci numbers less than or equal to 1.

## 3. Variable Initialization:

- `evenSum` is initialized to 0 to store the sum of even Fibonacci numbers.
- `prevFibNum` is initialized to 1, representing the previous Fibonacci number.
- `fibNum` is initialized to 2, representing the current Fibonacci number.

## 4. Fibonacci Sequence Generation and Summation:

- A `while` loop is used to generate Fibonacci numbers until the current Fibonacci number (`fibNum`) is less than or equal to `n`.
- Inside the loop, it checks if the current Fibonacci number is even. If it is, it adds the current Fibonacci number to the `evenSum`.
- The next Fibonacci number is generated by adding `prevFibNum` and `fibNum`. Then, the `prevFibNum` is updated to the previous `fibNum`, and `fibNum` is updated to the newly generated Fibonacci number.

## 5. Return Statement:

- Once the loop completes, the function returns the `evenSum`, which contains the sum of even Fibonacci numbers up to and including `n`.

**Even number of digits**

**Even number of digits**

```
class Solution {
    int countOfDigits(int number) {
        if(number == 0) {
            return 1;
        }
        int count = 0;
        while(number > 0) {
            count++;
            number /= 10;
        }
        return count;
    }

    List<Integer> getEvenDigitNumbers (int[] arr) {
        List<Integer> output = new ArrayList<Integer>();

        for(int i = 0; i < arr.length; i++) {
            if(countOfDigits(arr[i]) % 2 == 0) {
                output.add(arr[i]);
            }
        }
        return output;
    }
}
```

```

        }
    }
    return output;
}
}

```

### Solution:

Step-by-step breakdown of the code:

1. `int countOfDigits(int number)`: This method takes an integer `number` as input and calculates the number of digits in that integer.
2. `if(number == 0) { return 1; }`: This is a special case check. If the input number is 0, the method returns 1, as 0 has a single digit.
3. `int count = 0;`: This variable `count` is used to keep track of the number of digits in the input number.
4. `while(number > 0) { count++; number /= 10; }`: This loop iterates until the `number` becomes 0. In each iteration, the `count` is incremented, and the `number` is divided by 10 to remove the rightmost digit. This process continues until all the digits are counted.
5. `return count;`: Once the loop is completed, the method returns the `count`, which represents the number of digits in the input number.
6. `List<Integer> getEvenDigitNumbers (int[] arr)`: This method takes an array of integers `arr` as input and returns a list of integers that have an even number of digits.
7. `List<Integer> output = new ArrayList<Integer>();`: This initializes an `ArrayList` called `output` to store the integers with an even number of digits.
8. `for(int i = 0; i < arr.length; i++) {`: This loop iterates through each element in the input array.
9. `if(countOfDigits(arr[i]) % 2 == 0) { output.add(arr[i]); }`: This line checks if the number of digits in the current element `arr[i]` is even. If it is, the element is added to the `output` list.
10. After processing all elements in the input array, the method returns the `output` list, which contains the integers with an even number of digits.

Techniques used:

1. Java Programming Language: The code is written in the Java programming language, a widely used language for building various applications.
2. `ArrayList`: The `ArrayList` data structure from the `java.util` package is used to store the integers with an even number of digits.
3. Loop: The code uses a `while` loop to count the number of digits in the input integer and a `for` loop to iterate through the elements in the input array.
4. Conditional Statement: The `if` statement is used to check if the number of digits in an integer is even.

### Summary:

The provided Java code contains two methods: `countOfDigits`, which calculates the number of digits in an integer, and `getEvenDigitNumbers`, which returns a list of integers with an even number of digits from the input array.

## Excel Sheet Column Title

# Excel Sheet Column Title

```
function convertToTitle(columnNumber: number): string {
    let output: string[] = [];
    while (columnNumber > 0) {
        let j = columnNumber % 26;
        if (j === 0) {
            output.push("Z");
            columnNumber = Math.floor(columnNumber / 26) - 1;
        } else {
            output.push(String.fromCharCode((j - 1) + 'A'.charCodeAt(0)));
            columnNumber = Math.floor(columnNumber / 26);
        }
    }
    return output.reverse().join("");
};
```

### Solution:

Step-by-step breakdown of the code:

1. `let output: string[] = [];`: This initializes an empty array called `output` to store the characters of the Excel sheet column title.
2. `while (columnNumber > 0) {}`: This loop continues until the `columnNumber` becomes 0.
3. `let j: number = columnNumber % 26;`: This calculates the remainder `j` when `columnNumber` is divided by 26. The remainder represents the rightmost digit in the Excel sheet column title.
4. `if (j === 0) {}`: This condition checks if the remainder `j` is equal to 0. If it is, it means the rightmost digit should be 'Z', and the `columnNumber` needs to be adjusted accordingly.
5. `output.push("Z")`: If the remainder is 0, 'Z' is pushed into the `output` array.
6. `columnNumber = Math.floor(columnNumber / 26) - 1`: In this case, the `columnNumber` is divided by 26, and 1 is subtracted to adjust for the 'Z' digit. This step moves the `columnNumber` to the next left digit.
7. `else {}`: If the remainder `j` is not 0, this branch is executed.
8. `output.push(String.fromCharCode((j - 1) + 'A'.charCodeAt(0)))`: This line converts the current digit to its corresponding uppercase English letter and adds it to the `output` array. The calculation `(j - 1) + 'A'.charCodeAt(0)` maps the remainder `j` to the ASCII code of the corresponding uppercase English letter. The `String.fromCharCode()` method converts the ASCII code to a character.
9. `columnNumber = Math.floor(columnNumber / 26)`: This step moves the `columnNumber` to the next left digit.
10. After processing all the digits in the `columnNumber`, the loop ends, and the function returns the Excel sheet column title.
11. `return output.reverse().join("")`: The function reverses the `output` array (as we processed the digits from right to left) and joins the characters to form the final Excel sheet column title.

Summary:

The provided TypeScript function `convertToTitle` takes a column number as input and converts it to the corresponding Excel sheet column title represented by uppercase English letters.

Techniques used:

1. TypeScript: The code is written in the TypeScript programming language, which is a superset of JavaScript with added type annotations.
2. Loop: The code uses a `while` loop to process the digits in the `columnNumber` and convert them to the corresponding Excel sheet column title characters.
3. Array: The `output` array is used to store the characters of the Excel sheet column title before joining them into a string.
4. `String.fromCharCode()`: The `String.fromCharCode()` method is used to convert ASCII codes to characters and get the corresponding uppercase English letters.
5. `Math.floor()`: The `Math.floor()` method is used to round down the division result when calculating the next left digit in the `columnNumber`.

## Fibonacci Number

## Fibonacci Number

```
function fib(n: number): number {
  if (n <= 1) {
    return n;
  }

  let prev = 0;
  let curr = 1;

  for (let i = 2; i <= n; i++) {
    const temp = curr;
    curr = prev + curr;
    prev = temp;
  }

  return curr;
};
```

### Solution:

Step-by-step breakdown of the code:

1. `if (n <= 1) {`: This condition checks if the input `n` is less than or equal to 1. If `n` is 0 or 1, the function returns `n` itself as the Fibonacci number.
2. `let prev = 0;`: This initializes a variable `prev` to store the Fibonacci number from the previous iteration.
3. `let curr = 1;`: This initializes a variable `curr` to store the current Fibonacci number.

4. The loop `for (let i = 2; i <= n; i++)` starts from `i = 2` because we have already handled the cases when `n` is 0 or 1.
5. `const temp = curr;`: This creates a temporary variable `temp` to store the current `curr` value before updating it.
6. `curr = prev + curr;`: This updates the `curr` variable by adding the previous Fibonacci number (`prev`) to the current one (`curr`). This step calculates the next Fibonacci number in the sequence.
7. `prev = temp;`: This updates the `prev` variable with the previous `curr` value stored in `temp`. This prepares the `prev` variable for the next iteration.
8. The loop continues until `i` reaches the input `n`, and the Fibonacci numbers are calculated iteratively.
9. Finally, the function returns the value of `curr`, which is the nth Fibonacci number.

Summary:

The provided TypeScript function `fib` calculates the nth Fibonacci number using an iterative approach. It starts with `prev = 0` and `curr = 1`, then iteratively calculates the next Fibonacci number by adding the previous two numbers. If the input `n` is less than or equal to 1, the function returns `n` itself as the Fibonacci number.

Techniques used:

1. TypeScript: The code is written in the TypeScript programming language, which is a superset of JavaScript with added type annotations.
2. Loop: The code uses a `for` loop to iteratively calculate the Fibonacci numbers from 2 to `n`.
3. Variables: The variables `prev` and `curr` are used to store the current and previous Fibonacci numbers, respectively.
4. Iterative approach: The function uses an iterative method to calculate the Fibonacci numbers rather than a recursive approach.

## Fibonacci streaming

### Fibonacci streaming

```
function fibonacciSequence(): Iterator<number> {
    let index = 0;
    let prev = 0;
    let value = 1;

    return {
        next: () => {
            if (index >= Infinity) {
                return { value: value, done: true };
            }

            let result = { value, done: false };
            let next = value + prev;
            prev = value;
            value = next;
            index++;
            return result;
        }
    };
}
```

```

        index++;

        return result;
    }
}
}

```

### Solution:

Step-by-step breakdown of the code:

1. The function `fibonacciSequence` initializes three variables:
  - `index`: Represents the index of the current Fibonacci number in the sequence.
  - `prev`: Represents the previous Fibonacci number in the sequence.
  - `value`: Represents the current Fibonacci number in the sequence.
2. The function returns an object with a `next` method. This method is used to generate the next value in the Fibonacci sequence.
3. The `next` method checks if the `index` is greater than or equal to `Infinity`, which means the sequence has reached a very large value and should be terminated. In such a case, the method returns an object with the current `value` and `done` set to `true`, indicating the end of the sequence.
4. If the `index` is still within a reasonable range, the `next` method proceeds to calculate the next Fibonacci number.
5. The `next` method creates an object `result` with the current `value` and `done` set to `false`.
6. It calculates the next Fibonacci number (`next`) by adding the current `value` to the previous `prev`.
7. It updates the `prev` variable with the current `value` and the `value` variable with the calculated `next`.
8. It increments the `index` to move to the next position in the sequence.
9. The `next` method returns the `result` object, containing the current `value`, and the loop continues to generate the next Fibonacci number.

### Summary:

The provided TypeScript function `fibonacciSequence` returns an iterator that generates the Fibonacci sequence. It uses an object with a `next` method to produce the next Fibonacci number in the sequence. The `next` method continuously calculates the next Fibonacci number by adding the current value to the previous one and updates the variables accordingly.

### Techniques used:

1. TypeScript: The code is written in the TypeScript programming language, which is a superset of JavaScript with added type annotations.
2. Iterator: The function creates an iterator using an object with a `next` method to generate the Fibonacci sequence.
3. Fibonacci sequence: The function uses the Fibonacci sequence logic to calculate the next number by adding the previous two numbers.
4. Variables: The variables `index`, `prev`, and `value` are used to keep track of the current state and calculate the next Fibonacci number.
5. Infinite sequence: The function can generate Fibonacci numbers until the index reaches a very large value (`Infinity`).

## Find All Numbers Disappeared in an Array

# Find All Numbers Disappeared in an Array

```
function findDisappearedNumbers(nums: number[]): number[] {
    for (let i = 0; i < nums.length; i++) {
        let index = Math.abs(nums[i]) - 1;
        if (nums[index] > 0) {
            nums[index] = - nums[index];
        }
    }
    const result = [];
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] > 0) {
            result.push(i + 1);
        }
    }
    return result;
};
```

### Solution:

Step-by-step breakdown of the code:

1. The function `findDisappearedNumbers` takes an array `nums` as input, which contains integers.
2. The function uses an in-place approach to mark the visited numbers in the array.
3. In the first loop, it iterates through the `nums` array. For each element, it calculates the index where the corresponding number should be. It uses `Math.abs(nums[i]) - 1` to get the index (since the array is 0-indexed and the numbers are positive).
4. It checks if the value at the calculated index is positive. If it is, it means that the number at that index has not been visited yet. So, it updates the value at that index to its negative counterpart to mark it as visited.
5. After the first loop, all the numbers that appear in the array will be marked as negative, and the numbers that are not present will remain positive.
6. In the second loop, it iterates through the `nums` array again. For each element, if the value is positive, it means that the corresponding number is missing. So, it pushes `i + 1` (since the array is 0-indexed) to the `result` array.
7. After both loops, the `result` array will contain the numbers that are missing from the original array.
8. The function returns the `result` array containing the disappeared numbers.

### Summary:

The provided TypeScript function `findDisappearedNumbers` finds the disappeared numbers from an array of integers by using an in-place approach. It iterates through the array, marks the visited numbers by updating their values to negative, and then finds the missing numbers by looking for positive values in the array.

Techniques used:

1. TypeScript: The code is written in the TypeScript programming language, which is a superset of JavaScript with added type annotations.
2. Array manipulation: The function modifies the input array to mark visited numbers and then finds the missing numbers in a second loop.
3. Negative numbers as markers: The function uses negative values as markers to keep track of visited numbers in the array.

## Find Customer Referee

### Find Customer Referee

```
SELECT name
FROM customer
WHERE referee_id IS NULL OR referee_id != 2;
```

#### Solution:

This SQL query is used to retrieve the names of customers from the `customer` table based on certain conditions.

#### Query breakdown:

1. `SELECT name`: This part of the query specifies that we want to retrieve the `name` column from the `customer` table.
2. `FROM customer`: This part of the query specifies the table from which we want to retrieve the data, which is the `customer` table.
3. `WHERE referee_id IS NULL OR referee_id != 2`: This part of the query specifies the condition for filtering the data. The query retrieves customer names where either of the following conditions is true:
  - `referee_id` is `NULL` (i.e., the customer does not have a referee).
  - `referee_id` is not equal to `2` (i.e., the customer's referee is not with an ID of `2`).

#### Summary:

The SQL query retrieves the names of customers from the `customer` table where either the `referee_id` is `NULL` (indicating no referee) or the `referee_id` is not equal to `2` (indicating a different referee from the one with an ID of `2`).

Techniques used:

1. SQL SELECT statement: The query uses the `SELECT` statement to specify the columns to retrieve from the table.
2. SQL WHERE clause: The query uses the `WHERE` clause to specify the conditions for filtering the data.

## Find Mode in Binary Search Tree

### Find Mode in Binary Search Tree

```

/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */

class ModeCounter {
    maxCount: number;
    currentCount: number;
    currentVal: number;
    modes: number[];

    constructor() {
        this.maxCount = 0;
        this.currentCount = 0;
        this.currentVal = 0;
        this.modes = [];
    }

    update(val: number) {
        if (val === this.currentVal) {
            this.currentCount++;
        } else {
            this.currentVal = val;
            this.currentCount = 1;
        }

        if (this.currentCount > this.maxCount) {
            this.maxCount = this.currentCount;
            this.modes = [this.currentVal];
        } else if (this.currentCount === this.maxCount) {
            this.modes.push(this.currentVal);
        }
    }
}

function findMode(root: TreeNode | null): number[] {
    const counter = new ModeCounter();
    let current = root;
    let prev: TreeNode | null = null;

    while (current !== null) {
        if (current.left === null) {
            counter.update(current.val);
            current = current.right;
        }
    }
}

```

```

    } else {
        prev = current.left;

        while (prev.right !== null && prev.right !== current) {
            prev = prev.right;
        }

        if (prev.right === null) {
            prev.right = current;
            current = current.left;
        } else {
            prev.right = null;
            counter.update(current.val);
            current = current.right;
        }
    }
}

return counter.modes;
}

```

### Solution:

The provided TypeScript code is used to find the mode(s) in a binary search tree. The binary search tree (BST) is defined using the `TreeNode` class, which has a value (`val`) and two children (`left` and `right`). The code implements an iterative approach to find the mode(s) in the BST.

#### Class ModeCounter:

- `maxCount`: Keeps track of the maximum frequency of a value seen so far.
- `currentCount`: Keeps track of the frequency of the current value being processed.
- `currentVal`: Keeps track of the current value being processed.
- `modes`: An array to store the mode(s) found so far.

#### Method `update(val: number):`

This method is used to update the `ModeCounter` object when a new value `val` is encountered while traversing the BST.

- If the value `val` is equal to the current value (`currentVal`), increment the `currentCount`.
- If the value `val` is different from the current value (`currentVal`), update `currentVal` to `val` and set `currentCount` to 1.
- If the `currentCount` becomes greater than `maxCount`, update `maxCount` to `currentCount`, and reset the `modes` array with the current value `val`.
- If the `currentCount` is equal to `maxCount`, add the current value `val` to the `modes` array since it is another mode.

#### Function `findMode(root: TreeNode | null): number[]:`

This function takes the root of the binary search tree (`root`) and returns an array containing the mode(s) of the BST.

- Create a `ModeCounter` object named `counter` to keep track of the modes and their frequencies.
- Initialize `current` to `root` and `prev` to `null`.

- Use a while loop to traverse the tree using Morris Inorder Traversal:
  - If `current` does not have a left child, update the `counter` with the value of `current`, and move to its right child.
  - If `current` has a left child, find its inorder predecessor (rightmost node in the left subtree) named `prev`:
    - \* If `prev.right` is `null`, set `prev.right` to `current`, and move to the left child of `current`.
    - \* If `prev.right` is `current`, it means we have visited the left subtree, so set `prev.right` to `null`, update the `counter` with the value of `current`, and move to its right child.
- Return the `modes` array from the `counter`, which contains the mode(s) of the BST.

### **Summary:**

The given code implements an iterative Morris Inorder Traversal approach to find the mode(s) in a binary search tree. It uses the `ModeCounter` class to keep track of the modes and their frequencies during the traversal.

### **Techniques used:**

1. Binary Search Tree (BST): The code works with a binary search tree data structure, where each node's left child is less than the node, and each node's right child is greater than the node.
  2. Morris Inorder Traversal: This approach allows for traversing the BST without using recursion or an explicit stack, reducing space complexity to O(1). It helps find the mode(s) efficiently while maintaining the order of traversal.
- Go back

### **Find Smallest Multiple**

## **Find Smallest Multiple**

```
function findSmallestMultiple(n: number): number {
  let lcm = 1;
  for (let i = 2; i <= n; i++) {
    lcm = (lcm * i) / gcd(lcm, i);
  }
  return lcm;
}

function gcd(a: number, b: number): number {
  if (b === 0) {
    return a;
  } else {
    return gcd(b, a % b);
  }
}
```

### **findSmallestMultiple Function:**

1. Function Definition:

- The `findSmallestMultiple` function takes a parameter `n`, which is assumed to be a positive integer.

## 2. Initialization:

- Initialize a variable `lcm` to 1. This variable will store the least common multiple of numbers from 1 to `n`.

## 3. Loop Through Numbers:

- The loop starts from `i = 2` and goes up to `n`.

## 4. Calculate LCM:

- In each iteration, update `lcm` using the formula:  

$$lcm = (lcm * i) / gcd(lcm, i)$$
- Here, `gcd(lcm, i)` calculates the greatest common divisor (GCD) of `lcm` and `i`.
- The formula calculates the least common multiple using the relationship:  $LCM(a, b) = (a * b) / GCD(a, b)$ .

## gcd Function:

### 1. Function Definition:

- The `gcd` function takes two parameters, `a` and `b`, which are assumed to be non-negative integers.

### 2. GCD Calculation:

- If `b` is 0, return `a` as the GCD.
- Otherwise, recursively call the `gcd` function with arguments `b` and `a % b`. This continues until `b` becomes 0, at which point `a` is the GCD.

## Find the Difference

### Find the Difference

```
function findTheDifference(s: string, t: string): string {
    let sArr = s.split('').sort()
    let tArr = t.split('').sort()
    for( let i = 0; i < tArr.length; i++){
        if( sArr[i] !== tArr[i]){
            return tArr[i];
        }
    }
};
```

#### Solution:

The provided function `findTheDifference` takes two strings `s` and `t` as input and returns a string representing the character that is added to `t` and not present in `s`. In other words, it finds the difference between the two strings.

#### Steps:

1. Create two arrays, **sArr** and **tArr**, by splitting the input strings **s** and **t** into individual characters and sorting them.
2. Use a loop to iterate through the characters of the **tArr** array.
3. Inside the loop, compare the characters at the same index in **sArr** and **tArr**. If they are not equal, it means the current character in **tArr** is the added character (difference).
4. Return the character that is different (added) in **tArr**.

**Example:**

- Input: **s** = "abcd", **t** = "abcde"
- After splitting and sorting, **sArr** = ['a', 'b', 'c', 'd'] and **tArr** = ['a', 'b', 'c', 'd', 'e'].
- The loop iterates through the arrays, and when it reaches the last element, **sArr[i]** is 'd', and **tArr[i]** is 'e'. Since 'd' is not equal to 'e', the function returns 'e'.

**Note:**

The given code assumes that the input strings **s** and **t** are valid, and **t** is formed by adding a single character to **s**.

**Techniques used:**

1. Array Manipulation: The code splits the input strings **s** and **t** into arrays of characters using the **split()** function and then sorts the arrays.
2. Looping: The function uses a **for** loop to iterate through the characters of the **tArr** array and compare them with the characters of the **sArr** array.
3. String Comparison: The code compares characters at the same index in **sArr** and **tArr** to find the difference.

### Find the Index of the First Occurrence in a String

## Find the Index of the First Occurrence in a String

```
function strStr(haystack: string, needle: string): number {
    const haystackLength = haystack.length;
    const needleLength = needle.length;

    for(let i = 0; i < haystackLength - needleLength + 1; i++) {
        if (haystack.substring(i, i + needleLength) === needle) {
            return i;
        }
    }
    return -1;
};
```

**Solution:**

The provided function **strStr** takes two strings **haystack** and **needle** as input and returns the index of the first occurrence of the **needle** in the **haystack**. If the **needle** is not found in the **haystack**, the function returns -1.

**Steps:**

1. Calculate the lengths of the **haystack** and the **needle** and store them in variables **haystackLength** and **needleLength**, respectively.
2. Use a **for** loop to iterate through the **haystack**, starting from index 0 and stopping at **haystackLength - needleLength** (inclusive).
3. Inside the loop, extract a substring from the **haystack** starting from the current index **i** and with a length equal to the length of the **needle**.
4. Compare the extracted substring with the **needle**. If they are equal, it means the **needle** is found at index **i** in the **haystack**, so return **i**.
5. If the loop finishes without finding the **needle**, return -1 to indicate that the **needle** is not present in the **haystack**.

**Example:**

- Input: **haystack** = "hello", **needle** = "ll"
- The loop iterates through the **haystack** string:
  - When **i** = 0, the extracted substring is "he" which is not equal to the **needle**.
  - When **i** = 1, the extracted substring is "el" which is not equal to the **needle**.
  - When **i** = 2, the extracted substring is "ll" which is equal to the **needle**. The function returns 2.

**Note:**

The given code assumes that the inputs **haystack** and **needle** are valid strings.

**Techniques used:**

1. Looping: The function uses a **for** loop to iterate through the **haystack** and search for the **needle**.
2. String Manipulation: The code extracts substrings using the **substring()** function and compares them with the **needle** using the equality operator (**==**).

**Find the median**

**Find the median**

```
function findMedian(arr: number[]): number {
  arr = arr.sort((a,b) => a - b);
  return arr[Math.floor(arr.length / 2)];
}
```

**Solution:**

The function **findMedian** takes an array of numbers **arr** as input and returns the median of the array.

**Steps:**

1. Sort the input array **arr** in ascending order using the **sort** method with a custom comparison function **(a, b) => a - b**. This ensures that the numbers in the array are arranged in increasing order.

2. Calculate the index of the median element by dividing the length of the sorted array by 2 using `Math.floor(arr.length / 2)`.
3. Return the element at the calculated index, which is the median of the array.

**Example:**

- Input: `arr = [3, 1, 5, 2, 4]`
- After sorting the array: `[1, 2, 3, 4, 5]`
- Median index = `Math.floor(5 / 2) = 2`
- Median element at index 2 is 3.
- The function returns 3.

**Note:**

The given code assumes that the input array `arr` is not empty and contains valid numbers.

**Techniques used:**

1. Sorting: The function uses the `sort` method to sort the input array in ascending order.
2. Mathematical Operation: The code uses `Math.floor` to calculate the index of the median element.

**Find the Pivot Integer**

## Find the Pivot Integer

```
function pivotInteger(n: number): number {
    let total = (n * (n + 1)) / 2;
    let sum = 0;
    for(let i = 1; i <= n; i++) {
        sum += i;
        if(sum === (total - sum + i)) {
            return i;
        }
    }
    return -1;
};
```

**Solution:**

The provided function `pivotInteger` takes an integer `n` as input and finds the pivot point (also known as equilibrium point) in a sequence of consecutive integers from 1 to `n` (inclusive). The pivot point is the integer at which the sum of all integers before it is equal to the sum of all integers after it.

**Steps:**

1. Calculate the total sum of all integers from 1 to `n` using the formula `total = (n * (n + 1)) / 2`.
2. Initialize a variable `sum` to 0, which will keep track of the sum of integers encountered while iterating.
3. Iterate through integers from 1 to `n` (inclusive) using a for loop.
4. In each iteration, add the current integer `i` to the `sum` variable.

5. Check if the current sum is equal to `total - sum + i`. If it is, this means the current integer is the pivot point, and it divides the sequence into two parts with equal sums.
6. If a pivot point is found, return the value of `i`.
7. If no pivot point is found after the loop, return `-1`.

**Example:**

- Input: `n = 7`
- Total sum of integers from 1 to 7:  $(7 * (7 + 1)) / 2 = 28$
- Iterate through the integers:
  - `i = 1: sum = 1`, check if `sum === (28 - 1 + 1)`, not equal
  - `i = 2: sum = 3`, check if `sum === (28 - 3 + 2)`, not equal
  - `i = 3: sum = 6`, check if `sum === (28 - 6 + 3)`, not equal
  - `i = 4: sum = 10`, check if `sum === (28 - 10 + 4)`, not equal
  - `i = 5: sum = 15`, check if `sum === (28 - 15 + 5)`, not equal
  - `i = 6: sum = 21`, check if `sum === (28 - 21 + 6)`, not equal
  - `i = 7: sum = 28`, check if `sum === (28 - 28 + 7)`, not equal
- No pivot point is found, so the function returns `-1`.

**Note:**

The given code assumes that the input `n` is a positive integer.

**Techniques used:**

1. **Iteration:** The function iterates through integers from 1 to `n` (inclusive) using a for loop to calculate the sum and find the pivot point.
2. **Mathematical Operation:** The function uses mathematical formulas to calculate the total sum of integers from 1 to `n`.
3. **Conditional Statement:** The function uses a conditional statement (`if` statement) to check if the current sum is equal to `total - sum + i`, which helps identify the pivot point.
4. **Variable Manipulation:** The function uses variables such as `total`, `sum`, and `i` to keep track of the calculated values during the iteration and calculation process.
5. **Comparison Operator:** The function uses the equality comparison operator (`==`) to compare the sum with the calculated value and determine if a pivot point is found.
6. **Returning a Value:** The function returns a value (`i`) when a pivot point is found, or it returns `-1` if no pivot point is found.
7. **Arithmetic Operators:** The function uses arithmetic operators (`+, -, /`) to calculate the total sum and update the `sum` variable.
8. **Math.floor Function:** Although not explicitly used in the provided code, the `Math.floor` function is commonly used when dealing with floating-point values to ensure we get the nearest lower integer value.

These techniques collectively enable the function to efficiently find the pivot point in the sequence of consecutive integers from 1 to `n`.

## First Unique Character in a String

### First Unique Character in a String

```
function firstUniqChar(s: string): number {
    let map = [];
    for (let i = 0; i < s.length; i++) {
        if(map[s[i]]) {
            map[s[i]]++;
        } else {
            map[s[i]] = 1;
        }
    }
    for (let i = 0; i < s.length; i++) {
        if (map[s[i]] === 1) {
            return i;
        }
    }
    return -1;
};
```

1. Create an empty array `map` to store character counts. This array will be used as a simple hash map.
2. Iterate through the characters of the input string `s` using a `for` loop.
3. Check if the current character `s[i]` already exists in the `map` array. This is done by using the character `s[i]` as the index in the `map` array.
4. If the character already exists in the `map`, increment its count by 1. This is done by using the post-increment operator `map[s[i]]++`.
5. If the character does not exist in the `map`, set its count to 1. This is done by assigning `1` to `map[s[i]]`.
6. After the loop, we have an array `map` with counts of each character in the input string `s`.
7. Now, iterate through the characters of the input string `s` again using another `for` loop.
8. For each character `s[i]`, check its count in the `map` array.
9. If the count is equal to `1`, it means that this character is unique. Return the index `i` as the first occurrence of the unique character.
10. If no unique character is found during the second loop, return `-1` to indicate that there are no unique characters in the input string `s`.

**Techniques used in this code:** - Using an array as a simple hash map to store character counts. - Iterating through a string using a for loop to perform character counts and find the first unique character.

## First Bad Version

### First Bad Version

```

/**
 * The knows API is defined in the parent class Relation.
 * isBadVersion(version: number): boolean {
 *     ...
 * };
 */

```

var solution = function(isBadVersion: any) {

```

    return function(n: number): number {
        let left = 1, right = n
        while(left < right) {
            const mid = left + Math.floor((right-left)/2)
            if(isBadVersion(mid)) {
                right = mid
            } else {
                left = mid + 1
            }
        }
        return left
    };
}

```

This is a classic example of the Binary Search algorithm. Let's break down the code step by step:

1. The function **solution** takes a parameter **isBadVersion**, which is a function that returns a boolean value indicating whether a given version is a bad version or not.
2. The **solution** function returns an anonymous function that takes a parameter **n**, which represents the total number of versions.
3. Inside the anonymous function, two variables **left** and **right** are initialized with **1** and **n** respectively. These variables will be used to define the search space for the bad version.
4. A while loop is used to perform binary search on the versions.
5. In each iteration of the loop, the midpoint **mid** of the current search space is calculated using **left + Math.floor((right - left) / 2)**.
6. The **isBadVersion** function is called with the **mid** version to determine if it is a bad version or not.
7. If **isBadVersion(mid)** returns **true**, it means the bad version is in the first half of the search space. So, the **right** pointer is updated to **mid** to narrow down the search space.
8. If **isBadVersion(mid)** returns **false**, it means the bad version is in the second half of the search space. So, the **left** pointer is updated to **mid + 1** to narrow down the search space.
9. The loop continues until **left** is no longer less than **right**. At this point, the search space has been narrowed down to a single version, which is the first bad version.
10. The function returns the value of **left**, which represents the first bad version.

Techniques used in this code:

- Binary Search: The algorithm uses binary search to find the first bad version among a given range of versions. The search space is repeatedly halved until the first bad version is found.

- Mathematical calculation: The midpoint `mid` is calculated as `left + Math.floor((right - left) / 2)` to avoid integer overflow and provide the correct `mid` value during the binary search.

Source: <https://leetcode.com>

## Fizz Buzz

### Fizz Buzz

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

The function `fizzBuzz` takes an integer `n` as input and returns an array of strings with the following conditions:

1. If the number is divisible by 3 and 5 (i.e., a multiple of 15), it adds "FizzBuzz" to the array.
2. If the number is divisible by 3 (i.e., a multiple of 3), it adds "Fizz" to the array.
3. If the number is divisible by 5 (i.e., a multiple of 5), it adds "Buzz" to the array.
4. Otherwise, it adds the number itself as a string to the array.

Here's a step-by-step explanation of the code:

1. Initialize an empty array called `output` to store the result.
2. Use a `for` loop to iterate from 1 to `n`, inclusive.
3. Inside the loop, check the following conditions:
  - If the current number `i` is divisible by both 3 and 5 (i.e., `i % 15 === 0`), add "FizzBuzz" to the `output` array.
  - Else, if the current number `i` is divisible by 3 (i.e., `i % 3 === 0`), add "Fizz" to the `output` array.
  - Else, if the current number `i` is divisible by 5 (i.e., `i % 5 === 0`), add "Buzz" to the `output` array.
  - If none of the above conditions are true, add the current number `i` as a string to the `output` array.
4. After the loop completes, return the `output` array containing the FizzBuzz sequence for the numbers from 1 to `n`.

The technique used in this code is:

- Iteration: The code uses a `for` loop to iterate through the numbers from 1 to `n`, applying the FizzBuzz conditions and adding the corresponding strings to the `output` array.

## Game Play Analysis I

### Game Play Analysis I

```
SELECT player_id, MIN(event_date) AS first_login
FROM Activity
GROUP BY player_id
```

In the given SQL query, we are selecting the `player_id` and the minimum `event_date` (earliest login date) for each player from the `Activity` table. Here's a step-by-step explanation of the query:

1. `SELECT player_id, MIN(event_date) AS first_login`: This part of the query specifies the columns we want to select. We want to retrieve the `player_id` and the minimum (`MIN`) value of the `event_date`. We also use the `AS` keyword to give the minimum `event_date` column an alias name of `first_login`.
2. `FROM Activity`: This part of the query specifies the table we are selecting data from, which is the `Activity` table.

The query will return a result set with two columns: `player_id` and `first_login`. Each row in the result set will represent a unique `player_id` along with their earliest login date (`first_login`) recorded in the `Activity` table.

Technique used in the query:

- Aggregate function (`MIN`): The `MIN` function is used to find the minimum value of the `event_date` column for each `player_id`. It helps to determine the earliest login date for each player.

## Generate Parentheses

### Generate Parentheses

```
function generateParenthesis(n: number): string[] {
    const result = [];
    generator(result, "", 0, 0, n)
    return result;
};

function generator(result, s, open, close, n) {
    if(open === n && close === n) {
        result.push(s);
        return;
    }
    if(open < n) {
        generator(result, s + '(', open + 1, close, n);
    }
    if(close < open) {
        generator(result, s + ')', open, close + 1, n);
    }
}
```

The code is an implementation of a function to generate all valid combinations of parentheses given a positive integer  $n$ . The function `generateParenthesis` takes an integer  $n$  as input and returns an array of strings containing all valid combinations of parentheses.

Here's a step-by-step explanation of the code:

1. `function generateParenthesis(n: number): string[]`: This is the main function that takes an integer  $n$  as input and returns an array of strings representing all valid combinations of parentheses.
2. `const result = []`: This initializes an empty array `result` to store the generated combinations of parentheses.
3. `generator(result, "", 0, 0, n)`: This line calls the `generator` function to start generating valid combinations of parentheses. The initial values for `s` (current combination), `open` (number of open parentheses), `close` (number of close parentheses), and `n` (total number of parentheses to be generated) are passed as arguments.
4. `function generator(result, s, open, close, n)`: This is a recursive function that generates valid combinations of parentheses.
5. `if (open === n && close === n)`: This condition checks if we have generated the desired number of open and close parentheses, which is equal to  $n$ . If both are equal to  $n$ , it means we have a valid combination of parentheses, and it is added to the `result` array.
6. `if (open < n)`: This condition checks if the number of open parentheses is less than  $n$ . If true, we call the `generator` function recursively with an additional open parenthesis added to the current combination.
7. `if (close < open)`: This condition checks if the number of close parentheses is less than the number of open parentheses. If true, we call the `generator` function recursively with an additional close parenthesis added to the current combination.
8. The function will continue generating combinations of parentheses until all valid combinations are found.
9. Finally, the `result` array containing all valid combinations of parentheses is returned as the output.

Techniques used in the code:

- Recursion: The function `generator` is implemented using recursion to explore all possible combinations of parentheses.
- Backtracking: The recursion and condition checks for open and close parentheses help in generating valid combinations and avoiding invalid ones.

## Guess Number Higher or Lower

### Guess Number Higher or Lower

```
/**  
 * Forward declaration of guess API.  
 * @param {number} num    your guess  
 * @return       -1 if num is higher than the picked number  
 *               1 if num is lower than the picked number  
 *               otherwise return 0  
 */
```

```

/* var guess = function(num) {} */
*/



function guessNumber(n: number): number {
    let left = 1, right = n

    while(left < right) {
        const mid = left + Math.floor((right-left)/2)
        const current = guess(mid)
        if(current === 0) return mid
        if(current === -1) right = mid
        else left = mid+1
    }
    return left
};

```

The code is an implementation of a binary search algorithm to find the number that is guessed correctly within the range of numbers from 1 to `n`. The `guessNumber` function takes an integer `n` as input and returns the correct number.

Here's a step-by-step explanation of the code:

1. `function guessNumber(n: number): number`: This is the main function that takes an integer `n` as input and returns the correct number guessed within the range from 1 to `n`.
2. `let left = 1, right = n`: This initializes two variables `left` and `right` to represent the leftmost and rightmost boundaries of the search range, respectively. Initially, the range is from 1 to `n`.
3. `while (left < right)`: This is a loop that continues until the `left` and `right` boundaries meet or cross each other.
4. `const mid = left + Math.floor((right - left) / 2)`: This calculates the middle index of the current search range.
5. `const current = guess(mid)`: This calls the `guess` API with the `mid` value to check if the guessed number is correct or not. The `guess` API returns -1 if the guessed number is higher than the picked number, 1 if it's lower, and 0 if the guess is correct.
6. `if (current === 0) return mid`: If the guess is correct (i.e., `current === 0`), we found the correct number, and the function returns `mid`.
7. `if (current === -1) right = mid`: If the guessed number is higher than the picked number (i.e., `current === -1`), we update the `right` boundary to `mid`. This means we narrow down the search range to the left half.
8. `else left = mid + 1`: If the guessed number is lower than the picked number (i.e., `current === 1`), we update the `left` boundary to `mid + 1`. This means we narrow down the search range to the right half.
9. The loop continues, and the search range is reduced in each iteration.
10. When the loop ends, the correct number has been found, and the function returns `left`, which holds the correct number.

Techniques used in the code:

- Binary search: The function `guessNumber` uses binary search to efficiently find the correct number by narrowing down the search range in each iteration.

## Hamming Distance

# Hamming Distance

```
class Solution {
    public int hammingDistance(int x, int y) {
        int xor = x ^ y;
        int count = 0;
        while (xor > 0) {
            if ((xor & 1) == 1) {
                count++;
            }
            xor = xor >> 1;
        }
        return count;
    }
}
```

The code is a Java solution to calculate the Hamming distance between two integers **x** and **y**. The Hamming distance between two integers is the number of positions at which the corresponding bits are different.

Here's a step-by-step explanation of the code:

1. **public int hammingDistance(int x, int y):** This is the method that calculates the Hamming distance between the two integers **x** and **y**.
2. **int xor = x ^ y;:** This calculates the bitwise XOR of **x** and **y** and stores the result in the variable **xor**. The XOR operation results in a number where the bits are set to 1 if the corresponding bits in **x** and **y** are different.
3. **int count = 0;:** This initializes a variable **count** to keep track of the number of set bits (1s) in the **xor** variable.
4. **while (xor > 0) {:** This is a loop that continues as long as the **xor** variable is greater than 0.
5. **if ((xor & 1) == 1) {:** This checks if the least significant bit (LSB) of the **xor** variable is 1 (i.e., the rightmost bit is 1). If it is, it means that the corresponding bits in **x** and **y** are different, and the Hamming distance should be increased.
6. **count++;:** If the LSB is 1, we increment the **count** variable to keep track of the number of different bits.
7. **xor = xor >> 1;:** This right-shifts the **xor** variable by 1 bit, effectively moving to the next bit for comparison in the next iteration of the loop.
8. The loop continues, and the **xor** variable is right-shifted in each iteration to check each bit.
9. When the loop ends, the **count** variable holds the Hamming distance, which represents the number of positions at which the bits in **x** and **y** are different.
10. The method returns the **count**, which is the Hamming distance between **x** and **y**.

Techniques used in the provided solution:

1. Bitwise XOR: The code uses the bitwise XOR (`^`) operation to find the differences between the bits of two integers `x` and `y`. The XOR operation sets the corresponding bits to 1 if they are different between `x` and `y`, and 0 if they are the same.
2. Bitwise AND: The code uses the bitwise AND (`&`) operation in the line `if ((xor & 1) == 1)` to check the least significant bit (LSB) of the `xor` variable. The AND operation with 1 extracts the value of the rightmost bit in `xor`.
3. Right Shift: The code uses right-shift (`>>`) to move to the next bit in the `xor` variable. It right-shifts the bits by one position in each iteration of the loop, allowing the code to check all the bits in `xor`.
4. Counting Set Bits: The code counts the number of set bits (1s) in the `xor` variable using bitwise operations. It increments the `count` variable whenever a set bit is found in `xor`.

## Happy Number

# Happy Number

```
function isHappy(n: number): boolean {
    let slow = n;
    let fast = getNextSum(n);
    while(fast !== 1 && slow !== fast) {
        slow = getNextSum(slow);
        fast = getNextSum(getNextSum(fast));
    }
    return fast === 1;
};

const getNextSum = (number) => {
    let sum = 0;
    while(number > 0) {
        let digit = number % 10;
        sum += digit * digit;
        number = Math.floor(number / 10);
    }
    return sum;
}
```

The code is an implementation of the "Happy Number" problem. It determines if a given number is a "happy number" or not. A happy number is defined as a number in which the sum of the squares of its digits eventually becomes 1 after a series of calculations. If the sum never becomes 1, it is not a happy number.

Here's a step-by-step explanation of the code:

1. The `isHappy` function takes an integer `n` as input and returns a boolean indicating whether `n` is a happy number or not.
2. Two variables `slow` and `fast` are initialized with the value of `n`. These variables will be used to detect cycles in the sequence of sums of squares of digits.
3. The `getNextSum` function calculates the sum of the squares of the digits of a given number. It takes an integer `number` as input and returns the sum.

4. In the `isHappy` function, a loop is used to calculate the next sum of squares of digits for both `slow` and `fast` in each iteration. The loop continues until `fast` becomes 1 or until `fast` becomes equal to `slow`, indicating the presence of a cycle.
5. In each iteration of the loop, `slow` is updated by calculating the next sum of squares of digits using the `getNextSum` function.
6. `fast` is updated by calculating two consecutive sums of squares of digits using the `getNextSum` function. It is used to detect cycles more quickly.
7. If the loop exits because `fast` becomes 1, it means that the number `n` is a happy number, and the function returns `true`.
8. If the loop exits because `fast` becomes equal to `slow`, it means that there is a cycle in the sequence of sums, and the number `n` is not a happy number. The function returns `false`.

The techniques used in this code include:

- Mathematical operations to calculate the sum of squares of digits.
- Looping to iterate through the sequence of sums of squares of digits.
- Two pointers (slow and fast) to detect cycles in the sequence. This is also known as the Floyd's cycle-finding algorithm, or the "tortoise and hare" algorithm. It's a well-known technique for detecting cycles in linked lists or sequences. In this case, it helps to efficiently determine whether the number is a happy number or not.

## Identical twins

### Identical twins

```
class Solution {
    int getIdenticalTwinsCount (int[] arr) {
        HashMap<Integer, Integer> map = new HashMap<Integer, Integer>();
        for (int i = 0; i < arr.length; i++) {
            Integer val = map.get(arr[i]);
            if(val == null) {
                map.put(arr[i],1);
            } else {
                map.put(arr[i], ++val);
            }
        }

        int count = 0;
        for(Map.Entry<Integer, Integer> y: map.entrySet()){
            int x = (y.getValue() * (y.getValue() - 1)) / 2;
            count += x;

        }
        return count;
    }
}
```

The code is a Java implementation of a function to count the number of identical twins in an array of integers. Identical twins are pairs of elements in the array that have the same value.

Here's a step-by-step explanation of the code:

1. The `getIdenticalTwinsCount` function takes an integer array `arr` as input and returns the count of identical twins in the array.
2. A `HashMap` named `map` is created to store the frequency of each element in the array. The key represents the element value, and the value represents its frequency.
3. The function iterates through the array using a for loop. For each element `arr[i]`, it checks if the element is already present in the `map`. If the element is not present (i.e., `val` is null), it adds the element to the `map` with a frequency of 1. If the element is already present, it increments the frequency by 1.
4. After counting the frequencies of all elements in the array, the function proceeds to calculate the count of identical twins.
5. It initializes a variable `count` to 0, which will store the final count of identical twins.
6. The function then iterates through the entries of the `map` using a for-each loop. For each entry `y` in the `map`, it calculates the count of identical twins for the specific element value.
7. The count of identical twins for a specific element value is calculated as `x = (y.getValue() * (y.getValue() - 1)) / 2`. This formula computes the number of combinations of 2 elements with the same value, which represents the count of identical twins for that value.
8. The calculated `x` is added to the `count`, accumulating the total count of identical twins for all element values.
9. Finally, the function returns the total count of identical twins.

Techniques used in this code include:

- Using a `HashMap` to count the frequencies of elements in the array.
- Iterating through the array and updating the frequency counts in the `HashMap`.
- Iterating through the `HashMap` entries to calculate the count of identical twins for each element value.
- Mathematical computation to calculate the count of identical twins using combinations.

## Implement Queue using Stacks

### Implement Queue using Stacks

```
class MyQueue {  
    stack1: number[] = [];  
    stack2: number[] = [];  
  
    constructor() {}  
  
    push(x: number): void {  
        this.stack1.push(x);  
    }  
  
    pop(): number {
```

```

        this.move();
        return this.stack2.pop();
    }

    peek(): number {
        this.move();
        return this.stack2[this.stack2.length - 1];
    }

    empty(): boolean {
        return !this.stack1.length && !this.stack2.length;
    }

    move(): void {
        if (!this.stack2.length) {
            while (this.stack1.length) {
                this.stack2.push(this.stack1.pop());
            }
        }
    }
}

/**
 * Your MyQueue object will be instantiated and called as such:
 * var obj = new MyQueue()
 * obj.push(x)
 * var param_2 = obj.pop()
 * var param_3 = obj.peek()
 * var param_4 = obj.empty()
 */

```

The code is a TypeScript implementation of a queue data structure using two stacks. It defines a class **MyQueue** with various queue operations implemented using two stacks, **stack1** and **stack2**.

Here's a step-by-step explanation of the code:

1. The class **MyQueue** is defined, which represents a queue data structure.
2. The constructor function is defined, but it doesn't do anything since the stacks (**stack1** and **stack2**) are already initialized as empty arrays.
3. The **push** method allows adding an element to the queue. It simply pushes the element onto **stack1**, which acts as the back of the queue.
4. The **pop** method removes and returns the front element from the queue. It first calls the **move** method to transfer elements from **stack1** to **stack2**, ensuring that the order of elements is reversed (first-in, first-out). Then, it pops the last element from **stack2**, which is the front of the queue.
5. The **peek** method returns the front element of the queue without removing it. Similar to **pop**, it also calls the **move** method to transfer elements from **stack1** to **stack2**, and then returns the last element of **stack2**.
6. The **empty** method checks if the queue is empty. It returns **true** if both **stack1** and **stack2** are empty, indicating that the queue has no elements.

- The `move` method is a private helper function used to transfer elements from `stack1` to `stack2`. It is called by `pop` and `peek` methods when needed. The purpose of this method is to maintain the correct order of elements in the queue, allowing efficient front element retrieval.

Techniques used in this code include:

- Implementing a queue using two stacks to efficiently perform queue operations (enqueue, dequeue, and peek).
- Utilizing the LIFO (Last-In-First-Out) property of stacks to reverse the order of elements when moving them from `stack1` to `stack2`.
- Ensuring that elements are moved to `stack2` only when necessary (on calls to `pop` and `peek`) to avoid unnecessary element transfers.

### Implement Stack using Queues

## Implement Stack using Queues

```
class MyStack {
    public arr: any = [];
    constructor() {

    }

    push(x: number): void {
        const newq = [];
        newq.push(x);
        while (this.arr.length > 0) {
            newq.push(this.arr.shift());
        }
        this.arr = newq;
    }

    pop(): number {
        return this.arr.shift();
    }

    top(): number {
        return this.arr[0];
    }

    empty(): boolean {
        return this.arr.length === 0;
    }
}

/**
 * Your MyStack object will be instantiated and called as such:
 * var obj = new MyStack()
 * obj.push(x)
 * var param_2 = obj.pop()
 * var param_3 = obj.top()
 */
```

```
* var param_4 = obj.empty()
*/
```

The code is a TypeScript implementation of a stack data structure using a single array. It defines a class `MyStack` with various stack operations implemented using the array `arr`.

Here's a step-by-step explanation of the code:

1. The class `MyStack` is defined, which represents a stack data structure.
2. The constructor function is defined, but it doesn't do anything since the stack array (`arr`) is already initialized as an empty array.
3. The `push` method allows adding an element to the stack. It creates a new array `newq`, pushes the given element `x` to it, and then iteratively pops elements from the existing `arr` and pushes them to `newq`. This process effectively reverses the order of elements, making the newly pushed element `x` the top of the stack. Finally, it updates `arr` to be the new reversed array `newq`.
4. The `pop` method removes and returns the top element from the stack. It does this by using the `shift` method, which removes the first element (top) of the array `arr`, effectively simulating the behavior of popping the top element from the stack.
5. The `top` method returns the top element of the stack without removing it. It simply accesses the first element (top) of the array `arr`.
6. The `empty` method checks if the stack is empty. It returns `true` if the array `arr` is empty, indicating that the stack has no elements.

Techniques used in this code include:

- Implementing a stack using a single array.
- Reversing the order of elements in the array using another array to achieve stack behavior (LIFO - Last-In-First-Out).
- Utilizing array methods `push`, `shift`, and array indexing to perform stack operations efficiently.

## Integer to Roman

# Integer to Roman

```
function intToRoman(num: number): string {
    const M = ['', 'M', 'MM', 'MMM'];
    const C = ['', 'C', 'CC', 'CCC', 'CD', 'D', 'DC', 'DCC', 'DCC', 'CM'];
    const X = ['', 'X', 'XX', 'XXX', 'XL', 'L', 'LX', 'LXX', 'LXXX', 'XC'];
    const I = ['', 'I', 'II', 'III', 'IV', 'V', 'VI', 'VII', 'VIII', 'IX'];

    return (
        M[Math.floor(num / 1000)] +
        C[Math.floor((num % 1000) / 100)] +
        X[Math.floor((num % 100) / 10)] +
        I[num % 10]
    );
};
```

The code is a TypeScript function that converts an integer to its Roman numeral representation. It uses predefined arrays for thousands (M), hundreds (C), tens (X), and ones (I) places to build the Roman numeral string.

Here's a step-by-step explanation of the code:

1. The function `intToRoman` takes an integer `num` as input and returns a string representing its Roman numeral equivalent.
2. Four arrays `M`, `C`, `X`, and `I` are defined to represent Roman numerals for thousands, hundreds, tens, and ones places, respectively. Each array contains the Roman numerals for the numbers from 0 to 9 in their corresponding places.
3. The function returns the Roman numeral by concatenating the Roman numerals for each place together.
4. To convert the integer to its Roman numeral representation:
  - `Math.floor(num / 1000)` calculates the thousands place value and retrieves the corresponding Roman numeral from the `M` array.
  - `Math.floor((num % 1000) / 100)` calculates the hundreds place value and retrieves the corresponding Roman numeral from the `C` array.
  - `Math.floor((num % 100) / 10)` calculates the tens place value and retrieves the corresponding Roman numeral from the `X` array.
  - `num % 10` calculates the ones place value and retrieves the corresponding Roman numeral from the `I` array.
5. The Roman numerals for each place are concatenated to form the final Roman numeral representation of the given integer.

Techniques used in this code include:

- Utilizing predefined arrays to represent Roman numerals for different places (thousands, hundreds, tens, and ones).
- Utilizing integer division and remainder operations to extract the place values (thousands, hundreds, tens, and ones) from the given number.
- Concatenating the Roman numerals for each place to construct the final Roman numeral representation of the integer.

## Intersection of two arrays 2

### Intersection of two arrays 2

```
function intersect(nums1: number[], nums2: number[]): number[] {
    let output: number[] = [];
    nums1.sort((a,b) => a-b);
    nums2.sort((a,b) => a-b);

    let i = 0;
    let j = 0;
    while(i < nums1.length && j < nums2.length) {
        if(nums1[i] === nums2[j]) {
            output.push(nums1[i]);
        }
        if(nums1[i] < nums2[j]) {
            i++;
        } else {
            j++;
        }
    }
    return output;
}
```

```

        i++;
        j++;
    } else if(nums1[i] < nums2[j]) {
        i++;
    } else {
        j++;
    }
}
return output;
};

```

The code is a TypeScript function that finds the intersection of two arrays `nums1` and `nums2`. The intersection contains elements that are common to both arrays. The arrays are assumed to be sorted in ascending order.

Here's a step-by-step explanation of the code:

1. The function `intersect` takes two arrays `nums1` and `nums2` as input and returns an array representing their intersection.
2. An empty array `output` is initialized to store the intersected elements.
3. Both `nums1` and `nums2` arrays are sorted in ascending order using the `sort` method.
4. Two pointers `i` and `j` are initialized to traverse through the sorted `nums1` and `nums2` arrays, respectively.
5. A while loop is used to iterate through both arrays while `i` is less than the length of `nums1` and `j` is less than the length of `nums2`.
6. Inside the loop:
  - If `nums1[i]` is equal to `nums2[j]`, it means the elements match and they are added to the `output` array. The pointers `i` and `j` are incremented.
  - If `nums1[i]` is less than `nums2[j]`, it means the current element in `nums1` is smaller, so we move to the next element by incrementing `i`.
  - If `nums1[i]` is greater than `nums2[j]`, it means the current element in `nums2` is smaller, so we move to the next element by incrementing `j`.
7. After the loop, the `output` array contains the intersected elements.
8. The function returns the `output` array, which represents the intersection of the two input arrays.

Techniques used in this code include:

- Sorting arrays to enable efficient comparison and intersection.
- Using two pointers (`i` and `j`) to traverse through the arrays while comparing their elements.
- Determining whether to increment pointers based on the comparison of elements.
- Building the intersected array (`output`) as the pointers traverse the input arrays.

## Intersection of Two Linked Lists

### Intersection of Two Linked Lists

```

/**
 * Definition for singly-linked list.
 * class ListNode {
 *   val: number
 *   next: ListNode | null
 *   constructor(val?: number, next?: ListNode | null) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.next = (next===undefined ? null : next)
 *   }
 * }
 */

function getIntersectionNode(headA: ListNode | null, headB: ListNode | null): ListNode | null {
  let pointer1 = headA;
  let pointer2 = headB;

  while(pointer1 !== pointer2) {
    pointer1 = pointer1 === null ? headB: pointer1.next;
    pointer2 = pointer2 === null ? headA: pointer2.next;

  }
  return pointer1;
};

```

The code is a TypeScript function that finds the intersection node of two singly-linked lists `headA` and `headB`. The intersection node is the node at which the two linked lists meet if they share a common portion.

Here's a step-by-step explanation of the code:

1. The function `getIntersectionNode` takes two parameters: `headA` and `headB`, which are the heads of the two linked lists.
2. Two pointers `pointer1` and `pointer2` are initialized to point to the heads of `headA` and `headB`, respectively.
3. A while loop is used to traverse the linked lists. The loop continues as long as `pointer1` is not equal to `pointer2`, which means they haven't met at the intersection node.
4. Inside the loop:
  - The `pointer1` is moved to the next node (`pointer1.next`), or if it has reached the end of list A (`pointer1` is `null`), it is moved to the head of list B (`headB`).
  - Similarly, the `pointer2` is moved to the next node (`pointer2.next`), or if it has reached the end of list B (`pointer2` is `null`), it is moved to the head of list A (`headA`).
5. The loop continues until `pointer1` and `pointer2` point to the same node, which is the intersection node.
6. Once the intersection node is found, the function returns `pointer1` (or `pointer2`, as they are both pointing to the intersection node).
7. If there is no intersection between the linked lists, both `pointer1` and `pointer2` will eventually become `null`, and the loop will exit.

Techniques used in this code include:

- Using two pointers (`pointer1` and `pointer2`) to traverse the linked lists simultaneously.
- Utilizing the property of linked lists to move one pointer to the head of the other list when it reaches the end, effectively accounting for the difference in lengths between the lists.
- Detecting the intersection point by checking if `pointer1` and `pointer2` are equal.
- Returning the intersection node once it is found.

Overall, this algorithm finds the intersection node of two linked lists with a time complexity of  $O(m + n)$ , where  $m$  and  $n$  are the lengths of the two linked lists.

### Intersection of two arrays

## Intersection of two arrays

```
function intersection(nums1: number[], nums2: number[]) {
  let nums1Map = new Set(nums1);
  let result = new Set();
  nums2.forEach(item => {
    if (nums1Map.has(item)){
      result.add(item)
    }
  });
  return Array.from(result);
};
```

The code is a TypeScript function that finds the intersection of two arrays `nums1` and `nums2`. The intersection of two arrays is a set of elements that are common to both arrays.

Here's a step-by-step explanation of the code:

1. The function `intersection` takes two arrays as parameters: `nums1` and `nums2`.
2. A `Set` named `nums1Map` is created to store unique elements from the `nums1` array.
3. Another `Set` named `result` is created to store the intersection of the two arrays.
4. The `forEach` method is used to iterate through each element in the `nums2` array:
  - For each element `item` in `nums2`, it is checked whether `item` exists in the `nums1Map` set using the `has` method.
  - If `item` is found in `nums1Map`, it means it's an intersection element, so it is added to the `result` set using the `add` method.
5. Once all elements in `nums2` have been processed, the `result` set contains the intersection elements.
6. The `Array.from` method is used to convert the `result` set back into an array, which is then returned as the final result.

Techniques used in this code include:

- Using `Set` to efficiently store and check for unique elements in the arrays.
- Iterating through an array using the `forEach` method.
- Determining the intersection of two arrays by comparing elements using `has` method of a `Set`.
- Converting a `Set` back into an array using the `Array.from` method.

Overall, this algorithm finds the intersection of two arrays with a time complexity of  $O(m + n)$ , where  $m$  and  $n$  are the lengths of the two arrays.

## Inventory update

### Inventory update

```
function updateInventory(arr1: any, arr2: any) {
  const inventory = [...arr1]

  for (let i = 0; i < arr2.length; i++) {
    const item = arr2[i][1];
    const quantity = arr2[i][0];

    const position = inventory.indexOf(item);

    if (position !== -1) {
      const row = Math.floor(position / 2);
      arr1[row][0] += quantity;
      continue;
    }

    arr1.push([quantity, item]);
  }

  arr1.sort((previous: any, next: any) => (previous[1] > [next[1]] ? 1 : -1));
}

return arr1;
}

var curInv = [
  [21, "Bowling Ball"],
  [2, "Dirty Sock"],
  [1, "Hair Pin"],
  [5, "Microphone"],
];

var newInv = [
  [2, "Hair Pin"],
  [3, "Half-Eaten Apple"],
  [67, "Bowling Ball"],
  [7, "Toothpaste"],
];
const result = updateInventory(curInv, newInv);
console.log(result)
```

The code is a TypeScript function that updates the inventory based on two arrays: `curInv` and `newInv`. The inventory is represented as an array of arrays, where each inner array contains the quantity and name of an item.

Here's a step-by-step explanation of the code:

1. The function `updateInventory` takes two parameters: `arr1` (current inventory) and `arr2` (new inventory to be added or updated).
2. A copy of the current inventory, named `inventory`, is created using the spread operator `...arr1`.

3. The code then iterates through each item in the new inventory array `arr2` using a `for` loop:
  - For each item, the quantity (`quantity`) and name (`item`) are extracted from the inner array.
  - The `indexOf` method is used to find the position of the `item` in the `inventory` array.
4. If the `item` is found in the `inventory` array (`position !== -1`), the existing quantity for that item is updated by adding the `quantity` to the existing quantity.
  - The row index of the found item is calculated by dividing the `position` by 2 and taking the floor value.
  - The `arr1[row][0]` (`quantity`) is updated with the new quantity.
5. If the `item` is not found in the `inventory` array, a new inner array [`quantity`, `item`] is added to the `inventory` array.
6. After processing all items in `arr2`, the `inventory` array is sorted based on the item name using the `sort` method and a comparison function.
  - The comparison function compares the names of two items and returns 1 if the first item's name is greater than the second item's name, otherwise -1.
7. The updated `inventory` is returned as the final result.
8. The provided `curInv` and `newInv` arrays are used as test cases, and the `updateInventory` function is called with these arrays.
9. The resulting updated inventory is logged to the console.

Techniques used in this code include:

- Creating a copy of an array using the spread operator.
- Iterating through arrays using `for` loops.
- Finding the index of an element in an array using `indexOf`.
- Updating elements in a multi-dimensional array.
- Sorting an array using the `sort` method with a custom comparison function.

Overall, this algorithm efficiently updates and merges two inventories while ensuring that duplicate items are consolidated and the final inventory is sorted by item name.

## Invert Binary Tree

### Invert Binary Tree

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *   val: number
 *   left: TreeNode | null
 *   right: TreeNode | null
 *   constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
```

```

        }
    }

function invertTree(root: TreeNode | null): TreeNode | null {
    if (root == null) return null;

    const left = root.left;
    const right = root.right;
    root.left = invertTree(right);
    root.right = invertTree(left);
    return root;
}

```

The code is a TypeScript function that inverts a binary tree. It takes the root of a binary tree as input and returns the root of the inverted binary tree.

Here's a step-by-step explanation of the code:

1. The function `invertTree` takes a single parameter `root`, which represents the root of the binary tree to be inverted.
2. The base case of the recursion is handled at the beginning: if `root` is `null`, indicating an empty subtree, the function immediately returns `null`.
3. If `root` is not `null`, the code proceeds to swap the left and right subtrees of the current node, effectively inverting its children.
4. The values of the `left` and `right` children of the current node are stored in the variables `left` and `right`, respectively.
5. The `left` child of the current node is set to the result of recursively calling `invertTree` on the original `right` child. This step inverts the right subtree and assigns it to the left child position.
6. Similarly, the `right` child of the current node is set to the result of recursively calling `invertTree` on the original `left` child. This step inverts the left subtree and assigns it to the right child position.
7. After swapping the subtrees, the `root` of the inverted subtree is returned.

Overall, this algorithm efficiently performs a recursive inversion of a binary tree by swapping the left and right subtrees at each level of the recursion.

Techniques used in this code include:

- Recursive traversal of a binary tree.
- Swapping values or nodes of the tree.
- Handling base cases to terminate recursion.

## Island Perimeter

# Island Perimeter

```

function islandPerimeter(grid: number[][]): number {
    let perimeter = 0;

    for (let i = 0; i < grid.length; i++) {
        for (let j = 0; j < grid[i].length; j++) {
            if (grid[i][j] === 1) {
                perimeter += 4;

                if (j > 0 && grid[i][j - 1] === 1) {
                    perimeter -= 2;
                }

                if (i > 0 && grid[i - 1][j] === 1) {
                    perimeter -= 2;
                }
            }
        }
    }

    return perimeter;
};

```

The code is a TypeScript function that calculates the perimeter of an island represented by a grid. The grid is a two-dimensional array where each cell contains either 0 or 1. The function calculates the perimeter of the island by counting the number of sides that form the boundary of the island.

Here's a step-by-step explanation of the code:

1. The function `islandPerimeter` takes a single parameter `grid`, which is a 2D array representing the island.
2. It initializes the `perimeter` variable to 0. This variable will hold the final perimeter count.
3. The code then uses nested loops to iterate through each cell in the grid. The outer loop iterates through the rows, and the inner loop iterates through the columns.
4. For each cell  $(i, j)$  in the grid, if it is land (i.e., `grid[i][j]` is equal to 1), the perimeter is increased by 4. This is because each land cell contributes 4 sides to the perimeter.
5. The code then checks adjacent cells to see if they are also land cells. If the cell to the left  $(i, j - 1)$  is also land, then the perimeter is decreased by 2. Similarly, if the cell above  $(i - 1, j)$  is also land, the perimeter is decreased by 2. This adjustment is made because each common side between two adjacent land cells contributes 2 sides to the total perimeter.
6. The iteration continues until all cells in the grid have been visited.
7. Finally, the function returns the calculated `perimeter`, which represents the total perimeter of the island.

Techniques used in this code include:

- Nested loops to iterate through a 2D array.
- Conditional statements to check cell values and adjust perimeter accordingly.

## Isomorphic Strings

```
function isIsomorphic(s: string, t: string): boolean {
    if(s.length != t.length) return false;
    const m1 = [];
    const m2 = [];
    for(let i=0; i< s.length; i++){
        if(m1[s[i]] != m2[t[i]]) return false;

        m1[s[i]] = i+1;
        m2[t[i]] = i+1;
    }

    return true;
};
```

The code is a TypeScript function that determines whether two strings, `s` and `t`, are isomorphic. Two strings are considered isomorphic if there is a one-to-one mapping of characters between them such that each character in string `s` can be replaced by a corresponding character in string `t`.

Here's a step-by-step explanation of the code:

1. The function `isIsomorphic` takes two string parameters: `s` and `t`.
2. It first checks if the lengths of the two strings are not equal. If they have different lengths, the function immediately returns `false`, as two strings with different lengths cannot be isomorphic.
3. It initializes two arrays, `m1` and `m2`, to keep track of the mappings of characters between the two strings. The indices of the arrays represent characters in the strings, and the values represent the corresponding indices where these characters were last seen.
4. The function then iterates through each character in the strings using a `for` loop.
5. Inside the loop, it checks if the mappings of the current characters in `s` and `t` are not the same. If they are not the same, it means that the characters do not follow the one-to-one mapping, and the function returns `false`.
6. If the mappings of the characters are the same, the function updates the mappings in arrays `m1` and `m2`. It assigns the current index `i + 1` to the character in both arrays. Adding 1 to the index ensures that a mapping value of 0 (which is the default) is not confused with a valid index.
7. After iterating through all characters, the function returns `true`, indicating that the two strings are indeed isomorphic.

Techniques used in this code include:

- Iterating through strings using a `for` loop.
- Using arrays to store character mappings between the two strings.
- Conditional statements to check character mappings and determine isomorphism.

## Keyboard Row

### Keyboard Row

```
function findWords(words: string[]): string[] {
    const rowMap: Map<string, number> = new Map([
        ['Q', 1], ['W', 1], ['E', 1], ['R', 1], ['T', 1], ['Y', 1], ['U', 1], ['I', 1], ['O', 1], ['P', 1], ['A', 1]
    ]);

    const result: string[] = [];

    for (const word of words) {
        if (isInSameRow(word.toUpperCase(), rowMap)) {
            result.push(word);
        }
    }

    return result;
}

function isInSameRow(word: string, rowMap: Map<string, number>): boolean {
    if (word.length <= 1) {
        return true;
    }

    const rowNum: number = rowMap.get(word[0]) || 0;

    for (let i = 1; i < word.length; i++) {
        const currentChar: string = word[i];
        const currentRowNum: number = rowMap.get(currentChar) || 0;

        if (currentRowNum !== rowNum) {
            return false;
        }
    }

    return true;
}
```

The code is a TypeScript function that takes an array of words and returns an array of words that can be typed using only one row of a keyboard. The function checks if the characters of each word are all located in the same row of the keyboard.

Here's a step-by-step explanation of the code:

1. The `findWords` function takes an array of `words` as a parameter.
2. It initializes a `rowMap` using a `Map` data structure. Each key in the map represents a character, and the corresponding value represents the row number on the keyboard where the character is located.
3. It initializes an empty array called `result` to store the words that can be typed using a single row of the keyboard.
4. The function uses a `for...of` loop to iterate through each `word` in the input `words` array.

5. Inside the loop, it calls the `isInSameRow` function with the uppercase version of the current `word` and the `rowMap`.
6. If the `isInSameRow` function returns `true` for the current `word`, it means that all characters in the word can be typed using the same row of the keyboard. Therefore, the `word` is added to the `result` array.
7. After iterating through all words, the function returns the `result` array.
8. The `isInSameRow` function takes a `word` and the `rowMap` as parameters.
9. If the length of the `word` is less than or equal to 1, the function immediately returns `true` because a word with one or zero characters can be typed using a single row.
10. The function retrieves the row number of the first character of the `word` using the `rowMap`.
11. It then iterates through the remaining characters of the `word` using a `for` loop.
12. For each character, it retrieves its row number from the `rowMap`.
13. If the row number of the current character does not match the row number of the first character, the function returns `false`, indicating that the characters are not in the same row.
14. If the loop completes without returning `false`, it means all characters are in the same row, and the function returns `true`.

Techniques used in this code include:

- Using a `Map` data structure to store keyboard rows and characters.
- Iterating through arrays using `for...of` and `for` loops.
- Converting strings to uppercase using the `toUpperCase` method.
- Checking conditions and returning boolean values.

## Largest palindrome product

### Largest palindrome product

```
function largestPalindromeProduct(n: number): number {
  const upperLimit: number = Math.pow(10, n) - 1;
  const lowerLimit: number = Math.pow(10, n - 1);

  let result: number = 0;

  for (let i: number = upperLimit; i >= lowerLimit; i--) {
    for (let j: number = i; j >= lowerLimit; j--) {
      let product: number = i * j;
      let reversed: string = [...String(product)].reverse().join("");
      if (product === Number(reversed)) {
        result = Math.max(product, result);
        break;
      }
    }
  }

  return result;
}
```

### **Step-by-step breakdown:**

#### **1. Calculate Upper and Lower Limits:**

- `upperLimit` is calculated as  $10^n - 1$ . It represents the highest n-digit number.
- `lowerLimit` is calculated as  $10^{(n-1)}$ . It represents the lowest n-digit number.

#### **2. Initialization:**

- Initialize a variable `result` to 0. This variable will store the largest palindrome product found.

#### **3. Nested Loops:**

- The outer loop runs from `upperLimit` to `lowerLimit` (inclusive). It represents the first factor of the product.
- The inner loop runs from the current value of the outer loop variable `i` to `lowerLimit` (inclusive). It represents the second factor of the product.

#### **4. Calculate Product and Check for Palindrome:**

- Calculate the product of the current values of `i` and `j`.
- Convert the product to a string, reverse the string, and join it back. This gives the reversed version of the product.
- Check if the product is equal to its reverse. If yes, it's a palindrome.

#### **5. Update the Result:**

- If the current palindrome product is greater than the current `result`, update `result` with this palindrome product.
- Break out of the inner loop to move to the next value of `i`.

#### **6. Return the Result:**

- After both loops complete execution, return the `result`, which holds the largest palindrome product of two n-digit numbers.

### **Largest prime factor**

## **Largest prime factor**

```
function largestPrimeFactor(number: number): number {
    let largestFactor: number = 1;

    while (number % 2 === 0) {
        largestFactor = 2;
        number = number / 2;
    }

    for (let i: number = 3; i <= Math.sqrt(number); i += 2) {
        while (number % i === 0) {
            largestFactor = i;
            number = number / i;
        }
    }
}
```

```

    }

    if (number > 2) {
        largestFactor = number;
    }

    return largestFactor;
}

```

#### Step-by-step breakdown:

**Step 1:** The `while` loop checks if the given number is divisible by 2. If it is, it divides the number by 2 and updates the `largestFactor` variable to 2. This step continues until the number becomes odd.

**Step 2:** The `for` loop starts from 3 and iterates up to the square root of the remaining number (`Math.sqrt(number)`). It increments `i` by 2 in each iteration to check only odd numbers (skipping even numbers other than 2 for efficiency).

**Step 3:** Within the `for` loop, the `while` loop checks if the current factor (`i`) divides the remaining number. If it does, it updates the `largestFactor` variable and divides the number by `i`.

**Step 4:** If the current factor divides the number, the `largestFactor` is updated to the current factor, and the number is divided by the current factor.

**Step 5:** After the `for` loop, there might be a remaining number greater than 2, which could be a prime factor. The `if` statement checks if the remaining number is greater than 2 and updates the `largestFactor` variable if necessary.

**Step 6:** Finally, the function returns the `largestFactor` variable, which holds the largest prime factor of the given number.

#### Largest product in a grid

### Largest product in a grid

```

function largestGridProduct(arr: number[][]): number {
    let maxProduct: number = 0;
    let currProduct: number = 0;

    function maxProductChecker(n: number): void {
        if (n > maxProduct) {
            maxProduct = n;
        }
    }

    // loop rows
    for (let r = 0; r < arr.length; r++) {
        // loop columns
        for (let c = 0; c < arr[r].length; c++) {
            const limit = arr[r].length - 3;

            // check horizontal
            if (c < limit) {
                currProduct = arr[r][c] * arr[r][c + 1] * arr[r][c + 2] * arr[r][c + 3];
                maxProductChecker(currProduct);
            }
        }
    }
}

```

```

        maxProductChecker(currProduct);
    }

    // check vertical
    if (r < limit) {
        currProduct = arr[r][c] * arr[r + 1][c] * arr[r + 2][c] * arr[r + 3][c];
        maxProductChecker(currProduct);
    }

    // check diagonal []
    if (c < limit && r < limit) {
        currProduct =
            arr[r][c] * arr[r + 1][c + 1] * arr[r + 2][c + 2] * arr[r + 3][c + 3];
        maxProductChecker(currProduct);
    }

    // check diagonal []
    if (c > 3 && r < limit) {
        currProduct =
            arr[r][c] * arr[r + 1][c - 1] * arr[r + 2][c - 2] * arr[r + 3][c - 3];
        maxProductChecker(currProduct);
    }
}

return maxProduct;
}

```

#### Step-by-step breakdown:

##### 1. Initialize Variables:

- Initialize `maxProduct` and `currProduct` to 0. These variables will be used to keep track of the maximum product and the current product being calculated.

##### 2. `maxProductChecker` Function:

- Define a helper function `maxProductChecker` that takes a number `n` as an argument.
- If `n` is greater than the current `maxProduct`, update `maxProduct` with the value of `n`.

##### 3. Nested Loop for Grid Exploration:

- Use nested loops to iterate through each element in the 2D array (`arr`).
- The outer loop (`r`) iterates through rows, and the inner loop (`c`) iterates through columns.

##### 4. Check Horizontal Products:

- Inside the loops, check for horizontal products if there are at least 4 elements to the right (`c < limit`).
- Calculate the product of four consecutive elements in the same row and call `maxProductChecker` with the current product.

##### 5. Check Vertical Products:

- Check for vertical products if there are at least 4 elements below (`r < limit`).

- Calculate the product of four consecutive elements in the same column and call `maxProductChecker` with the current product.

#### 6. Check Diagonal Products (\ and /):

- Check for diagonal products (\) if there are at least 4 elements to the right and below (`c < limit` and `r < limit`).
- Check for diagonal products (/) if there are at least 4 elements to the left and below (`c > 3` and `r < limit`).
- Calculate the product of four consecutive elements along each diagonal and call `maxProductChecker` with the current product.

#### 7. Return the Maximum Product:

- Return the final `maxProduct` after exploring all possible products in the grid.

The function is designed to find the largest product of four consecutive elements (horizontally, vertically, or diagonally) in a 2D array.

### Largest product in a series

#### Largest product in a series

```
function largestProductInASeries(n) {

    let thousandDigits = [7,3,1,6,7,1,7,6,5,3,1,3,3,0,6,2,4,9,1,9,2,2,5,1,1,9,6,7,4,4,2,6,5,7,4,7,4,2,3,5

        let maxProduct = 0;
        for (let i = 0; i <= thousandDigits.length - n; i++) {
            let product = 1;
            for (let j = 0; j < n; j++) {
                product *= parseInt(thousandDigits[i + j], 10);
            }
            maxProduct = Math.max(maxProduct, product);
        }
        return maxProduct;
    }

    largestProductInASeries(13);
```

#### 1. Initialization:

- Initialize the `thousandDigits` array with 1000 digits represented as individual numbers.
- Initialize a variable `maxProduct` to store the maximum product found. Set its initial value to 0.

#### 2. Loop through the `thousandDigits` array:

- Use a `for` loop to iterate through the `thousandDigits` array up to the length minus `n` (since you are looking for `n` consecutive digits).
- Within the loop:

- Initialize a variable `product` to 1 to store the product of `n` consecutive digits.
- Use another `for` loop to iterate through the next `n` digits.
- Multiply the current digit with the `product`.
- Update `maxProduct` to be the maximum of the current `product` and the existing `maxProduct`.

### 3. Return the Result:

- Once the loop finishes, return the `maxProduct`, which represents the largest product of `n` consecutive digits in the `thousandDigits` array.

In this specific case, the function is finding the largest product of 13 consecutive digits in the given 1000-digit number sequence.

## Length of Last Word

### Length of Last Word

```
function lengthOfLastWord(s: string): number {
  s = s.trim();
  let counter = 0;
  for(let i = s.length - 1; i >= 0; i--) {
    if(s[i] === ' ') {
      break;
    }
    counter++;
  }

  return counter;
};
```

The code is a TypeScript function that calculates the length of the last word in a given string. It does this by iterating through the string from the end and counting the characters of the last word until a space character is encountered.

Here's a step-by-step explanation of the code:

1. The `lengthOfLastWord` function takes a single parameter `s`, which is the input string.
2. It first trims the input string using the `trim` method to remove any leading or trailing whitespace.
3. It initializes a variable `counter` to keep track of the length of the last word.
4. The function uses a `for` loop to iterate through the characters of the trimmed string in reverse order, starting from the end of the string (`s.length - 1`) and going backward.
5. Inside the loop, it checks if the current character (`s[i]`) is a space character (' '). If a space character is encountered, it means the last word has been counted, and the loop is terminated using the `break` statement.
6. If the current character is not a space character, the `counter` is incremented.
7. After the loop completes, the function returns the value of the `counter`, which represents the length of the last word.

Techniques used in this code include:

- Using the `trim` method to remove leading and trailing whitespace from a string.
- Iterating through a string in reverse order using a `for` loop.
- Using conditional statements (`if`) to check characters and control the loop.
- Using the `break` statement to exit a loop prematurely.

### Letter Combinations of a Phone Number

## Letter Combinations of a Phone Number

```
function letterCombinations(digits: string): string[] {  
    if (digits.length === 0) {  
        return [];  
    }  
  
    const digitMap: { [key: string]: string[] } = {  
        "2": ["a", "b", "c"],  
        "3": ["d", "e", "f"],  
        "4": ["g", "h", "i"],  
        "5": ["j", "k", "l"],  
        "6": ["m", "n", "o"],  
        "7": ["p", "q", "r", "s"],  
        "8": ["t", "u", "v"],  
        "9": ["w", "x", "y", "z"],  
    };  
  
    const result: string[] = [];  
  
    function backtrack(combination: string, index: number) {  
        if (index === digits.length) {  
            result.push(combination);  
            return;  
        }  
  
        const currentDigit = digits[index];  
        const letters = digitMap[currentDigit];  
        for (const letter of letters) {  
            backtrack(combination + letter, index + 1);  
        }  
    }  
  
    backtrack("", 0);  
    return result;  
}
```

#### 1. Input Validation:

- The function takes a string of digits as input.

#### 2. Base Case Check:

- If the input `digits` string is empty, the function returns an empty array, as there are no combinations to generate.

### **3. Digit to Letter Mapping:**

- Create a mapping of digits to their corresponding letters according to the telephone keypad. For example, '2' corresponds to ['a', 'b', 'c'].

### **4. Initialize Result Array:**

- Initialize an empty array called `result` to store the final combinations.

### **5. Backtracking Function:**

- Implement a recursive backtracking function. This function takes two parameters:
  - `combination`: A string representing the current combination being constructed.
  - `index`: An integer representing the index of the current digit in the input `digits` string.

### **6. Recursive Backtracking:**

- The backtracking function proceeds recursively, exploring all possible combinations.
- If the `index` becomes equal to the length of the input `digits` string, the current `combination` is complete and is added to the `result` array.
- If the `index` is not at the end of the `digits` string, get the letters corresponding to the current digit from the `digitMap`.
- For each letter obtained, append it to the current `combination` and make a recursive call to the backtracking function with the updated `combination` and the next index (`index + 1`).

### **7. Exploration and Recursion:**

- The recursive calls explore all possible combinations by trying different letters for each digit in the input string.
- The backtracking approach ensures that all possible combinations are explored.

### **8. Final Result:**

- After backtracking, the `result` array contains all valid letter combinations.
- Return the `result` array as the output of the function.

## **License Key Formatting**

# **License Key Formatting**

```

function licenseKeyFormatting(s: string, k: number): string {
  const str = s.replace(/-/g, '').toUpperCase();

  let result = '';
  let count = 0;

  for (let i = str.length - 1; i >= 0; i--) {
    result = str[i] + result;
    count++;

    if (count === k && i !== 0) {
      result = '-' + result;
      count = 0;
    }
  }

  return result;
};

```

The code is a TypeScript function that formats a license key string by removing hyphens and grouping the characters in blocks of a specified length. It then returns the formatted license key.

Here's a step-by-step explanation of the code:

1. The `licenseKeyFormatting` function takes two parameters: `s` (the input license key string) and `k` (the desired block length).
2. It first uses the `replace` method with a regular expression `(/-g)` to remove all hyphen characters from the input string `s`. Then, the `toUpperCase` method is applied to convert all characters to uppercase.
3. The function initializes two variables: `result`, which will store the formatted license key, and `count`, which keeps track of the number of characters added to the current block.
4. The function uses a `for` loop to iterate through the characters of the modified string `str` in reverse order, starting from the end of the string (`str.length - 1`) and going backward.
5. Inside the loop:
  - The current character `str[i]` is concatenated to the beginning of the `result` string.
  - The `count` is incremented.
6. If the `count` reaches the specified block length `k` and the loop index `i` is not at the beginning of the string (i.e., `i !== 0`), a hyphen `-` is concatenated to the beginning of the `result` string, and the `count` is reset to 0. This creates the desired block separation.
7. After the loop completes, the function returns the `result`, which now contains the formatted license key.

Techniques used in this code include:

- Using regular expressions (`replace`) to modify a string.
- Iterating through a string in reverse order using a `for` loop.
- Concatenating strings to build the formatted license key.
- Using conditional statements (`if`) to check conditions and control the loop.

## Linked List Cycle

```
/**
 * Definition for singly-linked list.
 * function ListNode(val) {
 *     this.val = val;
 *     this.next = null;
 * }
 */

/**
 * @param {ListNode} head
 * @return {boolean}
 */
var hasCycle = function(head) {

    let slowNode = head;
    let fastNode = head;

    while(fastNode !== null && fastNode.next !== null) {
        fastNode = fastNode.next.next;
        slowNode = slowNode.next;

        if(fastNode === slowNode) {
            return true
        }
    }

    return false;
};
```

The code is a JavaScript function that determines whether a singly-linked list has a cycle (a loop) or not. It uses the "tortoise and hare" approach, also known as Floyd's Cycle Detection Algorithm, to detect cycles in a linked list.

Here's a step-by-step explanation of the code:

1. The `hasCycle` function takes a single parameter `head`, which represents the head of the linked list.
2. It initializes two pointers: `slowNode` and `fastNode`, both pointing to the `head` of the linked list.
3. It enters a `while` loop that continues as long as `fastNode` is not `null` and `fastNode.next` is not `null`. This condition ensures that the fast pointer can make a valid move without encountering a null reference.
4. Inside the loop:
  - The `fastNode` pointer moves two steps ahead by accessing its `.next.next` reference. This simulates the "hare" moving faster in the linked list.
  - The `slowNode` pointer moves one step ahead by accessing its `.next` reference. This simulates the "tortoise" moving slower in the linked list.

5. After each iteration of the loop, the code checks whether the `fastNode` and `slowNode` pointers are pointing to the same node. If they are, it means a cycle has been detected, and the function returns `true`.
6. If the loop completes without finding a cycle, the function returns `false`.

The technique used in this code is the "tortoise and hare" algorithm for cycle detection in linked lists. This algorithm is based on the idea that if there is a cycle in the linked list, the fast and slow pointers will eventually meet at some point within the cycle. If there is no cycle, the fast pointer will reach the end of the list without encountering a null reference.

Overall, this algorithm has a time complexity of  $O(n)$ , where  $n$  is the number of nodes in the linked list, and a space complexity of  $O(1)$ , as it only uses a constant amount of extra space for the two pointers.

## Lonely Integer

# Lonely Integer

```
function lonelyInteger(a: number[]): number {
    return a.reduce((a: number, b: number) => a^b);
}
```

The code is a TypeScript function that finds the "lonely integer" in an array of integers. A "lonely integer" is an element that appears only once in the array, while all other elements appear in pairs.

Here's a step-by-step explanation of the code:

1. The `lonelyInteger` function takes a single parameter `a`, which is an array of integers.
2. It uses the `reduce` method to iterate through the array and apply the XOR (^) operation on each element.
  - The XOR operation returns 0 when two identical numbers are XORed together (even number of times).
  - If a number is XORed with 0, it remains unchanged.
  - XOR is commutative, so the order of the operands doesn't matter.
3. The initial value for the accumulator `a` is set to 0.
4. For each element `b` in the array, the XOR operation is applied between the current accumulator value `a` and the current array element `b`. The result is assigned back to the accumulator `a`.
5. The final result of the `reduce` operation is the XOR of all elements in the array.
6. The function returns the computed XOR value, which corresponds to the lonely integer in the array.

This solution works based on the property of the XOR operation where XORing a number with itself results in 0. Therefore, when XORing all pairs of identical integers, they cancel out, leaving only the lonely integer.

The technique used in this code is the XOR operation to efficiently find the lonely integer in an array. The XOR operation has the property that it is both commutative and associative, making it a suitable choice for this problem. The time complexity of this solution is  $O(n)$ , where  $n$  is the number of elements in the array.

## Longest common prefix

### Longest common prefix

```
function longestCommonPrefix(strs: string[]): string {
    const size = strs.length;
    if(size === 0) return "";
    if(size === 1) return strs[0];

    strs.sort();

    let end = Math.min(strs[0].length, strs[size - 1].length);
    let i = 0;

    while(i < end && strs[0][i] === strs[size - 1][i]) {
        i++
    }

    const output = strs[0].substring(0, i);
    return output;
};
```

#### Step-by-Step Explanation:

1. Define a function `longestCommonPrefix` that takes an array of strings `strs` as input.
2. Get the size of the input array `strs`.
3. If the array is empty (size is 0), return an empty string, as there can be no common prefix.
4. If the array contains only one string (size is 1), return that string as it is the common prefix.
5. Sort the array `strs` lexicographically to facilitate comparison between the first and last strings.
6. Determine the `end` index as the minimum of the lengths of the first and last strings in the sorted array.
7. Initialize an index variable `i` to 0 to traverse characters in the strings.
8. Use a loop to compare characters at index `i` of the first and last strings, continuing as long as the characters match and `i` is less than `end`.
9. Increment `i` until either a character mismatch is found or `i` reaches the `end` index.
10. After the loop exits, extract a substring from the first string from index 0 to index `i` (exclusive). This substring represents the longest common prefix.
11. Return the extracted substring as the result, which is the longest common prefix among the strings in the input array.

#### Techniques Used:

1. **Array Sorting:** The array of strings is sorted lexicographically to bring potentially common prefixes closer together for comparison.

2. **String Comparison:** The characters of the first and last strings are compared character by character to find the longest common prefix.
3. **Substring Extraction:** A substring is extracted from the first string based on the index where the characters of the first and last strings start to differ.

#### **Summary:**

The `longestCommonPrefix` function takes an array of strings, sorts them, and compares characters to find the longest common prefix among the strings. By sorting the array, it reduces the comparison complexity and efficiently identifies the common prefix. The technique of lexicographical sorting and character-by-character comparison ensures an optimal solution for finding the longest common prefix.

### **Longest Harmonious Subsequence**

## **Longest Harmonious Subsequence**

```
function findLHS(nums: number[]): number {
    const frequencyMap: Map<number, number> = new Map();
    for (const num of nums) {
        frequencyMap.set(num, (frequencyMap.get(num) || 0) + 1);
    }

    let maxLHS = 0;
    for (const key of frequencyMap.keys()) {
        if (frequencyMap.has(key + 1)) {
            const currentLHS = frequencyMap.get(key)! + frequencyMap.get(key + 1)!;
            maxLHS = Math.max(maxLHS, currentLHS);
        }
    }

    return maxLHS;
};
```

#### **Step-by-Step solution:**

##### **1. Initialize a Map:**

- `const frequencyMap: Map<number, number> = new Map();`
- Initialize a map called `frequencyMap` to store the frequency of each number.

##### **2. Populate the Frequency Map:**

- Use a `for...of` loop to iterate through each number in the input array (`nums`).
- Update the frequency in the map using `frequencyMap.set()`, ensuring that if the number is not in the map, it starts with a frequency of 1.

##### **3. Initialize Variables:**

- `let maxLHS = 0;`
- Initialize a variable `maxLHS` to store the maximum length of a harmonious subsequence (LHS).

##### **4. Iterate Through Frequency Map Keys:**

- `for (const key of frequencyMap.keys()) {`
- Iterate through the keys of the frequency map.

#### 5. Check for Key + 1:

- `if (frequencyMap.has(key + 1)) {`
- Check if the map contains the key incremented by 1.

#### 6. Calculate Current LHS Length:

- `const currentLHS = frequencyMap.get(key)! + frequencyMap.get(key + 1)!`;
- Calculate the length of the harmonious subsequence by adding the frequencies of the current key and key + 1.

#### 7. Update Maximum LHS Length:

- `maxLHS = Math.max(maxLHS, currentLHS);`
- Update the maximum LHS length if the current length is greater.

#### 8. Return Maximum LHS Length:

- `return maxLHS;`
- Return the final maximum length of a harmonious subsequence.

**Longest palindrome**

## Longest palindrome

```
function longestPalindrome(s: string): string {

    let length = s.length;
    let maxLength = 1;
    let start = 0;

    let table = new Array(length);
    for(let i = 0; i < length; i++) {
        table[i] = new Array(length);
    }

    for(let i = 0; i < length; i++) {
        table[i][i] = true;
    }

    for(let i = 0; i < length; i++) {
        if(s[i] === s[i+1]) {
            table[i][i + 1] = true;
            start = i;
            maxLength = 2;
        }
    }

    for(let k = 3; k <= length; k++) {
        for(let i = 0; i < length - k + 1; i++) {

```

```

        let j = i + k - 1;

        if(table[i + 1][j - 1] && s[i] == s[j]) {
            table[i][j] = true;
            if(k > maxLength) {
                start = i;
                maxLength = k;
            }
        }
    }

    return s.slice(start, start + maxLength);
};

```

#### Step-by-Step solution:

1. Define a function `longestPalindrome` that takes a string `s` as input.
2. Initialize variables:
  - `length`: Length of the input string.
  - `maxLength`: Length of the longest palindrome found.
  - `start`: Starting index of the longest palindrome.
3. Create a 2D array `table` to store information about palindromes. Initialize this array with `false` values.
4. Iterate through the characters of the string using a loop, and set `table[i][i]` to `true` for each character, indicating that single characters are palindromes.
5. Iterate through the string again, this time checking for adjacent identical characters. If found, set `table[i][i + 1]` to `true`, indicating palindromes of length 2, and update `start` and `maxLength` accordingly.
6. Use two nested loops to iterate through possible palindrome lengths `k` starting from 3. The outer loop handles the length of palindromes, and the inner loop iterates through the string, checking substrings of length `k`.
7. For each substring of length `k`, check if the characters at the current positions `i` and `j` match (`s[i] == s[j]`) and if the substring within `table[i + 1][j - 1]` is also a palindrome. If both conditions are met, set `table[i][j]` to `true` and update `start` and `maxLength`.
8. After completing all iterations, return the substring of `s` starting from index `start` and spanning `maxLength` characters. This substring represents the longest palindrome found in the input string.

#### Techniques Used:

1. **Dynamic Programming:** The algorithm utilizes a dynamic programming approach to solve the longest palindrome problem. The `table` array is used to store intermediate results, avoiding redundant calculations and improving efficiency.
2. **String Manipulation:** The algorithm works with the input string to identify palindromic substrings and their lengths.

- 3. Nested Loops:** The solution involves nested loops to iterate through various indices and lengths of substrings, allowing for the comparison and identification of palindromic segments.

#### Summary:

The `longestPalindrome` function employs dynamic programming to find the longest palindromic substring within the input string. By utilizing the `table` array to store and track palindromic substrings, the algorithm optimally identifies the longest palindrome. The technique of dynamic programming and efficient substring comparison ensures an effective solution for finding the longest palindrome in a string.

#### Longest Palindromic Substring

### Longest Palindromic Substring

```
function longestPalindrome(s: string): string {

    let length = s.length;
    let maxLength = 1;
    let start = 0;

    let table = new Array(length);
    for(let i = 0; i < length; i++) {
        table[i] = new Array(length);
    }

    for(let i = 0; i < length; i++) {
        table[i][i] = true;
    }

    for(let i = 0; i < length; i++) {
        if(s[i] === s[i+1]) {
            table[i][i + 1] = true;
            start = i;
            maxLength = 2;
        }
    }

    for(let k = 3; k <= length; k++) {
        for(let i = 0; i < length - k + 1; i++) {
            let j = i + k - 1;

            if(table[i + 1][j - 1] && s[i] == s[j]) {
                table[i][j] = true;
                if(k > maxLength) {
                    start = i;
                    maxLength = k;
                }
            }
        }
    }
}
```

```
    return s.slice(start, start + maxLength);
};
```

### Step-by-Step solution:

1. Define a function `longestPalindrome` that takes a string `s` as input.
2. Initialize variables:
  - `length`: Length of the input string.
  - `maxLength`: Length of the longest palindrome found.
  - `start`: Starting index of the longest palindrome.
3. Create a 2D array `table` to store information about palindromes. Initialize this array with `false` values.
4. Iterate through the characters of the string using a loop, and set `table[i][i]` to `true` for each character, indicating that single characters are palindromes.
5. Iterate through the string again, this time checking for adjacent identical characters. If found, set `table[i][i + 1]` to `true`, indicating palindromes of length 2, and update `start` and `maxLength` accordingly.
6. Use two nested loops to iterate through possible palindrome lengths `k` starting from 3. The outer loop handles the length of palindromes, and the inner loop iterates through the string, checking substrings of length `k`.
7. For each substring of length `k`, check if the characters at the current positions `i` and `j` match (`s[i] == s[j]`) and if the substring within `table[i + 1][j - 1]` is also a palindrome. If both conditions are met, set `table[i][j]` to `true` and update `start` and `maxLength`.
8. After completing all iterations, return the substring of `s` starting from index `start` and spanning `maxLength` characters. This substring represents the longest palindrome found in the input string.

### Techniques Used:

1. **Dynamic Programming:** The algorithm utilizes a dynamic programming approach to solve the longest palindrome problem. The `table` array is used to store intermediate results, avoiding redundant calculations and improving efficiency.
2. **String Manipulation:** The algorithm works with the input string to identify palindromic substrings and their lengths.
3. **Nested Loops:** The solution involves nested loops to iterate through various indices and lengths of substrings, allowing for the comparison and identification of palindromic segments.

### Summary:

The `longestPalindrome` function employs dynamic programming to find the longest palindromic substring within the input string. By utilizing the `table` array to store and track palindromic substrings, the algorithm optimally identifies the longest palindrome. The technique of dynamic programming and efficient substring comparison ensures an effective solution for finding the longest palindrome in a string.

## Longest Substring Without Repeating Characters

# Longest Substring Without Repeating Characters

```
function lengthOfLongestSubstring(s: string): number {
    if(s.length === 0) return 0;

    let start = 0;
    let end = 0;
    let maxLength = 0;
    const stringSet = new Set<string>();

    while(end < s.length) {
        if(!stringSet.has(s[end])) {
            stringSet.add(s[end]);
            end++;
            maxLength = Math.max(maxLength, stringSet.size);
        } else {
            stringSet.delete(s[start]);
            start++;
        }
    }

    return maxLength;
};
```

### Step-by-Step solution:

1. Define a function `lengthOfLongestSubstring` that takes a string `s` as input.
2. Handle the base case: If the input string's length is 0, return 0.
3. Initialize variables:
  - `start`: The starting index of the current substring.
  - `end`: The ending index of the current substring.
  - `maxLength`: The length of the longest substring without repeating characters.
  - `stringSet`: A set to store unique characters within the current substring.
4. Enter a loop that iterates through the characters of the input string while `end` is within the string's length:
  - a. If the character at `end` is not already in `stringSet`:
    - Add the character to `stringSet`.
    - Increment `end`.
    - Update `maxLength` to the maximum of its current value and the size of `stringSet`.
  - b. If the character at `end` is already in `stringSet`:
    - Remove the character at `start` from `stringSet`.
    - Increment `start`.

- After the loop completes, return the value of `maxLength`, which represents the length of the longest substring without repeating characters.

#### Techniques Used:

- Sliding Window:** The algorithm utilizes the sliding window technique to efficiently handle substrings and track the current substring's start and end indices.
- Set Data Structure:** The algorithm uses a Set to keep track of unique characters within the current substring, enabling efficient character uniqueness checking.
- Looping and Conditional Logic:** The solution involves iterating through the input string and using conditional logic to update the sliding window and track the longest substring.

#### Summary:

The `lengthOfLongestSubstring` function employs the sliding window technique along with a set to efficiently find the length of the longest substring without repeating characters in the given input string. By maintaining a sliding window and checking character uniqueness using a set, the algorithm optimally identifies the desired substring length. The use of looping, conditional logic, and a set data structure ensures an effective solution for this problem.

## Longest Uncommon Subsequence I

### Longest Uncommon Subsequence I

```
function findLUSlength(a: string, b: string): number {
    if (a === b) {
        return -1;
    } else {
        return Math.max(a.length, b.length);
    }
};
```

#### Step-by-Step solution:

- Define a function `findLUSlength` that takes two strings `a` and `b` as input.
- Check if strings `a` and `b` are equal:
  - If `a` is equal to `b`, return `-1`. This is because a common subsequence cannot be longer than the strings themselves, so there can be no uncommon subsequence.
- If `a` and `b` are not equal:
  - Return the maximum length between the lengths of strings `a` and `b`. This is because if the two strings are not equal, the longer string itself is the longest uncommon subsequence.
- The function returns the length of the longest uncommon subsequence.

#### Techniques Used:

- Conditional Logic:** The algorithm uses conditional statements to check if the input strings are equal and returns the appropriate result based on the condition.
- Math.max:** The algorithm uses the `Math.max` function to compare the lengths of strings `a` and `b` and return the maximum length.

### Summary:

The `findLUSlength` function determines the length of the longest uncommon subsequence between two input strings `a` and `b`. If the two strings are equal, it returns -1, indicating that there is no uncommon subsequence. If the strings are not equal, it returns the maximum length between the lengths of the two strings, as the longer string itself is the longest uncommon subsequence. The algorithm uses simple conditional logic and the `Math.max` function to achieve this.

## Majority Element

### Majority Element

```
function majorityElement(nums: number[]): number {

    let count = 0;
    let res = 0;

    for(let i = 0; i<=nums.length; i++) {
        if (count==0) {
            res = nums[i];
        }
        if (nums[i] !=res) {
            count--;
        }
        else {
            count++;
        }
    }
    return res;
};
```

### Step-by-Step solution:

- Define a function `majorityElement` that takes an array of integers `nums` as input.
- Initialize two variables:
  - `count`: To keep track of the count of the current majority element.
  - `res`: To store the potential majority element.
- Iterate through each element in the array `nums` using a for loop:
  - If `count` is 0, set `res` to the current element `nums[i]`. This is because we are looking for a majority element, which appears more than half the time. By setting `res` to the current element, we are starting a potential majority count.
  - If the current element `nums[i]` is not equal to `res`, decrement the `count`. This indicates that the current element is different from the potential majority element, reducing its count.

- If the current element `nums[i]` is equal to `res`, increment the `count`. This indicates that the current element matches the potential majority element.
- After iterating through the array, return the final value of `res`, which should represent the majority element.

#### Techniques Used:

- Loop Iteration:** The algorithm uses a `for` loop to iterate through the elements of the input array.
- Conditional Logic:** The algorithm uses conditional statements to update the `res` and `count` variables based on the conditions specified in the problem.

#### Summary:

The `majorityElement` function aims to find the majority element in an array of integers. It iterates through the array and uses `res` and `count` variables to keep track of a potential majority element and its count. The algorithm follows a simple logic to identify the majority element and returns the final result.

## Max Consecutive Ones

### Max Consecutive Ones

```
function findMaxConsecutiveOnes(nums: number[]): number {
  let maxCount = 0;
  let currentCount = 0;

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

  return maxCount;
};
```

#### Step-by-Step solution:

- Define a function `findMaxConsecutiveOnes` that takes an array of integers `nums` as input.
- Initialize two variables:
  - `maxCount`: To keep track of the maximum consecutive count of ones encountered so far.
  - `currentCount`: To keep track of the current consecutive count of ones being processed.
- Iterate through each element in the array `nums` using a `for` loop:
  - If the current element `nums[i]` is equal to 1, increment the `currentCount` by 1 and update `maxCount` to the maximum of `maxCount` and `currentCount`. This ensures that `maxCount` always holds the maximum consecutive count of ones.

- If the current element `nums[i]` is not equal to 1 (i.e., it's 0), reset `currentCount` to 0. This effectively breaks the consecutive count of ones sequence.
4. After iterating through the entire array, return the final value of `maxCount`, which represents the maximum consecutive count of ones.

#### Techniques Used:

1. **Loop Iteration:** The algorithm uses a `for` loop to iterate through the elements of the input array.
2. **Conditional Logic:** The algorithm uses conditional statements to increment and reset the `currentCount` based on the value of the current element.
3. **Math.max:** The `Math.max` function is used to update the `maxCount` with the maximum value between the current `maxCount` and `currentCount`.

#### Summary:

The `findMaxConsecutiveOnes` function aims to find the maximum consecutive count of ones in an array of integers. It iterates through the array, keeping track of the current consecutive count of ones and updating the maximum count encountered so far. The algorithm efficiently determines the maximum consecutive ones count and returns the result.

## Maximum Depth of Binary Tree

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */

function maxDepth(root: TreeNode | null): number {
    if(root === null) return 0;
    else {
        let leftDepth = maxDepth(root.left);
        let rightDepth = maxDepth(root.right);
        if(leftDepth > rightDepth) return leftDepth + 1;
        else return rightDepth + 1;
    }
};

};
```

#### Step-by-Step solution:

1. Define a function `maxDepth` that takes a binary tree node `root` as input.
2. Check if the `root` is `null` (indicating an empty subtree). If so, return 0 as the depth of the subtree is 0.
3. If the `root` is not `null`, recursively calculate the maximum depth of the left and right subtrees using two recursive calls to `maxDepth(root.left)` and `maxDepth(root.right)`.
4. Compare the depths of the left and right subtrees. If the `leftDepth` is greater than the `rightDepth`, return `leftDepth + 1`, indicating the maximum depth of the current subtree. Otherwise, return `rightDepth + 1`.

#### Techniques Used:

1. **Recursion:** The algorithm uses recursion to calculate the maximum depth of the binary tree. It breaks down the problem by considering the left and right subtrees.
2. **Conditional Logic:** The algorithm uses conditional statements to compare the depths of the left and right subtrees and determine the maximum depth of the current subtree.

#### Summary:

The `maxDepth` function calculates the maximum depth of a binary tree using a recursive approach. It handles both the base case (empty subtree) and the recursive case (non-empty subtree) to calculate the maximum depth. The algorithm returns the maximum depth of the given binary tree.

#### Maximum Depth of N-ary Tree

### Maximum Depth of N-ary Tree

```
/**
 * Definition for Node.
 * class Node {
 *   val: number
 *   children: Node[]
 *   constructor(val?: number, children?: Node[]) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.children = (children===undefined ? [] : children)
 *   }
 * }
 */

function maxDepth(root: Node | null): number {
  if (root === null) {
    return 0;
  }

  let maxChildDepth = 0;

  for (let child of root.children) {
    const childDepth = maxDepth(child);
    maxChildDepth = Math.max(maxChildDepth, childDepth);
  }

  return maxChildDepth + 1;
}
```

```

    }

    return maxChildDepth + 1;
};


```

### Step-by-Step solution:

1. Define a function `maxDepth` that takes a node of type `Node` (representing a node in a tree) as input.
2. Check if the `root` node is `null` (indicating an empty subtree). If so, return 0 as the depth of the subtree is 0.
3. If the `root` node is not `null`, initialize a variable `maxChildDepth` to keep track of the maximum depth among the children of the current node.
4. Iterate through each child node of the `root` node using a `for...of` loop.
5. For each child node, recursively calculate its depth using a recursive call to `maxDepth(child)`.
6. Update the value of `maxChildDepth` by taking the maximum between its current value and the depth of the current child node (`childDepth`).
7. After iterating through all the children nodes, return `maxChildDepth + 1`, indicating the maximum depth of the current subtree.

### Techniques Used:

1. **Recursion:** The algorithm uses recursion to traverse the tree and calculate the maximum depth. The recursive function is applied to each child node.
2. **Looping:** The algorithm uses a `for...of` loop to iterate through the children of the current node.
3. **Math.max:** The algorithm uses the `Math.max` function to compare and update the maximum child depth.

### Summary:

The `maxDepth` function calculates the maximum depth of a tree represented by nodes of type `Node`. It handles both the base case (empty subtree) and the recursive case (non-empty subtree) to calculate the maximum depth. The algorithm returns the maximum depth of the given tree.

## Median of two sorted arrays

### Median of two sorted arrays

```

function findMedianSortedArrays(nums1: number[], nums2: number[]): number {
    if(nums1.length > nums2.length) {
        return findMedianSortedArrays(nums2, nums1);
    }

    const lengthFirst = nums1.length;
    const lengthSecond = nums2.length;

```

```

let start = 0;
let end = lengthFirst;

while(start <= end) {
    let part1 = Math.floor((start+end)/2);
    let part2 = Math.floor((lengthFirst + lengthSecond + 1) / 2) - part1;

    let maxLeftNum1 = part1 === 0 ? Number.MIN_SAFE_INTEGER : nums1[part1 - 1];
    let minRightNum1 = part1 === lengthFirst ? Number.MAX_SAFE_INTEGER : nums1[part1];

    let maxLeftNum2 = part2 === 0 ? Number.MIN_SAFE_INTEGER : nums2[part2 - 1];
    let minRightNum2 = part2 === lengthSecond ? Number.MAX_SAFE_INTEGER : nums2[part2];

    if(maxLeftNum1 <= minRightNum2 && maxLeftNum2 <= minRightNum1) {
        if((lengthFirst + lengthSecond) % 2 == 0) {
            return (Math.max(maxLeftNum1, maxLeftNum2) + Math.min(minRightNum1, minRightNum2)) / 2
        } else {
            return Math.max(maxLeftNum1, maxLeftNum2);
        }
    } else if(maxLeftNum1 > minRightNum2) {
        end = part1 - 1;
    } else {
        start = part1 + 1;
    }
}
};


```

#### Step-by-Step solution:

1. Define a function `findMedianSortedArrays` that takes two sorted arrays `nums1` and `nums2` as input.
2. Check if the length of `nums1` is greater than the length of `nums2`. If it is, swap the arrays and recursively call the function to ensure that `nums1` is always the shorter array.
3. Calculate the lengths of `nums1` and `nums2` and store them in `lengthFirst` and `lengthSecond` variables.
4. Initialize the `start` and `end` pointers for binary search. `start` will initially be 0, and `end` will be the length of `nums1`.
5. Enter a `while` loop that continues as long as `start` is less than or equal to `end`.
6. Inside the loop, calculate the partition points `part1` and `part2` for dividing the two arrays. The goal is to find a partition that divides both arrays into two parts such that the maximum element on the left side is smaller than or equal to the minimum element on the right side.
7. Calculate the maximum left elements (`maxLeftNum1` and `maxLeftNum2`) and minimum right elements (`minRightNum1` and `minRightNum2`) for both arrays based on the calculated partition points.
8. Check if the conditions for a valid partition are met: `maxLeftNum1` is less than or equal to `minRightNum2` and `maxLeftNum2` is less than or equal to `minRightNum1`. This indicates a valid partition.
9. If the total length of the combined arrays is even, calculate and return the median as the average of the maximum of the left elements and the minimum of the right elements.
10. If the total length is odd, return the maximum of the two left elements as the median.

11. If the valid partition conditions are not met, adjust the `start` and `end` pointers based on the comparison between `maxLeftNum1` and `minRightNum2`.
12. If `maxLeftNum1` is greater than `minRightNum2`, move the `end` pointer to the left to reduce the partition in `nums1`.
13. Otherwise, move the `start` pointer to the right to increase the partition in `nums1`.
14. After exiting the loop, the function will return the calculated median.

#### Techniques Used:

1. **Binary Search:** The algorithm uses binary search to find the correct partition points in both arrays.
2. **Conditional Statements:** The algorithm uses conditional statements to handle various cases, such as even and odd total lengths of combined arrays, and adjusting partition pointers.

#### Summary:

The `findMedianSortedArrays` function calculates the median of two sorted arrays `nums1` and `nums2` using binary search and partitioning. It handles different cases for finding the median of even and odd total lengths of combined arrays and adjusts partition points accordingly. The algorithm returns the calculated median of the two arrays.

#### Memo

### Memo

```
function memo(func: any) {
  const cache: any = {};
  return (...args: any[]) =>{
    let key = JSON.stringify(args);
    if(cache[key]) {
      return cache[key];
    }
    const value = func.apply(null, args);
    cache[key] = value;
    return value;
  }
}
```

#### Function solution:

The `memo` function is a higher-order function that takes another function `func` as input and returns a new function with memoization applied.

**Memoization** is an optimization technique where the results of expensive function calls are cached and reused if the same inputs occur again. This can significantly improve the performance of functions that are computationally expensive.

Here's how the `memo` function works:

1. Declare an empty object called `cache` to store memoized results.

2. Return an anonymous function that takes any number of arguments (`...args`) using the rest parameter syntax.
3. Convert the arguments into a unique key by using `JSON.stringify(args)` and store it in the `key` variable.
4. Check if the computed result for the given key is already present in the `cache`. If it is, return the cached result.
5. If the result is not in the cache, execute the original function `func` with the provided arguments using the `apply` method (`func.apply(null, args)`) and store the result in the `value` variable.
6. Store the computed value in the `cache` using the generated key.
7. Finally, return the computed value.

#### Usage:

The `memo` function can be used to wrap any expensive or recursive function, and it will ensure that previously computed results are reused for the same set of input arguments.

For example, if you have a recursive Fibonacci function and you want to improve its performance using memoization:

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

const memoizedFibonacci = memo(fibonacci);

console.log(memoizedFibonacci(10)); // This will be faster than the non-memoized version
```

#### Techniques Used:

1. **Caching/Memoization:** The `memo` function implements the technique of memoization to optimize the performance of expensive or repetitive function calls by caching their results.

#### Summary:

The `memo` function is a higher-order function that adds memoization to another function, allowing it to efficiently cache and reuse computed results for the same input arguments. This can greatly improve the performance of functions that involve expensive computations or recursion.

## Merge Sorted Array

### Merge Sorted Array

```
/**
Do not return anything, modify nums1 in-place instead.
*/
function merge(nums1: number[], m: number, nums2: number[], n: number): void {
  let j = 0;
```

```

for(let i = 0; i <= nums1.length; i++) {
    if(nums1[i] === 0 && nums2[j] !== undefined) {
        nums1[i] = nums2[j];
        j++;
    }
}
nums1.sort((a, b) => a - b);

};

```

The function `merge` is intended to merge two sorted arrays `nums1` and `nums2` into the first array `nums1`. Here's an explanation of how the function works:

1. Initialize a variable `j` to keep track of the index of the elements in `nums2`.
2. Iterate through each element of the array `nums1` using a for loop. We iterate up to `nums1.length` because the array contains both the merged elements and additional zeros.
3. Check if the current element `nums1[i]` is 0 and if there are remaining elements in `nums2` (i.e., `nums2[j] !== undefined`).
4. If the condition is satisfied, update the value of `nums1[i]` with the next element from `nums2` (i.e., `nums2[j]`), and increment `j` to move to the next element in `nums2`.
5. After iterating through the entire array `nums1`, sort the array in ascending order using the `sort` method and a comparison function `(a, b) => a - b`.

The goal of this function is to merge the elements from `nums2` into `nums1` while ensuring that the merged array remains sorted.

**Note:** The initial array lengths `m` and `n` are provided to indicate the valid elements in `nums1` and `nums2` respectively.

#### Techniques Used:

1. **Array Manipulation:** The function modifies the elements of the `nums1` array in place by replacing 0 elements with the elements from `nums2`.
2. **Sorting:** The function sorts the merged `nums1` array after performing the merging operation.

#### Summary:

The `merge` function takes two sorted arrays `nums1` and `nums2`, and it merges the elements from `nums2` into `nums1` while maintaining the sorted order. The function achieves this by iterating through `nums1` and `nums2`, updating elements in `nums1`, and then sorting the merged array.

### Merge two sorted linked lists

### Merge two sorted linked lists

#### Solution 1

```

function mergeLists(head1, head2) {

    let newList = new SinglyLinkedList()

    while (head1 && head2) {
        if (head1.data < head2.data) {
            newList.insertNode(head1.data)
            head1 = head1.next
        } else {
            newList.insertNode(head2.data)
            head2 = head2.next
        }
    }

    newList.tail.next = (head1) ? head1 : head2
    return newList.head;
}

```

The given function `mergeLists` is intended to merge two sorted singly linked lists `head1` and `head2` into a single sorted linked list. Here's an explanation of how the function works:

1. Create a new instance of a singly linked list named `newList`.
2. Use a while loop to iterate while both `head1` and `head2` are not null. This loop compares the data of the current nodes of both lists.
3. If the data of the current node in `head1` is less than the data of the current node in `head2`, insert the data of the current node from `head1` into the `newList`, and then move `head1` to the next node.
4. If the data of the current node in `head2` is less than or equal to the data of the current node in `head1`, insert the data of the current node from `head2` into the `newList`, and then move `head2` to the next node.
5. After the loop finishes, if there are any remaining nodes in either `head1` or `head2`, attach the remaining nodes to the `newList`.
6. Return the `head` of the `newList`, which is the starting node of the merged sorted linked list.

#### Techniques Used:

1. **Linked List Manipulation:** The function iterates through the two input linked lists `head1` and `head2`, and constructs a new merged linked list.
2. **Comparison and Insertion:** The function compares the data of the nodes from `head1` and `head2`, and inserts the smaller data into the `newList`.

#### Summary:

The `mergeLists` function takes two sorted singly linked lists `head1` and `head2`, and it merges the nodes from both lists to create a single sorted linked list. The function achieves this by iteratively comparing node data and constructing the merged list using the `SinglyLinkedList` data structure.

## Solution 2

```

function mergeTwoLists(list1: ListNode | null, list2: ListNode | null): ListNode | null {

    if(!list1) return list2;
    if(!list2) return list1;

    let head = null;
    let temp = head;

    if (list1.val < list2.val) {
        temp = head = new ListNode(list1.val);
        list1 = list1.next;
    } else {
        temp = head = new ListNode(list2.val);
        list2 = list2.next;
    }

    while (list1 && list2) {
        if (list1.val < list2.val) {
            temp.next = new ListNode(list1.val);
            list1 = list1.next;
            temp = temp.next
        } else {
            temp.next = new ListNode(list2.val);
            list2 = list2.next;
            temp = temp.next
        }
    }

    while (list1) {
        temp.next = new ListNode(list1.val);
        list1 = list1.next;
        temp = temp.next;
    }

    while (list2) {
        temp.next = new ListNode(list2.val);
        list2 = list2.next;
        temp = temp.next;
    }

    return head;
};

```

The given function `mergeTwoLists` is intended to merge two sorted linked lists `list1` and `list2` into a single sorted linked list. Here's an explanation of how the function works:

1. First, it handles base cases. If either `list1` or `list2` is empty, it directly returns the other list as the merged result.
2. Next, it initializes variables `head` and `temp` to keep track of the merged list.
3. It compares the values of the first nodes of `list1` and `list2` and creates the head node of the merged list accordingly. The `temp` variable is updated to point to the head node.
4. Using a `while` loop, the function iterates through both `list1` and `list2` while comparing the values of their nodes.

5. Inside the loop, if the value of the current node in `list1` is less than the value of the current node in `list2`, a new node with the value of `list1` is created and attached to the `temp.next`. Then, `list1` and `temp` are moved to the next nodes.
6. If the value of the current node in `list2` is less than or equal to the value of the current node in `list1`, a new node with the value of `list2` is created and attached to the `temp.next`. Then, `list2` and `temp` are moved to the next nodes.
7. After merging both lists, if there are any remaining nodes in `list1`, they are appended to the merged list.
8. Similarly, if there are any remaining nodes in `list2`, they are also appended to the merged list.
9. Finally, the merged list is returned by returning the `head` node.

### Techniques Used:

1. **Linked List Manipulation:** The function constructs the merged list by iterating through both input linked lists `list1` and `list2`.
2. **Comparison and Insertion:** The function compares the values of the nodes from `list1` and `list2` and inserts the smaller value into the merged list.

### Summary:

The `mergeTwoLists` function takes two sorted linked lists `list1` and `list2`, and it merges the nodes from both lists to create a single sorted linked list. The function iterates through the nodes of both lists, comparing values and constructing the merged list using linked list manipulation.

## Minimum Absolute Difference in BST

### Minimum Absolute Difference in BST

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */
function getMinimumDifference(root: TreeNode | null): number {
    let minDiff = Infinity;
    let prevVal = null;

    const inorderTraversal = (node: TreeNode | null): void => {
        if (node === null) {
            return;
        }
        inorderTraversal(node.left);
        if (prevVal !== null) {
            minDiff = Math.min(minDiff, Math.abs(prevVal - node.val));
        }
        prevVal = node.val;
        inorderTraversal(node.right);
    };
    inorderTraversal(root);
    return minDiff;
}
```

```

        return;
    }

    inorderTraversal(node.left);

    if (prevVal !== null) {
        minDiff = Math.min(minDiff, node.val - prevVal);
    }
    prevVal = node.val;

    inorderTraversal(node.right);
};

inorderTraversal(root);

return minDiff;
};

```

The given function `getMinimumDifference` is intended to find the minimum absolute difference between any two nodes in a binary search tree (BST). Here's how the function works:

1. The function initializes `minDiff` with a value of positive infinity (to store the minimum absolute difference) and `prevVal` with `null` (to keep track of the previous visited node's value during the in-order traversal).
2. The function defines an `inorderTraversal` function that performs an in-order traversal of the BST. It takes a `node` as an argument and does the following:
  - If `node` is `null`, it returns, indicating the end of the traversal.
  - Recursively calls `inorderTraversal` on the `left` child of `node`.
  - Calculates the absolute difference between the current node's value (`node.val`) and the previous visited node's value (`prevVal`). If `prevVal` is not `null`, it updates `minDiff` with the minimum of the current `minDiff` and the calculated absolute difference.
  - Updates `prevVal` to the current node's value.
  - Recursively calls `inorderTraversal` on the `right` child of `node`.
3. The main function, `getMinimumDifference`, starts by invoking the `inorderTraversal` function on the root node of the BST.
4. Finally, the function returns the calculated `minDiff`, which represents the minimum absolute difference between any two nodes in the BST.

#### Techniques Used:

1. **Binary Search Tree Traversal (In-Order):** The function uses in-order traversal to visit nodes in increasing order of their values. This is a key property of BSTs, and it allows efficient calculation of the minimum absolute difference between adjacent nodes.
2. **Tracking Previous Value:** The function keeps track of the previous visited node's value (`prevVal`) during the in-order traversal. This is essential for calculating the absolute differences between adjacent nodes.
3. **Updating Minimum Difference:** The function calculates the absolute difference between adjacent nodes and updates the `minDiff` with the minimum value encountered so far.

### Summary:

The `getMinimumDifference` function utilizes in-order traversal on a binary search tree to calculate the minimum absolute difference between any two nodes. It takes advantage of the BST's properties to efficiently traverse the tree and calculate the required difference.

### Minimum Depth of Binary Tree

### Minimum Depth of Binary Tree

```
/**  
 * Definition for a binary tree node.  
 * class TreeNode {  
 *     val: number  
 *     left: TreeNode | null  
 *     right: TreeNode | null  
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {  
 *         this.val = (val===undefined ? 0 : val)  
 *         this.left = (left===undefined ? null : left)  
 *         this.right = (right===undefined ? null : right)  
 *     }  
 * }  
 */  
  
function minDepth(root: TreeNode | null): number {  
    let depth = 1;  
  
    if(root == null) return 0;  
    let que = []  
    que.push(root);  
    while(que.length > 0){  
        let size = que.length;  
        while(size>0){  
            let node = que.shift();  
  
            if(node.left == null && node.right == null) return depth;  
  
            if(node.left != null) que.push(node.left);  
  
            if(node.right != null) que.push(node.right);  
            size--;  
        }  
        depth++;  
    }  
    return depth;  
};
```

The given function `minDepth` aims to find the minimum depth of a binary tree, where the depth is defined as the number of nodes along the shortest path from the root node to any leaf node. Here's a step-by-step explanation of how the function works:

1. The function initializes `depth` with 1, which will be used to track the current depth level of the traversal.

2. The function starts by checking if the `root` node is `null`. If it is, it means the tree is empty, so the function returns 0.
3. Otherwise, the function initializes a queue `que` to perform a level-order traversal (Breadth-First Search) of the binary tree. It pushes the `root` node into the queue.
4. The function enters a loop that continues until the queue is empty. This loop represents the level-order traversal of the tree.
5. Inside the loop, the function retrieves the current size of the queue using `que.length`. This indicates the number of nodes at the current level.
6. The function enters another loop that processes each node at the current level. For each node, it performs the following steps:
  - Dequeues (removes) the front node from the queue.
  - Checks if the current node is a leaf node (both left and right children are `null`). If it is, the function returns the current `depth`, as this represents the minimum depth to a leaf node.
  - If the current node has a left child, it enqueues the left child into the queue.
  - If the current node has a right child, it enqueues the right child into the queue.
7. After processing all nodes at the current level, the function increments the `depth` by 1 to move to the next level.
8. The loop continues until all levels of the tree have been traversed.
9. Finally, the function returns the `depth`, which represents the minimum depth of the binary tree.

#### Techniques Used:

1. **Breadth-First Search (BFS):** The function utilizes BFS to traverse the binary tree level by level. This is important for finding the minimum depth, as BFS ensures that the shortest path to a leaf node is explored first.
2. **Queue Data Structure:** The function uses a queue to keep track of nodes at each level during BFS. This allows for efficient processing of nodes level by level.

#### Summary:

The `minDepth` function employs BFS to traverse the binary tree level by level and calculates the minimum depth by finding the shortest path to a leaf node. It efficiently utilizes a queue data structure to keep track of nodes at each level and returns the minimum depth of the tree.

## Minimum Index Sum of Two Lists

### Minimum Index Sum of Two Lists

```
function findRestaurant(list1: string[], list2: string[]): string[] {
  const indexSumMap: Map<string, number> = new Map();
  let minIndexSum = Infinity;

  for (let i = 0; i < list1.length; i++) {
    const restaurant = list1[i];
    if (list2.includes(restaurant)) {
```

```

        const indexSum = i + list2.indexOf(restaurant);
        indexSumMap.set(restaurant, indexSum);
        minIndexSum = Math.min(minIndexSum, indexSum);
    }
}

const result: string[] = [];
for (const [restaurant, indexSum] of indexSumMap) {
    if (indexSum === minIndexSum) {
        result.push(restaurant);
    }
}

return result;
};

```

#### 1. Initialize Data Structures:

- Create a `Map` called `indexSumMap` to store the sum of indices for common restaurants between the two lists.
- Initialize a variable `minIndexSum` to `Infinity` to keep track of the minimum index sum.

#### 2. Iterate Through First List:

- Use a `for` loop to iterate through each restaurant in the first list (`list1`).
- For each restaurant:
  - Check if it also exists in the second list (`list2`).

#### 3. Calculate Index Sum:

- If the restaurant is common between both lists:
  - Calculate the index sum by adding the current index in `list1` (`i`) and the index of the restaurant in `list2` (using `list2.indexOf(restaurant)`).

#### 4. Update Data Structures:

- Store the calculated index sum in the `indexSumMap` using the restaurant name as the key.
- Update `minIndexSum` to be the minimum value between its current value and the calculated index sum using `Math.min()`.

#### 5. Initialize Result Array:

- Create an empty array called `result` to store the final result.

#### 6. Iterate Through `indexSumMap`:

- Use a `for...of` loop to iterate through each entry in `indexSumMap`, which contains restaurant names and their corresponding index sums.

#### 7. Filter by Minimum Index Sum:

- For each entry, check if the index sum is equal to the minimum index sum.
- If it is, add the restaurant name to the `result` array.

#### 8. Return Result:

- Return the `result` array containing the names of restaurants with the minimum index sum.

## Missing number

### Missing number

```
function missingNumber(nums: number[]): number {
    let sum = 0;
    let expected = (1 + nums.length) * nums.length / 2;
    for (let i = 0; i < nums.length; i++) {
        sum += nums[i];
    }
    return expected - sum;
};
```

The given function `missingNumber` is designed to find the missing number in an array of consecutive integers from 0 to n. Here's a step-by-step explanation of how the function works:

1. The function initializes `sum` to 0. This variable will be used to calculate the sum of the elements in the input array.
2. The function calculates the expected sum of the consecutive integers using the formula `(1 + nums.length) * nums.length / 2`. This formula calculates the sum of integers from 1 to `nums.length` using the arithmetic series sum formula.
3. The function enters a loop that iterates through each element in the `nums` array. For each element, it adds the value to the `sum` variable.
4. After iterating through all elements in the array, the function subtracts the calculated `sum` from the `expected` sum. This operation effectively finds the missing number in the array.
5. Finally, the function returns the calculated missing number.

#### Techniques Used:

1. **Arithmetic Series Sum Formula:** The function uses the arithmetic series sum formula to calculate the expected sum of consecutive integers. This formula simplifies the calculation of the sum and allows for an efficient solution to finding the missing number.

#### Summary:

The `missingNumber` function calculates the missing number in an array of consecutive integers by finding the difference between the expected sum and the actual sum of the elements. It leverages the arithmetic series sum formula to optimize the computation of the expected sum.

## Move Zeroes

### Move Zeroes

```

/**
Do not return anything, modify nums in-place instead.
*/
function moveZeroes(nums: number[]): void {
    let j = 0;
    for (let i = 0; i < nums.length; i++) {
        if (nums[i] !== 0) {
            if (i !== j) {
                let tmp = nums[i];
                nums[i] = nums[j];
                nums[j] = tmp;
            }
            j++;
        }
    }
}

```

The given function `moveZeroes` is designed to move all the zeroes in an array to the end while preserving the relative order of the non-zero elements. Here's a step-by-step explanation of how the function works:

1. The function initializes two pointers, `i` and `j`, both starting at 0. The pointer `i` iterates through the array to find non-zero elements, while the pointer `j` keeps track of the position where the next non-zero element should be placed.
2. The function enters a loop that iterates through each element in the `nums` array.
3. For each element, if it is not equal to zero (`nums[i] !== 0`), the function checks if the current positions of `i` and `j` are different (`i !== j`). If they are different, it means that a non-zero element needs to be moved to the position indicated by `j`.
4. The function then swaps the element at position `i` with the element at position `j`. This effectively moves the non-zero element to the correct position while maintaining the relative order of non-zero elements.
5. After swapping, the pointer `j` is incremented by 1 to indicate that the next non-zero element should be placed at the next position.
6. The loop continues until all elements have been processed.
7. Once the loop is complete, all non-zero elements have been moved to their correct positions, and the remaining positions from `j` to the end of the array are filled with zeroes.

#### **Techniques Used:**

1. **Two-Pointers Approach:** The function uses a two-pointers approach to iterate through the array. The pointer `i` finds non-zero elements, and the pointer `j` keeps track of where the next non-zero element should be placed. Swapping is used to move elements to their correct positions.

#### **Summary:**

The `moveZeroes` function rearranges the elements in the input array `nums` in such a way that all zeroes are moved to the end of the array while preserving the relative order of non-zero elements. It achieves this by using a two-pointers approach and swapping elements as needed.

## Multiples of 3 and 5

### Multiples of 3 and 5

```
function multiplesOf3and5(number: number): number {
    let sum: number = 0;
    for (let i: number = 0; i < number; i++) {
        if (i % 3 === 0 || i % 5 === 0) {
            sum += i;
        }
    }
    return sum;
}

multiplesOf3and5(1000);
```

#### Step-by-Step:

##### 1. Initialization:

- `let sum: number = 0;`: Initialize a variable `sum` to store the sum of multiples of 3 and 5. Initially set it to 0.

##### 2. Loop through Numbers:

- `for (let i: number = 0; i < number; i++) {`: Start a loop from `i = 0` and continue until `i` is less than the given `number`.

##### 3. Check Multiples:

- `if (i % 3 === 0 || i % 5 === 0) {`: Check if the current number `i` is divisible by 3 or 5 (i.e., if the remainder of the division by 3 or 5 is 0).

##### 4. Sum Calculation:

- `sum += i;`: If the current number is a multiple of 3 or 5, add it to the `sum` variable.

##### 5. Return the Sum:

- `return sum;`: After the loop is complete, return the calculated sum of multiples of 3 and 5 within the given range.

##### 6. Function Call:

- `multiplesOf3and5(1000);`: Call the function with the argument 1000. In this case, the function will find and return the sum of all multiples of 3 or 5 below 1000.

The function calculates the sum of multiples of 3 or 5 up to the given number (1000 in this case) and returns the result.

## N-ary Tree Postorder Traversal

# N-ary Tree Postorder Traversal

```
/**  
 * Definition for node.  
 * class Node {  
 *     val: number  
 *     children: Node[]  
 *     constructor(val?: number) {  
 *         this.val = (val===undefined ? 0 : val)  
 *         this.children = []  
 *     }  
 * }  
 */  
  
function postorder(root: Node | null): number[] {  
    const result: number[] = [];  
  
    function traverse(node: Node | null) {  
        if (node === null) {  
            return;  
        }  
  
        // Traverse children  
        for (const child of node.children) {  
            traverse(child);  
        }  
  
        // Process current node  
        result.push(node.val);  
    }  
  
    traverse(root);  
    return result;  
};
```

## Step-by-Step:

### 1. Initialization:

- Initialize an array called **result** to store the values in postorder traversal.

### 2. Define a Recursive Function for Traversal:

- The **traverse** function is defined to perform the recursive traversal.

### 3. Check if the Node is Null:

- Within the **traverse** function, check if the current node is null. If so, return, as there's nothing to process.

### 4. Traverse Children:

- Use a `for...of` loop to iterate through each child of the current node.
- Recursively call the `traverse` function on each child. This step ensures that the entire subtree rooted at each child is processed before the current node.

#### 5. Process Current Node:

- After traversing all children, process the current node by pushing its value onto the `result` array.
- This ensures that the current node is processed after its children, following the postorder traversal strategy.

#### 6. Start the Traversal with the Root Node:

- Call the `traverse` function with the provided `root` parameter to start the postorder traversal from the root of the tree.

#### 7. Return the Result Array:

- Return the final `result` array containing the postorder traversal of the tree.

The algorithm uses a recursive approach to perform postorder traversal. It processes all children of a node before processing the node itself, ensuring a postorder traversal sequence. The result array collects the values in the order they are visited.

## N-ary Tree Preorder Traversal

## N-ary Tree Preorder Traversal

```
/**
 * Definition for node.
 * class Node {
 *   val: number
 *   children: Node[]
 *   constructor(val?: number) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.children = []
 *   }
 * }
 */

function preorder(root: Node | null): number[] {
  const result: number[] = [];
  if (!root) {
    return result;
  }

  const stack: Node[] = [root];

  while (stack.length > 0) {
    const currentNode = stack.pop()!;
    result.push(currentNode.val);

    for (let i = currentNode.children.length - 1; i >= 0; i--) {
      stack.push(currentNode.children[i]);
    }
  }

  return result;
}
```

```

        stack.push(currentNode.children[i]);
    }
}

return result;
}

```

### Step-by-Step:

#### 1. Function Definition:

- The function `preorder` takes a parameter `root` which is either a `Node` or `null`.
- It returns an array of numbers representing the preorder traversal of the tree.

#### 2. Initialize Result Array:

- Create an empty array called `result` to store the preorder traversal.

#### 3. Check if the Tree is Empty:

- If the root is `null` (indicating an empty tree), return the empty result array.

#### 4. Initialize Stack:

- Create a stack called `stack` to help with the iterative traversal.
- Push the root node onto the stack to start the traversal.

#### 5. Iterative Traversal Using Stack:

- Enter a while loop that continues until the stack is empty.
- Inside the loop, pop the top node (`currentNode`) from the stack.
- Push the value of the `currentNode` onto the result array.

#### 6. Push Children onto Stack:

- Iterate through the children of the `currentNode` in reverse order (from right to left).
- Push each child onto the stack.

#### 7. Repeat Until Stack is Empty:

- Continue the loop until the stack becomes empty. At this point, all nodes in the tree have been processed.

#### 8. Return Result:

- Return the final result array containing the preorder traversal of the tree.

The algorithm uses a stack to keep track of the nodes to be processed. It starts with the root, processes each node and its children, and continues until all nodes are visited. The order of processing is such that a node's value is added to the result array before its children are explored. This follows the preorder traversal strategy for a tree.

## Next Greater Element I

## Next Greater Element I

```

function nextGreaterElement(nums1: number[], nums2: number[]): number[] {
    const nextGreaterMap = new Map<number, number>();
    const stack: number[] = [];

    for (let i = 0; i < nums2.length; i++) {
        while (stack.length > 0 && nums2[i] > stack[stack.length - 1]) {
            const smallerNum = stack.pop();
            nextGreaterMap.set(smallerNum, nums2[i]);
        }
        stack.push(nums2[i]);
    }

    const result: number[] = [];

    for (let num of nums1) {
        result.push(nextGreaterMap.get(num) || -1);
    }

    return result;
};

```

The `nextGreaterElement` function is designed to find the next greater element for each element in `nums1` within the array `nums2`. It uses a stack-based approach and a map to achieve this. Let's break down the function step by step:

1. The function initializes a map named `nextGreaterMap`, which will store the next greater element for each element in `nums2`. It also initializes an empty stack named `stack`.
2. The function then loops through each element in `nums2`.
3. Inside the loop, the function checks if the `stack` is not empty and the current element in `nums2` (`nums2[i]`) is greater than the top element of the `stack`. If this condition is met, it means that the current element is the next greater element for the top element(s) in the stack. The function repeatedly pops elements from the stack and updates their next greater element in the `nextGreaterMap` until the condition is no longer satisfied.
4. After updating the `nextGreaterMap`, the current element is pushed onto the stack.
5. Once the loop through `nums2` is complete, the `nextGreaterMap` will contain the next greater element for each element in `nums2`.
6. The function initializes an empty array named `result`, which will store the next greater elements for each element in `nums1`.
7. The function loops through each element in `nums1`.
8. Inside the loop, the function uses the `nextGreaterMap` to retrieve the next greater element for the current element in `nums1`. If the element is found in the map, it is pushed onto the `result` array. If the element is not found, `-1` is pushed onto the array.
9. After processing all elements in `nums1`, the `result` array will contain the next greater elements for each element.
10. Finally, the function returns the `result` array.

#### **Techniques Used:**

1. **Stack:** The function uses a stack to keep track of elements in descending order while finding the next greater element in `nums2`.
2. **Map:** The function uses a map (`nextGreaterMap`) to store the next greater element for each element in `nums2`, allowing quick lookups when processing `nums1`.

### **Summary:**

The `nextGreaterElement` function efficiently finds the next greater element for each element in `nums1` using a stack-based approach and a map to store the results. It provides a solution with a time complexity of  $O(N)$ , where  $N$  is the length of `nums2`.

## Nim Game

### Nim Game

```
function canWinNim(n: number): boolean {
    return !(n % 4 == 0);
};
```

The `canWinNim` function determines whether a player can win the game of Nim with a given number of stones (`n`) based on a specific rule. In the game of Nim, two players take turns removing stones from a pile. On each turn, a player can remove 1 to 3 stones. The player who removes the last stone wins the game.

The function's logic is based on the observation that if there are 4 stones left, the player whose turn it is will lose, no matter what move they make. This is because regardless of whether they remove 1, 2, or 3 stones, the other player can always take the remaining stones and win the game. Therefore, the function returns `false` for `n` divisible by 4, indicating that the current player cannot win the game.

For all other values of `n`, the function returns `true`, indicating that the current player can make a winning move and eventually force their opponent to be the one left with 4 stones, ensuring their victory.

### Techniques Used:

- **Mathematical Logic:** The function employs a simple mathematical rule to determine whether the current player can win the game based on the number of stones (`n`) left.

### **Summary:**

The `canWinNim` function provides an efficient way to determine the winning strategy for the game of Nim by leveraging a mathematical rule related to the number of stones remaining in a pile.

## Number Complement

### Number Complement

```
function findComplement(num: number): number {
    let mask = 1;

    while (mask < num) {
```

```

        mask = (mask << 1) | 1;
    }

    return num ^ mask;
}

```

The `findComplement` function takes an integer `num` as input and returns its bitwise complement. The bitwise complement of an integer is obtained by flipping all its binary digits, changing 0s to 1s and vice versa.

Here's how the function works:

1. It initializes a variable `mask` to 1. This mask will be used to set all the bits in the binary representation of `num` to 1.
2. It enters a loop that continues as long as `mask` is less than `num`. In each iteration of the loop, the value of `mask` is shifted left by 1 (effectively adding a 0 at the least significant bit) and then a bitwise OR operation with 1 is performed. This sets all the bits of `mask` to 1 up to the highest bit position where `num` has a 1.
3. After the loop, the bitwise XOR operation (`^`) is performed between `num` and `mask`. This effectively flips all the bits in the binary representation of `num` where `mask` has 1s.
4. The final result of the XOR operation is the bitwise complement of `num`, and this value is returned.

#### Techniques Used:

- **Bit Manipulation:** The function uses bitwise shifting and bitwise XOR operations to manipulate the individual bits of the input integer to find its complement.

#### Summary:

The `findComplement` function employs bit manipulation techniques to efficiently compute the bitwise complement of an integer. It does so by using a mask that has 1s in positions corresponding to the significant bits of the input integer and then performing a bitwise XOR operation to flip the bits.

### Number of 1 Bits

### Number of 1 Bits

```

function hammingWeight(n: number): number {
    let pivot = 1;
    let sum = 0;
    for (let i = 0; i < 32; i ++){
        let p = pivot << i;
        if ((n & p) == p)
            sum++;
    }
    return sum;
};

```

The `hammingWeight` function takes an integer `n` as input and returns the number of '1' bits (also known as the Hamming weight) in its binary representation.

Here's how the function works:

1. It initializes a variable `pivot` to 1. This will be used to create a bitmask with only one bit set at a particular position.
2. It initializes a variable `sum` to 0. This will be used to keep track of the number of '1' bits.
3. It enters a loop that iterates 32 times (since an integer in JavaScript is represented using 32 bits). In each iteration, it shifts the `pivot` left by `i` positions to create a bitmask with only the `i`-th bit set to 1.
4. It performs a bitwise AND operation (`&`) between the input integer `n` and the bitmask `p`. If the result of this operation is equal to `p`, it means that the `i`-th bit of `n` is a '1', so the `sum` is incremented.
5. After the loop, the `sum` variable contains the count of '1' bits in the binary representation of the input integer, and this value is returned.

### Techniques Used:

- **Bit Manipulation:** The function uses bitwise shifting and bitwise AND operations to check the individual bits of the input integer and count the '1' bits.

### Summary:

The `hammingWeight` function effectively counts the number of '1' bits in the binary representation of an integer using bitwise manipulation techniques. It iterates through each bit position and checks if the bit is set, incrementing the count accordingly.

## Number of Segments in a String

### Number of Segments in a String

```
function countSegments(s: string): number {
  s = s.trim();
  if(s.length == 0) {
    return 0
  } else {
    return s.replace(/\s+/g, ' ').trim().split(' ').length
  }
};
```

The `countSegments` function takes a string `s` as input and returns the number of segments in the string, where a segment is defined as a contiguous sequence of non-space characters.

Here's how the function works:

1. It trims the input string `s` using the `trim` method to remove leading and trailing whitespace.
2. It checks if the trimmed string has a length of 0. If it does, it means there are no segments, so the function returns 0.
3. Otherwise, it uses regular expressions to replace multiple consecutive spaces with a single space using the `replace` method and the regular expression `/\s+/g`. This ensures that there is only one space between each segment.
4. It then trims the modified string again to remove any leading or trailing spaces.

- Finally, it splits the trimmed string into an array of segments using the `split` method with a space (' ') as the delimiter. The length of the resulting array is the number of segments, and this value is returned.

#### Techniques Used:

- String Manipulation:** The function uses string manipulation techniques such as trimming, replacing, and splitting to process the input string and count the segments.
- Regular Expressions:** Regular expressions are used to replace multiple consecutive spaces with a single space.

#### Summary:

The `countSegments` function effectively counts the number of segments in a given string by trimming, processing, and splitting the string based on spaces. It uses string manipulation and regular expressions to achieve this.

## Pairs

### Pairs

```
function pairs(k: number, arr: number[]): number {
  arr.sort((a, b) => a - b);
  let output = 0;
  let i = 0;
  let j = 0;
  while (j < arr.length) {
    let difference = arr[j] - arr[i];
    if (difference === k) {
      output++;
      j++;
      i++;
    } else if (difference > k) {
      i++;
    } else if (difference < k) {
      j++;
    }
  }
  return output;
}
```

The `pairs` function takes an integer `k` and an array of integers `arr` as input, and it returns the count of pairs of integers from the array whose absolute difference is equal to `k`.

Here's how the function works:

- It first sorts the input array `arr` in ascending order using the `sort` method and a comparator function.
- It initializes variables `output` to 0, `i` to 0, and `j` to 0. These variables are used to keep track of the indices of the elements being compared.

3. It enters a `while` loop that continues as long as the `j` index is less than the length of the array.
4. Inside the loop, it calculates the absolute difference between the elements at indices `i` and `j`, which is `arr[j] - arr[i]`.
5. If the difference is equal to `k`, it means a valid pair is found, so it increments the `output` counter and increments both `i` and `j` indices to explore the next possible pair.
6. If the difference is greater than `k`, it increments the `i` index to move towards a smaller difference.
7. If the difference is less than `k`, it increments the `j` index to move towards a larger difference.
8. The loop continues until all possible pairs have been considered.
9. Finally, the function returns the value of the `output` counter, which represents the count of pairs with an absolute difference of `k`.

### Techniques Used:

- **Array Sorting:** The function sorts the input array in ascending order to make it easier to find pairs with a specific difference.
- **Two-Pointers Approach:** The function uses two pointers, `i` and `j`, to traverse the sorted array and find pairs with the desired difference.

### Summary:

The `pairs` function effectively counts and returns the number of pairs of integers in the sorted array that have an absolute difference of `k`. It uses array sorting and a two-pointers approach to achieve this efficiently.

- Go back

## Pairwise

### Pairwise

```
export function pairwise(arr:number[], arg:number) {
    const index = [];

    for (let a in arr) {
        let temp = arr[a];

        for (let i = 1; i < arr.length; i++) {
            let temp2 = arr[i];
            if (temp + temp2 === arg && i > +a && index.indexOf(+a) === -1 && index.indexOf(+i) === -1)
                index.push(+a, +i);
            break;
        }
    }
    if (index.length >= 1) {
        const addAll = (a: any, b: any) => {
            return a + b;
        }
        return addAll(...index);
    }
}
```

```

    };
    return index.reduce(addAll);
} else
    return 0;
}

let res = pairwise([1, 3, 2, 4], 4);
console.log(res);

```

The `pairwise` function takes an array of integers `arr` and an integer `arg` as input, and it returns the sum of the indices of pairs in the array that add up to the given `arg`.

Here's how the function works:

1. It initializes an empty array `index` to store the indices of pairs that satisfy the condition.
2. It uses a nested loop to iterate over each element in the array. The outer loop iterates over the elements with index `a`, and the inner loop iterates over the elements starting from index `i = 1`. This is done to find pairs that haven't been considered before.
3. Inside the inner loop, it calculates the sum of the current element `temp` and the element at index `i` (`temp2`). If the sum is equal to the target `arg`, and the indices `i` and `a` are valid (i.e., `i > +a`), and the indices `a` and `i` are not already present in the `index` array, then it adds both `a` and `i` to the `index` array and breaks out of the loop.
4. After finding all the valid pairs, it checks if there is at least one pair in the `index` array.
5. If there are pairs in the `index` array, it defines a helper function `addAll` that takes two arguments and returns their sum. Then, it uses the `reduce` function with the `addAll` function to calculate the sum of all the indices in the `index` array and returns the result.
6. If there are no valid pairs, it returns 0.
7. Finally, it calls the `pairwise` function with the example input `[1, 3, 2, 4]` and `4`, and logs the result to the console.

### Techniques Used:

- **Nested Loops:** The function uses nested loops to compare each pair of elements in the array to find pairs with the desired sum.
- **Array Manipulation:** The function manipulates the `index` array to keep track of the indices of valid pairs.
- **Reduce Function:** The function uses the `reduce` function to calculate the sum of the indices in the `index` array.

### Summary:

The `pairwise` function effectively finds pairs of elements in the array that add up to the given target `arg`, calculates the sum of their indices, and returns the result. It uses nested loops, array manipulation, and the `reduce` function to achieve this.

## Palindrome linked list

```
/**  
 * Definition for singly-linked list.  
 * class ListNode {  
 *     val: number  
 *     next: ListNode | null  
 *     constructor(val?: number, next?: ListNode | null) {  
 *         this.val = (val===undefined ? 0 : val)  
 *         this.next = (next===undefined ? null : next)  
 *     }  
 * }  
 */  
  
function isPalindrome(head: ListNode | null): boolean {  
    const isP = (rHead) => {  
        if (rHead == null) {  
            return true;  
        }  
  
        const next = isP(rHead.next);  
  
        const valid = rHead.val === head.val;  
  
        head = head.next;  
        return next && valid;  
    }  
    return isP(head);  
};
```

The `isPalindrome` function determines whether a given singly-linked list is a palindrome or not.

Here's how the function works:

1. It defines an inner recursive function `isP` that takes a single argument `rHead`, which represents the current node being checked in the reverse order.
2. If `rHead` is `null`, it means we have reached the end of the reversed list, so it returns `true` since there is nothing more to compare.
3. Inside the `isP` function, it recursively calls itself with the next node in reverse order (`rHead.next`).
4. After the recursive call, it compares the value of the current node `rHead.val` with the value of the corresponding node in the original list `head.val`. If they are not equal, it means the linked list is not a palindrome, and it returns `false`.
5. If the values are equal, it updates the `head` pointer to the next node in the original list (`head.next`) to continue the comparison.
6. The `next` and `valid` values are combined using the logical AND (`&&`) operator. This ensures that both the recursive call (`next`) and the current value comparison (`valid`) are true for the entire linked list to be considered a palindrome.

7. The inner recursive function `isP` returns `true` or `false` based on the palindrome check for the given node.
8. The outer function `isPalindrome` returns the result of the inner recursive function `isP(head)`, which checks if the entire linked list is a palindrome.

#### Techniques Used:

- **Recursion:** The function uses recursion to traverse and compare the linked list in reverse order.
- **Pointer Manipulation:** The function uses pointers (`rHead` and `head`) to traverse and compare nodes in the reversed and original linked lists.
- **Logical Operators:** The function uses logical AND (`&&`) operator to combine the results of the recursive call and the value comparison.

#### Summary:

The `isPalindrome` function uses recursion and pointer manipulation to determine whether a given singly-linked list is a palindrome. It compares the values of nodes in the reversed and original order and returns `true` if the entire list is a palindrome and `false` otherwise.

#### Palindrome number

### Palindrome number

```
class Solution {
    public boolean isPalindrome(int x) {
        if(x < 0) {
            return false;
        }
        int number = x;
        int reverse = 0;
        while(number > 0) {
            reverse = reverse * 10 + number % 10;
            number /= 10;
        }
        return x == reverse;
    }
}
```

The `Solution` class contains a method `isPalindrome` that determines whether a given integer `x` is a palindrome or not.

Here's how the method works:

1. It first checks if the input integer `x` is negative. If it's negative, it immediately returns `false` because negative numbers cannot be palindromes.
2. It creates two integer variables: `number` (initialized with the value of `x`) and `reverse` (initialized with 0). The `number` variable will be used to extract the digits from the input integer, and the `reverse` variable will be used to build the reversed number.

3. It enters a loop that continues as long as `number` is greater than 0. Inside the loop:
  - It updates the `reverse` variable by multiplying it by 10 and adding the last digit of `number` (`number % 10`).
  - It divides the `number` by 10 to remove the last digit.
4. After the loop finishes, the value of `reverse` will be the integer obtained by reversing the digits of the input `x`.
5. Finally, it compares the original input `x` with the reversed value `reverse`. If they are equal, it means the input integer is a palindrome, and the method returns `true`. Otherwise, it returns `false`.

#### Techniques Used:

- **Looping and Arithmetic Operations:** The method uses a loop to reverse the digits of the input integer by repeatedly extracting the last digit and updating the reverse value.
- **Comparisons:** The method compares the original input integer with the reversed value to determine if it's a palindrome.

#### Summary:

The `isPalindrome` method in the `Solution` class checks if a given integer is a palindrome by reversing its digits and comparing it with the original input. It returns `true` if the integer is a palindrome and `false` otherwise.

- Go back

## Pangram

## Pangram

```
function isPangram(string:string){
  const set = new Set();
  string.toLowerCase().split('').forEach((letter:string) => {
    if(/^[A-Za-z]+$/i.test(letter)) {
      set.add(letter);
    }
  });
  return [...set].length === 26;
}
```

The `isPangram` function checks whether a given string is a pangram, which is a sentence that contains every letter of the alphabet at least once. Here's how the function works:

1. It initializes an empty `Set` named `set` to keep track of unique letters encountered in the string.
2. It converts the input `string` to lowercase using the `toLowerCase()` method to ensure that the function is case-insensitive.
3. It splits the lowercase string into an array of characters using the `split('')` method.
4. It iterates through each letter using the `forEach` method on the array. Inside the loop:

- It uses a regular expression (`/^ [A-Za-z] +$/`) to test if the current letter is an alphabetical character (uppercase or lowercase).
  - If the letter matches the regular expression, it adds the lowercase version of the letter to the `set`.
5. After iterating through all letters, the function converts the `set` to an array using the spread operator `[...set]` and calculates its length.
  6. It checks if the length of the array is equal to 26. If it is, it means that all 26 unique letters of the alphabet were encountered in the input string, and the function returns `true`, indicating that the input string is a pangram. Otherwise, it returns `false`.

### Techniques Used:

- **Set Data Structure:** The `Set` data structure is used to store unique letters encountered in the string.
- **Regular Expression:** A regular expression is used to test whether a character is an alphabetical character.
- **String Manipulation:** The function converts the input string to lowercase and splits it into an array of characters for processing.

### Summary:

The `isPangram` function determines if a given string is a pangram by checking if it contains all 26 unique letters of the alphabet at least once. It uses a `Set` to track encountered letters and a regular expression to identify alphabetical characters.

## Pascal triangle 2

```
function generate(numRows: number): number[][] {
  const pascal: any = [];
  for (let i = 0; i < numRows; i++) {
    pascal[i] = [];
    pascal[i][0] = 1;
    for (let j = 1; j < i; j++) {
      pascal[i][j] = pascal[i - 1][j - 1] + pascal[i - 1][j];
    }
    pascal[i][i] = 1;
  }
  return pascal;
}
```

The `generate` function generates Pascal's Triangle up to a specified number of rows `numRows`. Pascal's Triangle is a triangular array of binomial coefficients, where each element at row `i` and column `j` is the sum of the two elements directly above it in the previous row (`i-1`) at columns `j` and `j-1`.

Here's how the function works:

1. It initializes an empty array named `pascal` to hold the triangle's elements.

2. It iterates through each row from 0 to `numRows` - 1 using a loop. For each row:
  - It initializes an empty array `pascal[i]` to hold the elements of the current row.
  - It sets the first element `pascal[i][0]` of the row to 1.
3. For each row, starting from the second row (`i = 1`), it iterates through each column `j` from 1 to `i - 1`.
  1. For each column:
    - It calculates the current element `pascal[i][j]` by summing the elements from the previous row (`i-1`) at columns `j` and `j-1`.
  4. After calculating the middle elements of the current row, it sets the last element `pascal[i][i]` of the row to 1.
  5. Finally, the function returns the `pascal` array, which now holds Pascal's Triangle up to the specified number of rows.

### Techniques Used:

- **Nested Loops:** The function uses nested loops to iterate through each row and column of Pascal's Triangle.
- **Array Initialization and Manipulation:** The function initializes and manipulates arrays to store the elements of Pascal's Triangle.
- **Mathematical Logic:** The function uses mathematical logic to calculate binomial coefficients and generate Pascal's Triangle.

### Summary:

The `generate` function generates Pascal's Triangle up to the specified number of rows by calculating binomial coefficients and organizing them into a triangular array. It employs nested loops and array manipulation techniques to achieve this.

- Go back

### Pascal triangle 3

```
function getRow(rowIndex: number): number[] {
  const res = Array(rowIndex + 1);
  res[0] = 1;
  for (let i = 1; i <= rowIndex; i++) {
    res[i] = res[i - 1] * ((rowIndex - i + 1) / i);
  }
  return res;
}
```

The `getRow` function generates a specific row of Pascal's Triangle based on the given `rowIndex`. Each row in Pascal's Triangle represents the coefficients of the binomial expansion  $(a + b)^n$  for a specific value of  $n$ , where  $a$  and  $b$  are constants. The coefficients are also known as the binomial coefficients.

Here's how the function works:

1. It initializes an array `res` of length `rowIndex + 1` to store the coefficients of the specified row.
2. It sets the first element of the `res` array to `1`, as the first element of any row in Pascal's Triangle is always `1`.
3. It then iterates through each index `i` from `1` to `rowIndex`, calculating the coefficient at that index using the formula:

```
coefficient[i] = coefficient[i - 1] * ((rowIndex - i + 1) / i)
```

This formula is derived from the fact that each coefficient is obtained by multiplying the previous coefficient by the ratio of `(rowIndex - i + 1)` to `i`. This formula ensures that each coefficient is calculated efficiently and accurately.

4. After calculating all coefficients, the function returns the `res` array, which represents the specified row of Pascal's Triangle.

#### Techniques Used:

- **Array Initialization and Manipulation:** The function initializes an array and manipulates it to store the coefficients of the specified row.
- **Mathematical Logic:** The function uses a mathematical formula to calculate the coefficients efficiently based on the given `rowIndex`.

#### Summary:

The `getRow` function generates a specific row of Pascal's Triangle by calculating the coefficients using a formula that takes advantage of the properties of binomial coefficients. This allows for efficient computation of the coefficients for the desired row.

- Go back

#### Pascal triangle

```
class Solution {
    int[] pascalTriangleRow(int rowNo) {
        int pascalRow[] = new int[rowNo];
        pascalRow[0] = 1;
        rowNo--;
        for(int i = 1; i <= rowNo; i++) {
            int rowElement = pascalRow[i-1] * (rowNo - i + 1) / i;
            pascalRow[i] = rowElement;
        }
        return pascalRow;
    }
}
```

The given `Solution` class contains a method `pascalTriangleRow` that generates a specific row of Pascal's Triangle based on the given `rowNo`. Each row in Pascal's Triangle represents the coefficients of the binomial expansion  $(a + b)^n$  for a specific value of  $n$ , where  $a$  and  $b$  are constants. The coefficients are also known as the binomial coefficients.

Here's how the `pascalTriangleRow` method works:

1. It initializes an array `pascalRow` of length `rowNo` to store the coefficients of the specified row.
2. It sets the first element of the `pascalRow` array to 1, as the first element of any row in Pascal's Triangle is always 1.
3. It then iterates through each index `i` from 1 to `rowNo - 1`, calculating the coefficient at that index using the formula:

```
coefficient[i] = coefficient[i - 1] * (rowNo - i + 1) / i
```

This formula is derived from the fact that each coefficient is obtained by multiplying the previous coefficient by the ratio of  $(rowNo - i + 1)$  to  $i$ .

4. After calculating all coefficients, the method returns the `pascalRow` array, which represents the specified row of Pascal's Triangle.

#### Techniques Used:

- **Array Initialization and Manipulation:** The method initializes an array and manipulates it to store the coefficients of the specified row.
- **Mathematical Logic:** The method uses a mathematical formula to calculate the coefficients based on the given `rowNo`.

#### Summary:

The `pascalTriangleRow` method generates a specific row of Pascal's Triangle by calculating the coefficients using a formula that takes advantage of the properties of binomial coefficients. This allows for efficient computation of the coefficients for the desired row. The method can be used to retrieve any desired row of Pascal's Triangle by specifying the `rowNo` parameter.

- Go back

## Path Sum

### Path Sum

```
/**  
 * Definition for a binary tree node.  
 * class TreeNode {  
 *     val: number  
 *     left: TreeNode | null  
 *     right: TreeNode | null  
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {  
 *         this.val = (val===undefined ? 0 : val)  
 *     }  
 * }  
 */
```

```

        *         this.left = (left === undefined ? null : left)
        *         this.right = (right === undefined ? null : right)
        *
    }
*/

```

```

function hasPathSum(root: TreeNode | null, targetSum: number): boolean {

    if(root === null) {
        return false;
    }

    if(
        root.left === null &&
        root.right === null &&
        targetSum - root.val === 0
    ) {
        return true;
    }

    return hasPathSum(root.left, targetSum - root.val) || hasPathSum(root.right, targetSum - root.val)
};

```

The given `hasPathSum` function is designed to determine whether there exists a path from the root of a binary tree to a leaf node such that the sum of values along the path is equal to the given `targetSum`.

Here's how the function works:

1. If the `root` node is `null`, it means we have reached a null node in the tree. In this case, return `false` as there is no valid path to consider.
2. If the `root` node is a leaf node (both left and right children are `null`), check if the `targetSum` minus the value of the current `root` node is 0. If it is, return `true`, indicating that a valid path with the desired sum has been found.
3. Recursively call the `hasPathSum` function for both the left and right children of the current `root` node, with the `targetSum` reduced by the value of the current `root` node.
4. The function returns `true` if either the left or right subtree has a valid path with the remaining `targetSum`, otherwise it returns `false`.

### Techniques Used:

- **Binary Tree Traversal:** The function traverses the binary tree using a recursive approach to check for the existence of a path with the desired sum.
- **Recursive Approach:** The function utilizes recursion to explore each possible path in the binary tree and check whether the sum along that path matches the `targetSum`.

### Summary:

The `hasPathSum` function recursively explores the binary tree to find a path from the root to a leaf node such that the sum of values along the path is equal to the given `targetSum`. It employs a depth-first search approach to navigate the tree and determine whether a valid path exists.

## Perfect Number

# Perfect Number

```
function checkPerfectNumber(num: number): boolean {
    if (num <= 1) {
        return false;
    }

    let sum = 1;
    const sqrt = Math.floor(Math.sqrt(num));

    for (let i = 2; i <= sqrt; i++) {
        if (num % i === 0) {
            sum += i;
            if (i !== num / i) {
                sum += num / i;
            }
        }
    }

    return sum === num;
}
```

The `checkPerfectNumber` function determines whether a given number is a perfect number. A perfect number is a positive integer that is equal to the sum of its proper divisors (excluding itself).

Here's how the function works:

1. If the given number `num` is less than or equal to 1, it's not a perfect number. In this case, the function returns `false`.
2. Initialize a variable `sum` with the value 1. This is because all numbers are divisible by 1, so we include it as a proper divisor.
3. Calculate the square root of the given number `num` using `Math.sqrt(num)` and store it in the variable `sqrt`. We only need to check divisors up to the square root of the number.
4. Iterate from `i = 2` to `i <= sqrt`. For each value of `i`, check if `num` is divisible by `i` using the condition `num % i === 0`.
  - a. If `num` is divisible by `i`, add `i` to the `sum` to account for the proper divisor.
  - b. Additionally, if `i` is not equal to `num / i`, then add `num / i` to the `sum`. This ensures that both divisors are counted if they are distinct.
5. After the loop, compare the `sum` with the original `num`. If they are equal, then the given number is a perfect number, and the function returns `true`. Otherwise, it returns `false`.

### Techniques Used:

- **Mathematical Properties:** The function leverages the concept of perfect numbers, which are defined as numbers equal to the sum of their proper divisors.

- **Looping and Conditionals:** The function uses loops to iterate through possible divisors and conditional statements to handle proper divisor addition.

### Summary:

The `checkPerfectNumber` function checks whether a given number is a perfect number by calculating its proper divisors and comparing their sum to the original number. If the sum of proper divisors is equal to the original number, the function returns `true`, indicating that the number is a perfect number. Otherwise, it returns `false`.

- Go back

### Permutations

```
function permute(nums: number[]): number[][] {
  let output = [];
  dfs(nums, 0, output);
  return output;
}

function dfs(nums, index, output) {
  if (index == nums.length) {
    let l = [...nums];
    output.push(l);
  } else {
    for (let j = index; j < nums.length; j++) {
      let tmp = nums[index];
      nums[index] = nums[j];
      nums[j] = tmp;

      dfs(nums, index + 1, output);
      tmp = nums[index];
      nums[index] = nums[j];
      nums[j] = tmp;
    }
  }
}
```

The code defines a recursive algorithm to generate all possible permutations of a given array of numbers `nums`.

Here's how the code works:

1. The `permute` function takes an array `nums` as input and initializes an empty array `output` to store the generated permutations. It then calls the `dfs` function to start the recursive process.
2. The `dfs` function performs depth-first search to generate permutations. It takes three parameters: `nums` (the array of numbers), `index` (the current index being considered), and `output` (the array to store permutations).

3. The base case of the recursion is when `index` is equal to the length of the `nums` array. In this case, a copy of the `nums` array is added to the `output` array.
4. If the base case is not met, the function iterates over the remaining elements in the array starting from index `j`. It swaps the element at index `index` with the element at index `j` to generate a new permutation.
5. The function then recursively calls itself with `index + 1` to generate permutations for the remaining elements.
6. After the recursive call, the elements are swapped back to their original positions to backtrack and explore other possibilities.
7. Overall, the code uses a depth-first search approach to generate all permutations of the input array `nums`.

### Techniques Used:

- **Recursion:** The code uses recursion to systematically generate all possible permutations of the input array.
- **Backtracking:** The swapping of elements and then swapping them back after the recursive call is a backtracking technique to explore different possibilities.

### Summary:

The `permute` function generates all possible permutations of a given array of numbers using a depth-first search and backtracking approach. It starts with the `dfs` function, which explores different element positions and swaps them to create permutations. The base case is reached when all positions are filled, and a copy of the current permutation is added to the output array.

- Go back

### Plus minus

## Plus minus

```
function plusMinus(arr: number[]): void {
  // Write your code here
  const length = arr.length;
  let tempArr: number[] = [0, 0, 0];

  for (let i = 0; i < length; i++) {
    if (arr[i] > 0) {
      tempArr[0]++;
    } else if (arr[i] < 0) {
      tempArr[1]++;
    } else {
      tempArr[2]++;
    }
  }
}
```

```

        console.log((tempArr[0] / length).toFixed(6));
        console.log((tempArr[1] / length).toFixed(6));
        console.log((tempArr[2] / length).toFixed(6));
    }
}

```

The code defines a function `plusMinus` that calculates the fractions of positive, negative, and zero elements in an array. It then prints these fractions with a precision of 6 decimal places.

Here's how the code works:

1. The function `plusMinus` takes an array of numbers `arr` as input.
2. It initializes an array `tempArr` with three elements representing the counts of positive, negative, and zero elements, respectively. The initial values of all elements are set to 0.
3. A loop iterates through each element of the input array `arr`.
4. Inside the loop, the code checks whether the current element is positive, negative, or zero. Based on the comparison, the corresponding element in `tempArr` is incremented.
5. After the loop, the code calculates the fractions by dividing the count of positive, negative, and zero elements by the total length of the array.
6. The `toFixed(6)` method is used to format the fractions with a precision of 6 decimal places.
7. Finally, the code prints the calculated fractions using the `console.log` function.

### Techniques Used:

- **Looping:** The code uses a loop to iterate through each element of the input array.
- **Conditional Statements:** Conditional statements (if-else) are used to determine whether each element is positive, negative, or zero and update the corresponding count.

### Summary:

The `plusMinus` function calculates and prints the fractions of positive, negative, and zero elements in an array, each with a precision of 6 decimal places. It provides insights into the distribution of elements in the array.

- Go back

## Plus One

```

/**
 * @param {number[]} digits
 * @return {number[]}
 */
var plusOne = function (digits) {
    let carry = 1;
    let i = digits.length - 1;

```

```

while (i >= 0 && digits[i] === 9) {
    i--;
}

if (i == -1) {
    const result = new Array(digits.length + 1).fill(0);
    result[0] = 1;
    return result;
}

let result = new Array(digits.length).fill(0);
result[i] = digits[i] + 1;
for (let j = 0; j < i; j++) {
    result[j] = digits[j];
}
return result;
};

```

This JavaScript code defines a function `plusOne` that adds one to a given array of digits representing a non-negative integer. The code handles cases where the addition results in a carry. Here's how the code works:

1. The function `plusOne` takes an array of digits `digits` as input.
2. It initializes a variable `carry` with a value of 1, representing the initial carry for addition.
3. It initializes a variable `i` to the index of the last digit in the `digits` array.
4. A `while` loop is used to find the rightmost non-nine digit. The loop decrements `i` while the current digit is 9.
5. After exiting the loop, the code checks whether `i` is -1. If it is, all digits were nines, and a new result array is created with a length of `digits.length + 1`, filled with zeros. The first digit is set to 1 to handle the carry. The result array is then returned.
6. If `i` is not -1, it means there is a non-nine digit in the array. A new result array is created, and the digit at index `i` is incremented by 1 to handle the carry. Then, the digits before index `i` are copied from the original `digits` array to the result array.
7. Finally, the result array is returned, representing the integer obtained by adding one to the original integer represented by the input `digits`.

#### Techniques Used:

- **Looping:** The code uses a `while` loop to find the rightmost non-nine digit.
- **Conditional Statements:** Conditional statements are used to handle cases where all digits are nines and to determine the proper action based on the value of `i`.
- **Array Manipulation:** The code creates new arrays to store the result of addition and copies digits from the original array to the result array.

#### Summary:

The `plusOne` function adds one to a given non-negative integer represented by an array of digits. It handles carry and edge cases where all digits are nines, ensuring accurate addition of the one.

- Go back

## Power of Four

### Power of Four

```
function isPowerOfFour(n: number): boolean {
    if (n <= 0) {
        return false;
    }
    while (n > 3) {
        if (n % 4 != 0) {
            return false;
        }
        n = n / 4;
    }
    return n == 1;
}
```

The provided JavaScript code defines a function `isPowerOfFour` that checks if a given number is a power of 4. Here's how the code works:

1. The function `isPowerOfFour` takes an integer `n` as input.
2. It first checks if `n` is less than or equal to 0. If it is, the function immediately returns `false` since negative numbers and zero are not powers of 4.
3. It enters a `while` loop that continues as long as `n` is greater than 3 (indicating that `n` is at least 4).
4. Inside the loop, it checks if `n` is not divisible by 4. If this condition is met, it means that `n` is not a power of 4, so the function returns `false`.
5. If the `n` is divisible by 4, it divides `n` by 4 to reduce it and potentially continue checking.
6. Once the loop exits, the function checks if `n` is equal to 1. If it is, that means the original number was a power of 4, and the function returns `true`. Otherwise, it returns `false`.

#### Techniques Used:

- **Looping:** The code uses a `while` loop to repeatedly divide `n` by 4 until it becomes less than or equal to 3.
- **Conditional Statements:** The code uses conditional statements to check if `n` is not divisible by 4 and to determine if `n` is equal to 1.

#### Summary:

The `isPowerOfFour` function determines whether a given number is a power of 4 by repeatedly dividing the number by 4 and checking for divisibility and final equality to 1. If the conditions are met, the function returns `true`, indicating that the number is a power of 4; otherwise, it returns `false`.

- Go back

## Power of two

```
function isPowerOfTwo(n: number): boolean {
    if (n <= 0) {
        return false;
    }
    for (let i = 0; i < 32; i++) {
        if ((n & 1) == 1) {
            break;
        }
        n = n >>> 1;
    }
    n = n >>> 1;
    return n == 0;
}
```

This JavaScript code defines a function `isPowerOfTwo` that checks if a given number is a power of 2. Here's how the code works:

1. The function `isPowerOfTwo` takes an integer `n` as input.
2. It first checks if `n` is less than or equal to 0. If it is, the function immediately returns `false` since negative numbers and zero are not powers of 2.
3. The code enters a `for` loop that runs for 32 iterations. This loop is used to check if all bits of `n` are zero after repeatedly right-shifting `n` by one bit.
4. Inside the loop, it checks if the least significant bit of `n` is 1 using the bitwise AND operator (`&`). If the least significant bit is 1, it indicates that `n` is not a power of 2, so the loop is exited.
5. If the least significant bit of `n` is 0, the code right-shifts `n` by one bit (equivalent to dividing by 2).
6. After the loop, the code performs one more right-shift operation on `n`.
7. Finally, the code returns `true` if `n` is 0 after the right-shift operations, indicating that all bits of `n` were zero and it is a power of 2. Otherwise, it returns `false`.

### Techniques Used:

- **Looping:** The code uses a `for` loop to iterate through the bits of `n` and check if any of them are set.
- **Bitwise Operations:** The code uses bitwise AND (`&`) and bitwise right shift (`>>>`) operations to manipulate and check the bits of `n`.

### Summary:

The `isPowerOfTwo` function determines whether a given number is a power of 2 by repeatedly right-shifting the number's bits and checking if any of them are set. If all bits become zero after the operations, the function returns `true`, indicating that the number is a power of 2; otherwise, it returns `false`.

- Go back

## Primes upon to N

# Primes upon to N

```
class Solution {
    List<Integer> primesUptoN(int n) {
        boolean isPrime[] = new boolean[n + 1];
        ArrayList<Integer> output = new ArrayList<Integer>();

        for(int i = 2; i <= n; i++) {
            isPrime[i] = true;
        }

        isPrime[0] = false;
        isPrime[1] = false;
        for(int i = 2; i * i <= n; i++) {
            for(int j = i * i; j <= n; j += i) {
                if(isPrime[j] == true) {
                    isPrime[j] = false;
                }
            }
        }
        for(int i = 2; i <= n; i++) {
            if(isPrime[i] == true) {
                output.add(i);
            }
        }
        return output;
    }
}
```

This Java code defines a class `Solution` with a method `primesUptoN` that generates a list of prime numbers up to a given integer `n`. Here's how the code works:

1. The `primesUptoN` method takes an integer `n` as input and returns a list of prime numbers.
2. It creates a boolean array `isPrime` of size `n + 1`, where each element represents whether the corresponding index is a prime number or not. It initializes all elements to `true`.
3. The code sets `isPrime[0]` and `isPrime[1]` to `false` since 0 and 1 are not prime numbers.
4. The code uses a nested loop structure to mark the multiples of each prime number as non-prime. The outer loop runs from 2 to the square root of `n`, and the inner loop runs through multiples of the current prime number, starting from its square.
5. If `isPrime[i]` is `true`, it means `i` is a prime number. The code then marks its multiples (`j`) as non-prime by setting `isPrime[j]` to `false`.
6. After marking all non-prime numbers, the code creates an `ArrayList` named `output` to store the prime numbers.
7. It iterates through the `isPrime` array and adds the indices with `true` values (which represent prime numbers) to the `output` list.

- Finally, the `output` list containing prime numbers up to `n` is returned.

### Techniques Used:

- Sieve of Eratosthenes:** The code uses the Sieve of Eratosthenes algorithm to efficiently find prime numbers.

### Summary:

The `primesUptoN` method of the `Solution` class generates a list of prime numbers up to a given integer `n` using the Sieve of Eratosthenes algorithm. It uses a boolean array to mark non-prime numbers and then constructs an `ArrayList` containing the prime numbers found.

- Go back

### Promises sequence

## Promises sequence

```
const a = new Promise((resolve: any, reject: any) => {
  setTimeout(() => {
    console.log("Promise a");
    resolve();
  }, 5000);
});

const b = new Promise((resolve: any, reject: any) => {
  setTimeout(() => {
    console.log("Promise b");
    resolve();
  }, 4000);
});

const c = new Promise((resolve: any, reject: any) => {
  setTimeout(() => {
    console.log("Promise c");
    resolve();
  }, 1000);
});

let q = [a, b, c].reduce((acc: any, f: any) => {
  return acc.then(() => {
    return f;
  });
}, Promise.resolve());
```

This JavaScript code demonstrates the use of Promises to create three asynchronous tasks (`a`, `b`, and `c`). It then uses the `reduce` function to sequence these promises in a specific order, ensuring that each promise resolves before the next one starts. Here's a breakdown of how the code works:

1. Three Promises (`a`, `b`, and `c`) are created. Each Promise simulates an asynchronous operation using the `setTimeout` function. Each promise will print a message when resolved after a specific time delay.
2. The `reduce` function is used to sequentially chain the promises in the order [`a`, `b`, `c`].
3. The `reduce` function starts with an initial value of `Promise.resolve()`. This initial value is a resolved promise, acting as a starting point for the promise chaining.
4. For each promise `f` in the array [`a`, `b`, `c`], the accumulator promise (`acc`) is chained using the `then` method. This creates a sequence where each promise in the array waits for the previous promise to resolve before starting.
5. The `then` method takes a function as an argument, which returns the promise `f`. This ensures that the next promise in the sequence starts only after the previous promise has resolved.

In this specific code, the output will be as follows (assuming that each promise resolves without any errors):

```
Promise c
Promise b
Promise a
```

This is because `Promise c` has the shortest delay (1000 ms), followed by `Promise b` (4000 ms), and then `Promise a` (5000 ms).

#### **Techniques Used:**

- **Promises:** Promises are used to manage asynchronous operations and ensure that they are executed in a specific order.

#### **Summary:**

The code demonstrates how to use Promises and the `reduce` function to sequence asynchronous operations, ensuring that they are executed in a particular order based on the timing of their resolutions.

- Go back

#### **Queue using two stacks**

### **Queue using two stacks**

```
function processData(input) {
  let arr = input.split("\n");
  const stack = [];

  for (let i = 0; i < arr.length; i++) {
    let [cmd, value] = arr[i].split(" ");
    if (cmd == 1) {
      stack.push(value);
    } else if (cmd == 2) {
      stack.splice(0, 1);
    } else if (cmd == 3) {
      console.log(stack[0]);
    }
  }
}
```

This JavaScript code defines a function `processData` that processes input commands to perform stack operations. Here's a breakdown of how the code works:

1. The `processData` function takes a single argument `input`, which is expected to be a string containing newline-separated commands.
2. The `input` string is split into an array of strings `arr` using the newline character as the delimiter.
3. A stack (`stack`) is initialized to hold elements.
4. A loop iterates over each element in the `arr` array:
  - Each element in `arr` is split into two parts: `cmd` and `value`, using the space character as the delimiter.
  - If `cmd` is '1', it indicates a push operation. The `value` is pushed onto the stack.
  - If `cmd` is '2', it indicates a pop operation. The first element (top) of the stack is removed using the `splice` method.
  - If `cmd` is '3', it indicates a print operation. The value at the top of the stack (index 0) is printed to the console.
5. The code processes each command and performs the specified stack operation based on the `cmd` value.

### Techniques Used:

- **String Manipulation:** The `split` method is used to split the input string into an array of commands and values.
- **Stack:** The code uses an array (`stack`) to implement a stack data structure and performs push, pop, and print operations.

### Example:

If the `input` string is:

```
3
1 42
2
3
```

The output will be:

```
42
```

This is because the commands 1 42 push 42 onto the stack, the command 2 pops an element from the stack (which is now empty), and the command 3 prints the value 42 that was previously pushed onto the stack.

Please note that the provided code assumes valid input and does not include error handling for cases where commands are invalid or the stack is empty during a pop operation.

- Go back

## Range Addition II

# Range Addition II

```

function maxCount(m: number, n: number, ops: number[][]): number {
    if (ops.length === 0) {
        return m * n;
    }

    let minAi = m;
    let minBi = n;

    for (const op of ops) {
        minAi = Math.min(minAi, op[0]);
        minBi = Math.min(minBi, op[1]);
    }

    return minAi * minBi;
};

```

#### 1. Check for Empty Operations Array:

- Check if the array of operations (`ops`) is empty. If it is, return the product of `m` and `n` (the maximum count of elements when no operations are performed).

#### 2. Initialize Variables:

- Initialize two variables, `minAi` and `minBi`, to the initial dimensions `m` and `n`, respectively.

#### 3. Iterate Through Operations Array:

- Use a `for...of` loop to iterate through each operation in the array (`ops`).

#### 4. Update Minimum Values for Dimensions:

- For each operation, update the minimum values for dimensions (`minAi` and `minBi`) by taking the minimum of the current value and the values from the operation.

#### 5. Return Product of Minimum Dimensions:

- After iterating through all operations, return the product of the minimum dimensions (`minAi * minBi`), representing the maximum count of elements after all operations are performed.

## Range Sum Query

### Range Sum Query

```

class NumArray {
    public sums = [];
    constructor(nums: number[]) {
        for (var i = 0; i < nums.length; i++) {
            this.sums[i] = i === 0 ? nums[0] : (this.sums[i - 1] + nums[i]);
        }
    }

    sumRange(left: number, right: number): number {

```

```

        var sums = this.sums;
        return left === 0 ? sums[right] : (sums[right] - sums[left - 1]);
    }
}

/**
 * Your NumArray object will be instantiated and called as such:
 * var obj = new NumArray(nums)
 * var param_1 = obj.sumRange(left,right)
 */

```

This TypeScript code defines a class `NumArray` that implements a data structure for efficient range sum queries. Here's a breakdown of how the code works:

1. The `NumArray` class has two main methods: the constructor `constructor(nums: number[])` and `sumRange(left: number, right: number): number`.
2. The public `sums` array is used to store the cumulative sums of the input `nums`.
3. In the constructor, the cumulative sums are calculated and stored in the `sums` array. For each index `i`, the cumulative sum up to index `i` is computed as `this.sums[i] = i === 0 ? nums[0] : (this.sums[i - 1] + nums[i])`.
4. The `sumRange` method takes two parameters, `left` and `right`, representing the range for which the sum is to be calculated.
5. The cumulative sum up to index `right` is used to calculate the sum of the range `[left, right]`. If `left` is not 0, the cumulative sum up to index `left - 1` is subtracted from the cumulative sum up to index `right` to get the sum of the desired range.
6. The class is used by creating an instance of `NumArray` using the `nums` array, and then the `sumRange` method can be called to retrieve the sum of a specific range.

#### Example:

```

var nums = [-2, 0, 3, -5, 2, -1];
var obj = new NumArray(nums);

var sum1 = obj.sumRange(0, 2); // Returns 1 (sum of elements from index 0 to 2)
var sum2 = obj.sumRange(2, 5); // Returns -1 (sum of elements from index 2 to 5)
var sum3 = obj.sumRange(0, 5); // Returns -3 (sum of all elements from index 0 to 5)

```

In this example, the `NumArray` object is instantiated with the `nums` array, and then the `sumRange` method is called to compute the sum of specific ranges.

#### Ransom Note

## Ransom Note

```

function canConstruct(ransomNote: string, magazine: string): boolean {
  let dicArr = [...magazine];
  for (const c of ransomNote) {
    const index = dicArr.indexOf(c);
    if (index < 0) return false;
    dicArr.splice(index, 1);
  }
  return true;
};

```

### Explanation with Steps:

**Step 1:** Create an array `dicArr` by spreading the characters of the `magazine` string.

**Step 2:** Iterate through each character `c` in the `ransomNote` string.

**Step 3:** Inside the loop:

- Use the `indexOf` method to find the index of character `c` in the `dicArr` array.
- If the index is negative (character not found), return `false` immediately as the ransom note cannot be constructed.
- If the character is found, use the `splice` method to remove it from the `dicArr`, simulating its use in constructing the ransom note.

**Step 4:** After the loop completes, return `true`, indicating that the ransom note can be constructed.

### Techniques Used:

1. **Array Manipulation:** Utilizes array manipulation techniques to simulate constructing the ransom note using characters from the magazine.
2. **String Conversion to Array:** Converts the `magazine` string into an array of characters (`dicArr`) to facilitate manipulation and character tracking.
3. **Looping Through Characters:** Iterates through each character of the `ransomNote` using a loop.
4. **Character Lookup:** For each character in the `ransomNote`, uses the `indexOf` method to check if the character exists in the `dicArr` (magazine characters).
5. **Character Removal:** If the character exists in the `dicArr`, removes the character from the array using the `splice` method, simulating using that character to construct the ransom note.
6. **Early Return:** If `indexOf` returns a negative index (character not found), returns `false` immediately, optimizing the process.
7. **Return Result:** If all ransom note characters can be found and removed from the `dicArr`, returns `true`, indicating successful construction.

At the end of this explanation, you have a clear understanding of how the `canConstruct` function works and the techniques applied in the code.

## Regular Expression Matching

### Regular Expression Matching

```

function isMatch(s: string, p: string): boolean {
  const dp: boolean[][] = [];

  for (let i = 0; i <= s.length; i++) {
    dp[i] = [];
    for (let j = 0; j <= p.length; j++) {
      dp[i][j] = false;
    }
  }

  dp[0][0] = true;

  for (let i = 0; i <= s.length; i++) {
    for (let j = 1; j <= p.length; j++) {
      if (p[j - 1] === '*') {
        dp[i][j] = dp[i][j - 2] || (i > 0 && dp[i - 1][j] && (s[i - 1] === p[j - 2] || p[j - 2] === '.'));
      } else {
        dp[i][j] = i > 0 && dp[i - 1][j - 1] && (s[i - 1] === p[j - 1] || p[j - 1] === '.');
      }
    }
  }

  return dp[s.length][p.length];
};

```

### Explanation with Steps:

**Step 1:** Create a 2D array `dp` of booleans to store the matching status of substrings of `s` and `p`.

**Step 2:** Initialize the first cell `dp[0][0]` as `true` since an empty string matches an empty pattern.

**Step 3:** Iterate through each possible length of substring of `s` from 0 to its length.

**Step 4:** Inside the loop, iterate through each possible length of pattern substring from 1 to its length.

**Step 5:** If the current character in pattern `p[j - 1]` is a `'.'`, *handle the case where " "* can match zero preceding element or multiple preceding elements.

**Step 6:** If the current character in pattern `p[j - 1]` is not a `'*'`, check if the previous characters match and the current characters match (or the current character in pattern is `'.'`). Update `dp[i][j]` accordingly.

**Step 7:** After completing the loops, `dp[s.length][p.length]` contains the result indicating whether the entire string `s` matches the entire pattern `p`.

### Techniques Used:

1. **Dynamic Programming (DP):** Utilizes a 2D DP array to store the matching status of substrings of `s` and `p`.
2. **2D Array Initialization:** Initializes the 2D array `dp` and sets all values to `false` initially.
3. **Base Case Initialization:** Sets `dp[0][0]` to `true`, indicating that an empty string matches an empty pattern.
4. **Iterative Looping:** Iterates through each possible length of substring of `s` and pattern `p`.
5. **Pattern Matching Logic:** Uses pattern matching logic for different cases, including handling `'*'`, matching characters, and matching `'.'`.

6. **Updating DP Array:** Updates the `dp` array based on the matching conditions and previous values in the array.
7. **Final Result:** The value at `dp[s.length][p.length]` represents whether the entire string `s` matches the entire pattern `p`.

## Relative Ranks

### Relative Ranks

```
function findRelativeRanks(score: number[]): string[] {
  const sortedNums = [...score].sort((a, b) => b - a);
  const ranks: string[] = new Array(score.length);

  for (let i = 0; i < score.length; i++) {
    const rank = sortedNums.indexOf(score[i]) + 1;

    if (rank === 1) {
      ranks[i] = "Gold Medal";
    } else if (rank === 2) {
      ranks[i] = "Silver Medal";
    } else if (rank === 3) {
      ranks[i] = "Bronze Medal";
    } else {
      ranks[i] = rank.toString();
    }
  }

  return ranks;
};
```

1. **Sorting the Array:** The input `score` array is copied using the spread operator `[...score]` to create a new array `sortedNums`. This new array is sorted in descending order using the `sort()` method, which takes a comparison function `(a, b) => b - a`. This ensures that the largest scores come first in the sorted array.
2. **Initializing an Empty Array:** An empty array `ranks` is created with a length equal to the length of the input `score` array. This array will store the corresponding rank or medal label for each score.
3. **Iterating Through the Scores:** A `for` loop is used to iterate through each element in the `score` array. The loop variable `i` represents the index of the current score being processed.
4. **Calculating Rank:** The `indexOf()` method is used on the `sortedNums` array to find the index of the current score within the sorted array. Adding 1 to this index gives the rank of the current score.
5. **Assigning Medals or Rank:** Conditional statements (`if`, `else if`, `else`) are used to assign the appropriate medal or rank label to the `ranks` array based on the calculated rank value. If the rank is 1, "Gold Medal" is assigned; if the rank is 2, "Silver Medal" is assigned; if the rank is 3, "Bronze Medal" is assigned; otherwise, the rank is converted to a string using `toString()` and assigned.
6. **Returning the Result:** After iterating through all scores and assigning their corresponding ranks or medal labels, the `ranks` array is returned as the output of the function.

## Techniques Used:

1. Array Copy (`[...score]`): Copying an array using the spread operator.
2. Array Sorting (`sort()`): Sorting an array in descending order using a comparison function.
3. Array Initialization (`new Array()`): Creating an empty array with a specified length.
4. Iterative Looping (`for` loop): Iterating through elements of an array using a loop.
5. Array Indexing (`indexOf()`): Finding the index of an element within an array.
6. Conditional Statements (`if, else if, else`): Making decisions based on different conditions.
7. String Conversion (`toString()`): Converting a value to a string.
8. Returning a Value: Using the `return` statement to provide a function's output.

This function processes an array of scores, calculates ranks, and assigns medal labels or rank values based on the calculated ranks, returning an array with the corresponding information.

## Remove duplicate words

### Remove duplicate words

```
function removeDuplicateWords(s: string): string {
  const set = new Set();
  s.split(" ").forEach((str: string) => {
    set.add(str);
  });
  return [...set].join(" ");
}
```

1. **Initializing a Set:** A new empty `Set` is created to store unique words from the input string `s`.
2. **Splitting the Input String:** The input string `s` is split into an array of words using the `split(" ")` method. The space character " " is used as the delimiter.
3. **Iterating Through Words:** A `forEach` loop is used to iterate through each word in the array of words obtained from the split operation.
4. **Adding Words to the Set:** Within the loop, each word is added to the `set` using the `add()` method of the `Set`. This ensures that duplicate words are automatically eliminated due to the nature of a `Set`.
5. **Converting Set to Array and Joining:** After adding all unique words to the `set`, the `set` is converted back to an array using the spread operator `[...set]`. The array is then joined using the `join(" ")` method, with a space " " as the separator, to reconstruct the string with duplicate words removed.
6. **Returning the Result:** The final string with duplicate words removed is returned as the output of the function.

## Techniques Used:

1. Set (`Set`): Using a `Set` to store unique values and automatically remove duplicates.
2. String Splitting (`split()`): Splitting a string into an array of substrings based on a delimiter.
3. Iterative Looping (`forEach` loop): Iterating through elements of an array using a loop.
4. Set Methods (`add()`): Adding elements to a `Set`.
5. Array Conversion (`[...set]`): Converting a `Set` to an array using the spread operator.

6. Array Joining (`join()`): Joining elements of an array into a string using a separator.

This function takes an input string, removes duplicate words, and returns a new string with only the unique words present.

- Go back

### Remove duplicates

## Remove duplicates

```
class Solution {
    public int removeDuplicates(int[] nums) {
        if(nums.length == 0) {
            return 0;
        }
        int i = 0, j = 0;
        nums[j] = nums[i];
        for(i = 1; i < nums.length; i++) {
            if(nums[j] != nums[i]) {
                j = j + 1;
                nums[j] = nums[i];
            }
        }
        return j + 1;
    }
}
```

1. **Input Array Check:** The function `removeDuplicates` takes an array of integers `nums` as input. It first checks whether the length of the array is zero. If so, it returns 0, indicating that there are no elements in the array.
2. **Initialization:** Two integer pointers `i` and `j` are initialized to 0. The variable `j` is used to keep track of the index where unique elements will be placed.
3. **First Element Assignment:** The value of the first element (at index 0) of the input array `nums` is assigned to the first position (index 0) of the array.
4. **Loop Through Array:** A `for` loop is used to iterate through the array starting from index 1 (`i = 1`) to the end of the array.
5. **Comparing Elements:** Within the loop, the current element at index `i` is compared with the element at index `j`. If they are not equal, it means a new unique element is found.
6. **Storing Unique Elements:** The value of the current element at index `i` is stored in the next position (index `j + 1`) of the array, and `j` is incremented by 1 to mark the new position for the next unique element.
7. **Returning New Length:** After iterating through the array, the length of the array with duplicates removed is given by `j + 1`, where `j` represents the index of the last unique element. This value is returned.

### Techniques Used:

1. Array Length Check (`nums.length`): Checking the length of an array.
2. Two-Pointer Technique (`i` and `j`): Using two pointers to traverse and manipulate an array.
3. Element Assignment (`nums[j] = nums[i]`): Assigning the value of one array element to another.
4. Looping (`for` loop): Iterating through an array.
5. Element Comparison (`nums[j] != nums[i]`): Comparing two elements in an array.
6. Updating Pointer (`j = j + 1`): Incrementing a pointer to mark the position of the next unique element.
7. Returning Length: Calculating and returning the new length of the array with duplicates removed.

This function removes duplicates from an input array of integers and returns the new length of the array with only the unique elements remaining.

- Go back

### Remove element in place

## Remove element in place

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

1. **Input Array Modification:** The function `removeElement` takes an array of integers `nums` and an integer `val` as input. It aims to remove all instances of `val` from the array.
2. **Count Initialization:** A variable `count` is initialized to 0. This variable will be used to keep track of the valid (non-val) elements in the modified array.
3. **Loop Through Array:** A `for` loop is used to iterate through the input array `nums`.
4. **Valid Element Check:** For each element at index `i` in the array, it is checked whether the element is not equal to the given value `val`. If the element is valid (not equal to `val`), the next steps are performed.
5. **Updating Array:** The valid element at index `i` is assigned to the position `count` in the array, effectively overwriting the element at index `count` with the valid element. Then, `count` is incremented by 1 to mark the next position for a valid element.
6. **Returning New Length:** After iterating through the array, the value of `count` represents the new length of the array with the specified element (`val`) removed. This new length is returned.

### Techniques Used:

1. Looping (`for` loop): Iterating through an array.
2. Element Comparison (`nums[i] != val`): Comparing an array element with a given value.
3. Updating Array (`nums[count++] = nums[i]`): Assigning array elements to new positions while updating a counter.

4. Returning Length: Calculating and returning the new length of the array after removal.

This function removes all instances of a specified value from an input array of integers and returns the new length of the array with the specified value removed.

### Remove Linked List Elements

## Remove Linked List Elements

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     val: number
 *     next: ListNode | null
 *     constructor(val?: number, next?: ListNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.next = (next===undefined ? null : next)
 *     }
 * }
 */

function removeElements(head: ListNode | null, val: number): ListNode | null {
    while (head != null && head.val == val) {
        head = head.next;
    }

    if (head == null) {
        return null;
    }

    let temp = head;

    while (temp.next != null) {
        const nextNode = temp.next;
        if (nextNode.val == val) {
            temp.next = nextNode.next;
        } else {
            temp = temp.next;
        }
    }
    return head;
};
```

1. **Input Linked List Modification:** The function `removeElements` takes a singly-linked list `head` and an integer `val` as input. It aims to remove all nodes with the specified value `val` from the linked list.
2. **Initial Value Check:** In the beginning, a `while` loop is used to repeatedly check if the current `head` node is not `null` and if its `val` is equal to the given value `val`. If so, the current `head` node is moved to the next node (`head = head.next`) to remove nodes with the specified value from the beginning of the linked list.

3. **Empty List Check:** After the initial removal of nodes with the specified value from the beginning, a check is performed to see if the `head` node has become `null` (i.e., the list is empty). If so, `null` is returned, indicating an empty list.
4. **Traversal and Removal:** A temporary node `temp` is initialized to the `head` of the linked list. A `while` loop is used to traverse the linked list until the end. For each node, the following checks are performed:
  - If the value of the next node (`nextNode.val`) is equal to the specified value `val`, the `next` pointer of the current node (`temp`) is updated to skip the next node (i.e., `temp.next = nextNode.next`), effectively removing the node with the specified value from the list.
  - If the value of the next node is not equal to the specified value, the `temp` pointer is moved to the next node.
5. **Returning Modified List:** After completing the traversal and removal process, the modified linked list is returned, starting from the `head` node.

#### Techniques Used:

1. Looping (`while` loop): Iterating through a linked list and checking conditions.
2. Linked List Traversal and Modification: Iterating through a linked list and modifying its nodes by updating pointers.
3. Conditional Checks (`head != null`, `head.val == val`, `nextNode.val == val`): Checking conditions involving linked list nodes and values.
4. Returning Modified Linked List: Returning the modified linked list after removal of nodes with the specified value.

This function removes all nodes with a specified value from a given singly-linked list and returns the modified linked list.

#### Repeated Substring Pattern

### Repeated Substring Pattern

```
function repeatedSubstringPattern(s: string): boolean {
    for (let size=1;size<=s.length/2;size++) {
        if (s.length%size==0) {
            let curr=s.substring(0,size);
            let j=size;
            while (j<s.length && s.substring(j,j+size) === curr) {
                j+=size;
            }
            if (j==s.length) return true;
        }
    }
    return false;
};
```

1. **Input and Loop:** The function `repeatedSubstringPattern` takes a string `s` as input and aims to determine whether the string can be formed by repeatedly concatenating a substring. It uses a `for` loop to iterate through different sizes of substrings, starting from size 1 up to `s.length / 2`.

2. **Size-based Division Check:** For each substring size, it checks if the total length of the string is divisible by the current size (i.e., `s.length % size == 0`). If not, the substring of the current size cannot be repeated to form the entire string, so the loop proceeds to the next size.
3. **Substring Formation Check:** If the string length is divisible by the current size, it attempts to form the string by repeatedly concatenating the substring of the current size (`curr`). It uses a `while` loop with an index `j` starting from the size to traverse the string in chunks of the current size. Inside the loop, it compares the substring from index `j` to `j + size` with the `curr` substring. If they match, it increments `j` by the current size.
4. **Final Comparison:** If the `while` loop completes successfully (i.e., `j` reaches the end of the string), it means the string can be formed by repeatedly concatenating the current substring size. In this case, the function returns `true`.
5. **Returning False:** If none of the substring sizes can form the entire string, the function returns `false`.

### Techniques Used:

1. Looping (`for` loop and `while` loop): Iterating through different substring sizes and traversing the string to compare substrings.
2. Substring Comparison (`s.substring()`): Checking if substrings match by comparing them using the `==` operator.
3. Divisibility Check (`s.length % size == 0`): Checking if the string length is divisible by the current substring size.
4. String Concatenation (`curr = s.substring(0, size)`): Forming substrings of the current size to attempt to construct the entire string.

This function checks if a given string can be formed by repeatedly concatenating a substring of various sizes. If so, it returns `true`; otherwise, it returns `false`.

## Reshape the Matrix

### Reshape the Matrix

```
function matrixReshape(mat: number[][] , r: number, c: number): number[][] {
    const m = mat.length;
    const n = mat[0].length;

    if (m * n !== r * c) {
        return mat;
    }

    const reshapedMatrix: number[][] = [];
    let row: number[] = [];

    for (let i = 0; i < m; i++) {
        for (let j = 0; j < n; j++) {
            row.push(mat[i][j]);

            if (row.length === c) {
                reshapedMatrix.push(row);
                row = [];
            }
        }
    }
}
```

```

        }
    }

    return reshapedMatrix;
};


```

- Input and Matrix Dimensions:** The function `matrixReshape` takes a matrix `mat` and two integers `r` and `c` as input. It aims to reshape the given matrix into a new matrix with `r` rows and `c` columns. It first calculates the dimensions of the input matrix `mat` using `m` for the number of rows and `n` for the number of columns.
- Reshaping Validation:** If the total number of elements in the input matrix (`m * n`) is not equal to the total number of elements in the target reshaped matrix (`r * c`), then the reshaping is not possible. In this case, the function returns the original matrix `mat` as it cannot be reshaped.
- Reshaped Matrix Initialization:** If the reshaping is possible, an empty array `reshapedMatrix` is created to hold the new matrix. Additionally, an empty array `row` is initialized to temporarily store elements of each row during the reshaping process.
- Reshaping Loop:** The function uses nested `for` loops to traverse through each element of the input matrix `mat` row by row. For each element, it adds the element's value to the `row` array.
- Row Completion:** After adding an element to the `row` array, it checks if the length of `row` has reached the desired number of columns (`c`). If so, it appends the completed `row` to the `reshapedMatrix` and resets `row` to an empty array.
- Returning the Reshaped Matrix:** Once the reshaping process is complete, the function returns the `reshapedMatrix`, which now holds the desired reshaped matrix.

### Techniques Used:

- Nested Looping (`for` loop): Iterating through each element of the input matrix and constructing the reshaped matrix row by row.
- Conditional Statements (`if` statements): Checking whether the reshaping is possible and whether the `row` has reached the desired number of columns.
- Array Initialization and Manipulation: Creating arrays to store the reshaped matrix and the temporary row during reshaping.
- Arithmetic Operations (`m * n` and `r * c`): Calculating the total number of elements in the input and reshaped matrices to validate if reshaping is possible.

This function reshapes a given matrix into a new matrix with the specified number of rows and columns, preserving the original order of elements.

### Reverse array

## Reverse array

### Solution Steps

- Place the two pointers (let start and end) at the start and end of the array.
- Swap `a[start]` and `a[end]`

3. Increment start and decrement end with 1
4. If start reached to the value length/2 or start =/> end , then terminate otherwise repeat from step 2.

### Complexity Analysis

- Time Complexity: O(n)
- Space Complexity: O(1)

```
function reverseArray(a: number[]): number[] {
    let start = 0;
    let end = a.length - 1;
    while(start < end) {
        let temp = a[start];
        a[start] = a[end];
        a[end] = temp;
        start++;
        end--;
    }
    return a;
}
```

### Reverse bits

## Reverse bits

```
/**
 * @param {number} n - a positive integer
 * @return {number} - a positive integer
 */
var reverseBits = function(n) {
    let reversedArray = n.toString(2).split("").reverse()
    while(reversedArray.length <32){ reversedArray.push('0')}
    return parseInt(reversedArray.join(""),2)
};
```

1. **Function Input and Output:** The `reverseBits` function takes a positive integer `n` as input and returns another positive integer, which is the result of reversing the binary representation of `n`.
2. **Binary Conversion and Reversal:**
  - `n.toString(2)`: The input integer `n` is converted to its binary representation as a string.
  - `.split("")`: The binary string is split into an array of characters.
  - `.reverse()`: The array of binary digits is reversed to obtain the reversed binary representation.
3. **Padding to 32 Bits:** While the length of the reversed binary array is less than 32 (since JavaScript uses 32-bit representation for integers), the array is padded with '0' at the end to ensure it has a total of 32 bits.
4. **Binary to Integer Conversion:** The reversed binary array is then joined back into a string and converted to an integer using `parseInt` with base 2, which represents binary.

5. **Return Result:** The final integer, which is the result of reversing the binary representation of the input integer  $n$ , is returned.

#### Techniques Used:

1. String Manipulation: Converting the integer to binary string, splitting, and joining it for reversing and padding.
2. Looping and Array Manipulation: Adding '0' padding to the binary array until it reaches a length of 32.
3. Integer Conversion (`parseInt`): Converting the reversed binary string back to an integer using base 2.

This function essentially takes a positive integer, reverses its binary representation, and returns the resulting integer.

## Reverse String II

### Reverse String II

```
function reverseStr(s: string, k: number): string {
  const chars = s.split('');

  for (let i = 0; i < chars.length; i += 2 * k) {
    let start = i;
    let end = Math.min(i + k - 1, chars.length - 1);

    while (start < end) {
      [chars[start], chars[end]] = [chars[end], chars[start]];
      start++;
      end--;
    }
  }

  return chars.join('');
};
```

#### 1. Function Input and Output:

- **Input:** The `reverseStr` function takes a string  $s$  and an integer  $k$  as input.
- **Output:** It returns a modified string where every block of  $k$  characters is reversed, starting from the beginning.

#### 2. String to Character Array Conversion:

- The input string  $s$  is split into an array of individual characters using the `split('')` method.

#### 3. Reversing Blocks of Characters:

- The function iterates through the array of characters with a step size of  $2 * k$ .
- For each block, `start` is set as the current index  $i$ , and `end` is set as the minimum of  $(i + k - 1)$  and  $(\text{chars.length} - 1)$ .

#### 4. Two-Pointer Character Reversal:

- Inside each block, characters at `start` and `end` indices are swapped using array destructuring.
- The pointers `start` and `end` move towards each other until they meet, effectively reversing the characters within the block.

#### 5. Character Array to String Conversion:

- After reversing the characters, the modified character array is joined back into a string using the `join('')` method.

#### 6. Return Result:

- The modified string, where every block of `k` characters has been reversed, is returned as the output.

### Techniques Used:

#### 1. Looping and Array Manipulation:

- Iterating through the character array with a step size.
- Reversing characters within each block using two pointers.

#### 2. Swapping Variables:

- Using array destructuring to swap characters at `start` and `end` indices.

#### 3. String Manipulation:

- Converting the input string to an array of characters.
- Joining the character array back into a string to form the modified string.

This TypeScript function takes a string and an integer, reverses blocks of characters within the string, and returns the modified string.

### Reverse string

## Reverse string

```
function reverseString(s: string[]): void {
    let l = 0;
    let r = s.length - 1;

    while (l < r) {
        let temp = s[l];
        s[l++] = s[r];
        s[r--] = temp;
    }
};
```

#### 1. Function Input and Output:

- **Input:** The `reverseString` function takes an array of characters `s` as input.
- **Output:** It modifies the input array in-place to reverse its contents.

## 2. Two-Pointer Reversal Technique:

- The function initializes two pointers, `l` (left) and `r` (right), to the start and end of the input array, respectively.

## 3. Character Swapping:

- The function enters a `while` loop that continues as long as `l` is less than `r`.
- Inside the loop, the characters at `l` and `r` indices are swapped using a temporary variable `temp` and array assignment.
- After swapping, both `l` is incremented (using `l++`) and `r` is decremented (using `r--`) to move the pointers towards each other.

## 4. Loop Termination:

- The loop continues until `l` is no longer less than `r`. This ensures that the characters have been swapped until the middle of the array, effectively reversing the order of characters.

## 5. In-Place Modification:

- The input array `s` is modified in-place, and no new arrays are created.

## Techniques Used:

### 1. Two-Pointer Reversal:

- Reversing the contents of the array using two pointers (`l` and `r`) that move towards each other.

### 2. Array Element Swapping:

- Swapping characters at two indices within the array using a temporary variable.

### 3. In-Place Modification:

- Modifying the input array directly without creating a new array.

This TypeScript function reverses the order of characters in the input array in-place using a two-pointer reversal technique.

## Reverse Vowels of a String

### Reverse Vowels of a String

```
function reverseVowels(s: string): string {
    const arr = s.split("");
    let left = 0, right = arr.length;

    const vowels = ['A', 'E', 'I', 'O', 'U', 'a', 'e', 'i', 'o', 'u'];

    while (left < right) {
        if (vowels.indexOf(arr[left]) === -1) {
            left++;
            continue;
        }
        if (vowels.indexOf(arr[right]) === -1) {
            right--;
            continue;
        }
        [arr[left], arr[right]] = [arr[right], arr[left]];
        left++;
        right--;
    }
    return arr.join("");
}
```

```

    }

    if (vowels.indexOf(arr[right]) === -1) {
        right--;
        continue;
    }

    const temp = arr[left];
    arr[left] = arr[right];
    arr[right] = temp;

    left++;
    right--;
}

return arr.join("");
};

```

#### 1. Function Input and Output:

- **Input:** The `reverseVowels` function takes a string `s` as input.
- **Output:** It returns a new string with the vowels of the input string reversed while keeping the non-vowel characters in the same positions.

#### 2. Array Conversion:

- The function first converts the input string `s` into an array of characters `arr` using the `split` method.

#### 3. Two-Pointer Approach:

- The function initializes two pointers, `left` and `right`, to the start and end of the array, respectively.

#### 4. Vowel Detection:

- An array `vowels` is created containing uppercase and lowercase vowels.
- The function enters a `while` loop that continues as long as `left` is less than `right`.
- Inside the loop, the function checks if the character at the `left` pointer is not a vowel. If so, `left` is incremented.
- Similarly, if the character at the `right` pointer is not a vowel, `right` is decremented.

#### 5. Vowel Reversal:

- If both characters at the `left` and `right` pointers are vowels, their positions are swapped using a temporary variable `temp`.

#### 6. Pointer Movement:

- After swapping or skipping non-vowel characters, both `left` and `right` pointers are moved accordingly (`left++` and `right--`).

#### 7. Array to String Conversion:

- After the loop, the modified `arr` is converted back to a string using the `join` method, and the reversed-vowel string is returned.

## Techniques Used:

1. Two-Pointer Approach:
  - Reversing characters while maintaining relative positions using two pointers (`left` and `right`) that move towards each other.
2. Vowel Detection and Swapping:
  - Identifying vowels using an array of vowel characters and swapping vowel characters' positions within the array.
3. Array to String Conversion:
  - Converting an array of characters back to a string using the `join` method to obtain the final result.

This TypeScript function reverses the positions of vowels in a string while keeping the positions of non-vowel characters unchanged.

## Reverse Words in a String 3

### Reverse Words in a String 3

```
function reverseWords(s: string): string {  
    const words = s.split(" ");  
    const reversedWords = [];  
  
    for (let word of words) {  
        const reversedWord = word.split("").reverse().join("");  
        reversedWords.push(reversedWord);  
    }  
  
    return reversedWords.join(" ");  
}
```

#### 1. Function Input and Output:

- **Input:** The `reverseWords` function takes a string `s` as input, where `s` consists of words separated by spaces.
- **Output:** It returns a new string with each word in the input string reversed.

#### 2. Word Splitting:

- The input string `s` is split into an array of words using the `split` method with space (' ') as the delimiter. The result is stored in the `words` array.

#### 3. Word Reversal Loop:

- The function enters a loop that iterates through each word in the `words` array.
- For each word, it creates a new array `reversedWord` by splitting the word into an array of characters, reversing the characters, and then joining them back together using the `reverse` and `join` methods.

#### 4. Reversed Words Collection:

- The reversed word is pushed into the `reversedWords` array.

#### 5. Array to String Conversion:

- After reversing all words, the `reversedWords` array is joined into a string using space (' ') as the delimiter, resulting in a new string where each word is reversed but the order of words is maintained.

#### 6. Output:

- The final string with reversed words is returned as the output of the function.

### Techniques Used:

#### 1. String Splitting and Joining:

- Splitting the input string into an array of words using the `split` method with space as the delimiter, and joining arrays of characters back into words using the `join` method.

#### 2. Word Reversal:

- Reversing each word in the input string by splitting it into characters, reversing the characters, and then joining them back together.

#### 3. Array to String Conversion:

- Converting an array of reversed words back to a string using the `join` method to obtain the final result.

This TypeScript function reverses the characters within each word of a string while maintaining the order of the words.

- Go back

### Reverse words

## Reverse words

```
function reverseWords(str: string): string {
  return str
    .split(" ")
    .map((word: string) => {
      let w = word.split("").reverse();
      return w.join("");
    })
    .join(" ");
}
```

#### 1. Function Input and Output:

- **Input:** The `reverseWords` function takes a string `str` as input, where `str` consists of words separated by spaces.
- **Output:** It returns a new string with each word in the input string reversed.

## 2. Word Splitting and Mapping:

- The input string `str` is split into an array of words using the `split` method with space (' ') as the delimiter. This creates an array of words.
- The `map` method is then applied to each word in the array. For each word, the following steps are performed.

## 3. Word Reversal:

- Each word is further split into an array of characters using the `split` method. This array of characters is reversed using the `reverse` method.
- The reversed array of characters is then joined back together using the `join` method to form the reversed word.

## 4. Reversed Words Array:

- The `map` method returns an array of reversed words.

## 5. Array to String Conversion:

- The array of reversed words is joined into a string using space (' ') as the delimiter, resulting in a new string where each word is reversed but the order of words is maintained.

## 6. Output:

- The final string with reversed words is returned as the output of the function.

## Techniques Used:

### 1. String Splitting and Joining:

- Splitting the input string into an array of words using the `split` method with space as the delimiter, and joining arrays of characters back into words using the `join` method.

### 2. Mapping and Transforming:

- Using the `map` method to apply a transformation to each word in the array.

### 3. Word Reversal:

- Reversing each word in the input string by splitting it into characters, reversing the characters, and then joining them back together.

### 4. Array to String Conversion:

- Converting an array of reversed words back to a string using the `join` method to obtain the final result.

This TypeScript function achieves the same result as the previous implementation, reversing the characters within each word of a string while maintaining the order of the words.

- Go back

## Rising Temperature

# Rising Temperature

```
# Write your MySQL query statement below

SELECT w1.id
FROM Weather w1, Weather w2
WHERE w1.Temperature > w2.Temperature
AND datediff(w1.recordDate, w2.recordDate) = 1;
```

### 1. SELECT Statement:

- The query starts with the **SELECT** statement, indicating that we want to retrieve certain columns from the database.

### 2. Column Selection:

- The column selected in the query is **w1.id**. This suggests that we are interested in fetching the **id** values from the **Weather** table.

### 3. FROM Clause:

- The **FROM** clause specifies the tables involved in the query. In this case, it lists two instances of the **Weather** table: **w1** and **w2**. These instances are used as aliases to differentiate between two separate occurrences of the same table.

### 4. WHERE Clause:

- The **WHERE** clause specifies the conditions that the data must satisfy for the query to return results.
- **w1.Temperature > w2.Temperature**: This condition checks if the temperature of **w1** is greater than the temperature of **w2**.
- **datediff(w1.recordDate, w2.recordDate) = 1**: This condition checks if the difference in days between the **recordDate** of **w1** and **w2** is equal to 1. This effectively checks if the two records have consecutive dates.

### 5. Result:

- The query will retrieve the **id** values from the **Weather** table where the temperature of the first record (**w1**) is greater than the temperature of the second record (**w2**) and the dates of the two records are consecutive.

In summary, this SQL query retrieves the **id** values of records from the **Weather** table where the temperature of the first record is higher than the temperature of the next consecutive record. It essentially finds records with a temperature increase between consecutive days.

**Rotate image**

**Rotate image**

```

/**
Do not return anything, modify matrix in-place instead.
*/
function rotate(matrix: number[][]): void {
    if(matrix == null || matrix.length == 0 || matrix[0].length == 0) return;

    const rows = matrix.length;
    const cols = matrix[0].length;

    for(let first = 0, last = rows - 1; first < last; first++, last--) {
        const tmp = matrix[first];
        matrix[first] = matrix[last];
        matrix[last] = tmp;
    }

    for(let i = 0; i < rows; i++) {
        for(let j = i + 1; j < cols; j++) {
            const tmp = matrix[i][j];
            matrix[i][j] = matrix[j][i];
            matrix[j][i] = tmp;
        }
    }
};

```

#### Description:

##### 1. Input Validation:

- The function takes a 2D matrix `matrix` as input.
- It performs initial checks to ensure the matrix is not null, and that it has non-zero dimensions. If any of these conditions is met, the function returns immediately.

##### 2. Rows and Columns:

- The number of rows and columns in the matrix is stored in variables `rows` and `cols`, respectively.

##### 3. Matrix Vertical Reflection (Step 1):

- A loop iterates over the top half of the matrix's rows (up to `rows / 2`).
- In each iteration, the rows at index `first` and `last` are swapped using a temporary variable `tmp`.
- This step effectively reflects the matrix vertically along its horizontal center.

##### 4. Matrix Transposition (Step 2):

- A nested loop iterates over each element above the main diagonal (where `i < j`).
- In each iteration, the element at `matrix[i][j]` is swapped with the element at `matrix[j][i]` using a temporary variable `tmp`.
- This step effectively transposes the matrix along its main diagonal.

##### 5. In-Place Rotation Completed:

- After both steps, the matrix has been rotated 90 degrees clockwise in-place.

#### Techniques used:

1. In-place manipulation of the matrix elements.
2. Swapping elements using temporary variables.

### **Algorithm Complexity:**

- Time Complexity:  $O(N^2)$ , where  $N$  is the number of rows or columns in the matrix.
- Space Complexity:  $O(1)$ , as the operations are performed in-place without using extra space.

This code performs a two-step process: vertical reflection and matrix transposition, to achieve the desired 90-degree clockwise rotation of the given matrix in-place.

### **Same tree**

### **Same tree**

```
function isSameTree(p: TreeNode | null, q: TreeNode | null): boolean {
    if(p === null && q === null) return true;

    if (p === null || q === null || p.val !== q.val){
        return false;
    }

    return isSameTree(p.left,q.left) && isSameTree(p.right, q.right);
};
```

### **Description:**

#### **1. Base Case Check:**

- The function takes two binary tree nodes  $p$  and  $q$  as inputs.
- The first base case checks if both  $p$  and  $q$  are `null`, which indicates that the corresponding subtrees are identical. In this case, the function returns `true`.

#### **2. Comparison and Recursion:**

- If both  $p$  and  $q$  are not `null` and their values are not equal, it means that the subtrees are not identical, and the function returns `false`.
- If the values are equal, the function proceeds to check whether the left and right subtrees of  $p$  and  $q$  are identical by making recursive calls to `isSameTree`.

#### **3. Recursive Call:**

- The function recursively calls itself for the left and right subtrees of  $p$  and  $q$ .
- The function checks if the left subtrees of both trees are the same (`isSameTree(p.left, q.left)`) and if the right subtrees of both trees are the same (`isSameTree(p.right, q.right)`).

#### **4. Return Result:**

- The final result is the logical AND (`&&`) of the comparisons for both left and right subtrees. This means that for both subtrees to be the same, both left and right subtrees must be the same.

### **Techniques used:**

1. Recursive traversal of binary trees.
2. Comparison of tree nodes and their values.

### **Algorithm Complexity:**

- Time Complexity:  $O(N)$ , where  $N$  is the total number of nodes in the smaller of the two trees. Each node is visited exactly once.
- Space Complexity:  $O(H)$ , where  $H$  is the height of the smaller of the two trees. The space is used for the recursive call stack.

This code checks if two binary trees  $p$  and  $q$  are the same by comparing their structures and values through recursive traversal. If the trees have identical structures and node values, the function returns `true`, otherwise, it returns `false`.

### **Search insert position**

## **Search insert position**

```
function searchInsert(nums: number[], target: number): number {
  for (let i = 0; i < nums.length; i++) {
    if (nums[i] === target) {
      return i;
    } else if (nums[i] > target) {
      return i;
    }
  }
  return nums.length;
}
```

### **Description:**

#### **1. Linear Search:**

- The function `searchInsert` takes a sorted array `nums` and a target value `target` as inputs.
- It performs a linear search through the array by iterating over each element using a loop.

#### **2. Comparison and Insertion Position:**

- Inside the loop, the code checks two conditions:
  - If the current element `nums[i]` is equal to the target value `target`, it means the target is found at index `i`, and the function returns `i`.
  - If the current element `nums[i]` is greater than the target value `target`, it means the target should be inserted at index `i` to maintain the sorted order. The function returns `i`.

#### **3. Default Insertion Position:**

- If neither of the above conditions is met during the loop, it means the target value is greater than all elements in the array. Therefore, the insertion position for the target is at the end of the array, which is indicated by `nums.length`.

### **Techniques used:**

1. Linear search through an array.
2. Comparison of array elements and target value.

### **Algorithm Complexity:**

- Time Complexity:  $O(N)$ , where  $N$  is the length of the array `nums`. In the worst case, the entire array needs to be traversed.
- Space Complexity:  $O(1)$ , as no additional data structures are used, and the space used remains constant regardless of the input size.

This code efficiently finds the insertion position of a target value in a sorted array by utilizing the sorted property of the array and performing a linear search. It returns the index where the target value should be inserted while maintaining the sorted order.

- Go back

### **Simple text editor**

## **Simple text editor**

```
function processData(input) {
  const [numOps, ...queries] = input.split("\n");
  const undoStack = [];
  let str = "";

  for (let query of queries) {
    const [op, param] = query.split(" ");

    switch (op) {
      case "1":
        undoStack.push(str);
        str += param;
        break;
      case "2":
        undoStack.push(str);
        str = str.substring(0, str.length - Number(param));
        break;
      case "3":
        console.log(str[Number(param) - 1]);
        break;
      case "4":
        str = undoStack.pop();
        break;
    }
  }
}
```

### **Description:**

1. Input Parsing:

- The `processData` function takes a string `input` as input and processes it to perform various string manipulation operations.

## 2. Operations and Stack:

- The input is split into an array of lines, where the first line contains the number of operations (`numOps`), and the remaining lines contain the actual queries.
- A stack named `undoStack` is used to keep track of the history of string changes to support the "undo" operation.

## 3. String Manipulation Operations:

- The code iterates over each query using a loop and splits the query into an operation `op` and a parameter `param`.
- Depending on the operation, the following actions are performed:
  - "1": Appends the parameter `param` to the current string `str` and pushes the current `str` onto the `undoStack`.
  - "2": Removes the last `param` characters from the current string `str` and pushes the current `str` onto the `undoStack`.
  - "3": Outputs the character at the `(param - 1)` index of the current string `str`.
  - "4": Restores the previous string state by popping from the `undoStack` and assigning it to `str`.

### Techniques used:

1. Stack data structure for maintaining a history of string changes.
2. String manipulation and substring operations.

### Algorithm Complexity:

- Time Complexity:  $O(Q * M)$ , where  $Q$  is the number of queries and  $M$  is the average length of the string being manipulated.
- Space Complexity:  $O(Q * M)$ , due to the space used by the `undoStack` to store previous string states.

This code efficiently processes a series of string manipulation operations, including appending, removing, and retrieving characters, while providing support for undoing previous operations. It demonstrates the use of a stack to maintain history and perform string manipulations.

- Go back

### Single Number

## Single Number

```
/**
 * @param {number[]} nums
 * @return {number}
 */
const singleNumber = function (nums) {
  let ans = 0;
```

```

for (let i = 0; i < nums.length; i++) {
    ans ^= nums[i];
}

return ans;
};

```

#### Description:

##### 1. Input and Output:

- The `singleNumber` function takes an array of integers `nums` as input and returns the integer that appears only once in the array.

##### 2. Bitwise XOR Operation:

- The `ans` variable is initialized to 0. The XOR operation has a property that if the same number is XORed twice, it becomes 0. XORing any number with 0 gives the number itself.
- The loop iterates over each element in the `nums` array.
- For each element `nums[i]`, it performs the XOR operation with the current value of `ans`. This effectively cancels out all duplicate numbers and leaves only the single number.

##### 3. Result:

- The final value of `ans` will be the single number that appears only once in the array, as all other duplicate numbers will cancel out due to the XOR property.

#### Techniques used:

- Bitwise XOR operation for finding the unique number.

#### Algorithm Complexity:

- Time Complexity:  $O(n)$ , where  $n$  is the number of elements in the `nums` array. Each element is processed once.
- Space Complexity:  $O(1)$ , as only a constant amount of extra space is used (for the `ans` variable).

This code efficiently finds the single number in the array using the bitwise XOR operation, leveraging the XOR property to eliminate duplicate numbers and keep only the unique one.

- Go back

#### Special pythagorean triplet

#### Special pythagorean triplet

```

function specialPythagoreanTriplet(n: number): number | null {
    for (let a = 1; a < n / 3; a++) {
        for (let b = a + 1; b < n / 2; b++) {
            const c = n - a - b;
            if (a * a + b * b === c * c) {
                return a * b * c;
            }
        }
    }
    return null;
}

```

### Description:

#### 1. Outer Loop (Variable a):

- The outer loop iterates through possible values of  $a$  from 1 to  $(n / 3) - 1$ .
- The condition  $a < n / 3$  ensures that  $a$  is less than one-third of the total perimeter  $n$ .

#### 2. Inner Loop (Variable b):

- The inner loop iterates through possible values of  $b$  from  $a + 1$  to  $(n / 2) - 1$ .
- The condition  $b < n / 2$  ensures that  $b$  is less than half of the total perimeter  $n$ .
- This condition also guarantees that  $c = n - a - b$  is greater than 0.

#### 3. Calculate c and Check for Pythagorean Triple:

- Inside the inner loop, calculate  $c$  as the remaining value to fulfill the perimeter condition:  $c = n - a - b$ .
- Check if the current combination of  $a$ ,  $b$ , and  $c$  forms a Pythagorean triplet using the condition  $a^2 + b^2 === c^2$ .
- If a Pythagorean triplet is found, return the product of  $a$ ,  $b$ , and  $c$  ( $a * b * c$ ).

#### 4. Return null if No Triplet is Found:

- If the loops finish without finding any Pythagorean triplet, the function returns `null` to indicate that no such triplet exists for the given perimeter  $n$ .

**Sqrt(x)**

**Sqrt(x)**

```

/**
 * @param {number} x
 * @return {number}
 */
const mySqrt = function (x) {
    if (x < 2) return x;

    let start = 0;
    let end = x;
    while (start <= end) {

```

```

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

if (mid === x / mid) {
    return mid;
}

if (mid < x / mid) {
    start = mid + 1;
} else {
    end = mid - 1;
}
}

if (start > x / start) {
    return start - 1;
}
return start;
};

```

#### Description:

##### 1. Input and Output:

- The `mySqrt` function takes an integer `x` as input and returns the integer square root of `x`.

##### 2. Binary Search Algorithm:

- The binary search algorithm is used to find the square root of `x`.
- The `start` variable is initialized to 0, and the `end` variable is initialized to `x`.
- The algorithm continues to narrow down the range between `start` and `end` until `start` is greater than `end`.

##### 3. Binary Search Iteration:

- In each iteration, the middle value `mid` is calculated as the average of `start` and `end`.
- Since we are only interested in integer square roots, `mid` is converted to an integer using `Math.floor(mid)`.
- If `mid` is equal to `x / mid`, we found the exact square root and return `mid`.
- If `mid` is less than `x / mid`, we adjust `start` to `mid + 1` to search in the upper half of the range.
- If `mid` is greater than `x / mid`, we adjust `end` to `mid - 1` to search in the lower half of the range.

##### 4. Final Result:

- After the binary search completes, if `start` becomes greater than `x / start`, we return `start - 1` as the integer square root.
- If `start` is not greater than `x / start`, we return `start` as the integer square root.

#### Techniques used:

1. Binary search algorithm for finding the square root.
2. Integer conversion using `Math.floor()`.

#### Algorithm Complexity:

- Time Complexity:  $O(\log n)$ , where  $n$  is the input number  $x$ . The binary search algorithm reduces the search range by half in each iteration.
- Space Complexity:  $O(1)$ , as only a constant amount of extra space is used (for variables like `start`, `end`, and `mid`).

This code efficiently calculates the integer square root of a number using the binary search algorithm.

- Go back

### String to integer (atoi)

## String to integer (atoi)

```
/** 
 * @param {string} s
 * @return {number}
 */
const myAtoi = function (s) {
    if (!s) {
        return 0;
    }

    s = s.trim();
    const INT_MAX = 2147483647;
    const INT_MIN = -2147483648;
    let i = 0;

    const isNegative = s[0] === "-";
    const isPositive = s[0] === "+";

    if (isNegative) {
        i++;
    } else if (isPositive) {
        i++;
    }

    let number = 0;

    while (i < s.length && s[i] >= "0" && s[i] <= "9") {
        number = number * 10 + (s[i] - "0");
        i++;
    }

    number = isNegative ? -number : number;

    if (number < INT_MIN) {
        return INT_MIN;
    }
    if (number > INT_MAX) {
        return INT_MAX;
    }
}
```

```
    return number;
};
```

### Description:

#### 1. Input and Output:

- The `myAtoi` function takes a string `s` as input and returns an integer value that is extracted from the string.

#### 2. Trimmer and Constants:

- The input string `s` is trimmed to remove leading and trailing whitespace.
- Constants `INT_MAX` and `INT_MIN` are defined to represent the maximum and minimum integer values.

#### 3. Parsing Signs:

- The code checks if the first character of the trimmed string is `'-'` or `'+'` to determine if the number is negative or positive.

#### 4. Conversion:

- The code iterates through the string from the starting position `i`, converting the characters into a numeric value.
- The numeric value is built using the formula: `number = number * 10 + (s[i] - '0')`.

#### 5. Negative Number:

- If the number is determined to be negative (`isNegative` is true), the final numeric value is negated.

#### 6. Boundary Checks:

- The code checks whether the calculated numeric value falls within the range of 32-bit signed integers (`INT_MAX` and `INT_MIN`).
- If the number is outside this range, the function returns the corresponding boundary value.

#### 7. Returning the Result:

- If the numeric value is within the valid range, it is returned as the result.

### Techniques used:

- String manipulation and parsing.
- Handling integer overflow and underflow cases.
- String trimming.
- Conversion of characters to numeric values.

### Algorithm Complexity:

- Time Complexity:  $O(n)$ , where  $n$  is the length of the input string `s`. The code iterates through the string once.
- Space Complexity:  $O(1)$ , as only a constant amount of extra space is used (for variables like `i`, `number`, etc.).

This code efficiently converts a string to an integer, considering various sign and boundary conditions.

- Go back

## Student Attendance Record I

### Student Attendance Record I

```
function checkRecord(s: string): boolean {
    let absentCount = 0;
    let lateCount = 0;

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

        if (char === "A") {
            absentCount++;

            if (absentCount > 1) {
                return false;
            }
        }

        if (char === "L") {
            lateCount++;

            if (lateCount > 2 && s[i - 1] === "L" && s[i - 2] === "L") {
                return false;
            }
        } else {
            lateCount = 0;
        }
    }

    return true;
}
```

#### Description:

##### 1. Input and Output:

- The `checkRecord` function takes a string `s` representing a student's attendance record and returns a boolean value indicating whether the record meets certain conditions.

##### 2. Absent and Late Counts:

- `absentCount` keeps track of the number of "A" (absent) occurrences in the attendance record.
- `lateCount` keeps track of consecutive "L" (late) occurrences in the attendance record.

##### 3. Iteration:

- The code iterates through the string `s` character by character.

##### 4. Checking Absences:

- If the current character is "A" (absent), the `absentCount` is incremented.
- If `absentCount` exceeds 1 (more than one "A"), the function returns `false`.

### 5. Checking Lates:

- If the current character is "L" (late), the `lateCount` is incremented.
- If `lateCount` reaches 3 and the previous two characters were also "L", the function returns `false`.

### 6. Resetting Late Count:

- If the current character is not "L", the `lateCount` is reset to 0.

### 7. Returning the Result:

- If the attendance record meets all conditions (at most one "A" and no more than two consecutive "L"s), the function returns `true`.

### Techniques used:

1. String manipulation and parsing.
2. Tracking counts and conditions based on specific characters.
3. Iterating through a string and accessing characters.

### Algorithm Complexity:

- Time Complexity:  $O(n)$ , where  $n$  is the length of the input string `s`. The code iterates through the string once.
- Space Complexity:  $O(1)$ , as only a constant amount of extra space is used (for variables like `absentCount` and `lateCount`).

This code efficiently checks a student's attendance record to ensure it meets the specified conditions.

- Go back

### Submission Detail

### Submission Detail

```
function isSubsequence(s: string, t: string): boolean {
    let i = 0;
    let j = 0;
    while (i < s.length) {
        if (j === t.length) {
            return false;
        }
        if (s[i] === t[j]) {
            i++;
        }
        j++;
    }
    return true;
}
```

### Description:

## 1. Input and Output:

- The `isSubsequence` function takes two strings, `s` and `t`, and returns a boolean value indicating whether `s` is a subsequence of `t`.

## 2. Iteration:

- The code uses two pointers, `i` and `j`, to iterate through the strings `s` and `t`, respectively.

## 3. Comparison:

- While iterating, the code compares each character of `s` with the character of `t` at the same index (`i` and `j`).

## 4. Subsequence Check:

- If `s[i]` matches `t[j]`, the pointer `i` is advanced, indicating that the current character in `s` is found in `t`.
- If `j` reaches the end of string `t` and `i` hasn't reached the end of string `s`, it means there are characters left in `s` that haven't been found in `t`, and the function returns `false`.

## 5. Continuation:

- Regardless of whether there is a match, the pointer `j` is always advanced to move through string `t`.

## 6. Return Result:

- If all characters in string `s` are successfully matched in order in string `t`, the function returns `true`, indicating that `s` is a subsequence of `t`.

## Techniques used:

1. Two-pointer approach for iterating through two strings simultaneously.
2. Character comparison between two strings.
3. Loop control using conditions.

## Algorithm Complexity:

- Time Complexity:  $O(n)$ , where  $n$  is the length of string `t`. The code iterates through the characters of string `t` once.
- Space Complexity:  $O(1)$ , as only a constant amount of extra space is used for variables (`i` and `j`).

This code efficiently determines whether string `s` is a subsequence of string `t`.

- Go back

## Subtree of Another Tree

# Subtree of Another Tree

```

/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */

function isSubtree(s: TreeNode | null, t: TreeNode | null): boolean {
    if (s === null) {
        return false;
    }

    if (isSameTree(s, t)) {
        return true;
    }

    return isSubtree(s.left, t) || isSubtree(s.right, t);
}

function isSameTree(s: TreeNode | null, t: TreeNode | null): boolean {
    if (s === null && t === null) {
        return true;
    }

    if (s === null || t === null) {
        return false;
    }

    if (s.val !== t.val) {
        return false;
    }

    return isSameTree(s.left, t.left) && isSameTree(s.right, t.right);
}

```

### Description:

#### 1. Input and Output:

- The `isSubtree` function takes two binary tree nodes, `s` and `t`, and returns a boolean value indicating whether `t` is a subtree of `s`.

#### 2. Base Case:

- If `s` is null, there's no way `t` can be a subtree, so the function returns `false`.

#### 3. Check Same Tree:

- The code first checks if  $t$  is the same as  $s$ . This is done using the `isSameTree` helper function.
- If  $t$  is found to be the same as  $s$ , it means that  $t$  is a subtree of  $s$ , so the function returns `true`.

#### 4. Recursion:

- If  $t$  is not the same as  $s$ , the code recursively checks whether  $t$  is a subtree of  $s$ 's left child or right child.
- This is done by calling the `isSubtree` function recursively on  $s$ 's left and right subtrees.

#### 5. Helper Function `isSameTree`:

- This function checks if two binary trees,  $s$  and  $t$ , are the same.
- It handles the base cases when both  $s$  and  $t$  are null, and when one of them is null and the other is not.
- It compares the values of the nodes and recursively checks the left and right subtrees.

#### 6. Return Result:

- If none of the above conditions are met, the function returns `false`, indicating that  $t$  is not a subtree of  $s$ .

#### Techniques used:

1. Recursion to traverse and compare binary trees.
2. Base case handling for null nodes.
3. Node value comparison.

#### Algorithm Complexity:

- Time Complexity:  $O(m \cdot n)$ , where  $m$  is the number of nodes in tree  $s$  and  $n$  is the number of nodes in tree  $t$ . In the worst case, the `isSameTree` function can be called  $m \cdot n$  times.
- Space Complexity:  $O(\max(m, n))$ , where  $m$  and  $n$  are the heights of trees  $s$  and  $t$ , respectively. The maximum space used by the recursive call stack is determined by the height of the taller tree.

This code efficiently determines whether binary tree  $t$  is a subtree of binary tree  $s$ .

- Go back

#### Sum of Left Leaves

## Sum of Left Leaves

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *     }
}
```

```

*           this.right = (right === undefined ? null : right)
*     }
*   }
*/
function sumOfLeftLeaves(root: TreeNode | null): number {
  let result = 0;
  const dfs = (root) => {
    if (root === null) {
      return;
    }
    if (
      root.left !== null &&
      root.left.left === null &&
      root.left.right === null
    ) {
      result += root.left.val;
    }
    dfs(root.left);
    dfs(root.right);
  };
  dfs(root);
  return result;
}

```

### Description:

#### 1. Input and Output:

- The `sumOfLeftLeaves` function takes the root node of a binary tree and returns an integer, which is the sum of all left leaves' values.

#### 2. DFS Traversal:

- The code uses Depth-First Search (DFS) traversal to traverse the binary tree.
- The traversal is implemented using a recursive function named `dfs`.

#### 3. Base Case:

- The base case of the recursion is when `root` is null. In that case, the function simply returns without doing anything.

#### 4. Checking for Left Leaves:

- For each node, the code checks if it has a left child (`root.left !== null`) and if that left child is a leaf node (`root.left.left === null` and `root.left.right === null`).
- If both conditions are met, it means the left child is a left leaf. The value of the left leaf is added to the `result`.

#### 5. DFS Recursive Calls:

- The `dfs` function is recursively called on the left child and then on the right child of the current node.
- This ensures that all nodes of the binary tree are visited.

## 6. Return Result:

- After traversing the entire binary tree, the function returns the final **result**, which holds the sum of all left leaves' values.

## Techniques used:

1. Depth-First Search (DFS) traversal of a binary tree.
2. Recursive function for tree traversal.
3. Condition checking for left leaves.

## Algorithm Complexity:

- Time Complexity:  $O(n)$ , where  $n$  is the number of nodes in the binary tree. Each node is visited exactly once.
- Space Complexity:  $O(h)$ , where  $h$  is the height of the binary tree. The maximum space used by the recursive call stack is determined by the height of the tree.

This code efficiently calculates the sum of all left leaves' values in a binary tree.

- Go back

## Sum square difference

## Sum square difference

### Naive solution

```
function sumSquareDifference(n: number):number {
    let sumSquares: number = 0;
    let squareSum: number = 0;
    for(let i = 1; i <= n; i++) {
        sumSquares += Math.pow(i, 2);
        squareSum += i;
    }

    return Math.pow(squareSum, 2) - sumSquares;
}
```

## 1. Initialize Variables:

- Initialize two variables: **sumSquares** and **squareSum** to keep track of the sum of squares and the sum of the numbers, respectively. Both are initially set to 0.

## 2. Loop through Numbers:

- Use a **for** loop to iterate through numbers from 1 to  $n$ .
- In each iteration:
  - Add the square of the current number to the **sumSquares** variable using `Math.pow(i, 2)`.
  - Add the current number to the **squareSum** variable.

### 3. Calculate Square of the Sum:

- Calculate the square of the sum of numbers from 1 to  $n$  using `Math.pow(squareSum, 2)`.

### 4. Calculate the Difference:

- Calculate the difference between the square of the sum and the sum of squares.
- Subtract `sumSquares` from the square of the sum.
- Return the result.

### 5. Return Result:

- Return the calculated difference, which represents the difference between the sum of the squares and the square of the sum of the first  $n$  natural numbers.

## Optimized solution

```
function sumSquareDifference(n: number): number {  
    const sumOfSquares = (n * (n + 1) * (2 * n + 1)) / 6;  
    const sum = (n * (n + 1)) / 2;  
    const squareOfSum = Math.pow(sum, 2);  
    return squareOfSum - sumOfSquares;  
}
```

### 1. Calculate the Sum of Squares:

- The sum of squares of the first  $n$  natural numbers can be calculated using the formula
- Calculate `sumOfSquares` using the formula.

### 2. Calculate the Sum of Numbers:

- The sum of the first  $n$  natural numbers can be calculated using the formula
- Calculate `sum` using the formula.

### 3. Calculate the Square of the Sum:

- Calculate the square of the sum by squaring the value obtained in step 2 (`sum`) using `Math.pow(sum, 2)`.

### 4. Calculate the Difference:

- Calculate the difference between the square of the sum and the sum of squares.
- Subtract `sumOfSquares` from `squareOfSum`.

### 5. Return Result:

- Return the calculated difference, which represents the difference between the sum of the squares and the square of the sum of the first  $n$  natural numbers.

## Summary Ranges

## Summary Ranges

```

function summaryRanges(nums: number[]): string[] {
  const output: any[] = [];

  for (let i = 0; i < nums.length; ++i) {
    let begin = nums[i];
    while (i + 1 < nums.length && nums[i] == nums[i + 1] - 1) {
      ++i;
    }

    let end = nums[i];
    if (begin == end) {
      output.push(begin.toString());
    } else output.push(begin.toString() + "->" + end);
  }

  return output;
}

```

#### Description:

##### 1. Input and Output:

- The `summaryRanges` function takes an array of integers `nums` and returns an array of strings, where each string represents a summary range of consecutive numbers.

##### 2. Loop through Numbers:

- The code uses a loop to iterate through each number in the `nums` array.

##### 3. Finding Consecutive Ranges:

- Inside the loop, the code initializes a variable `begin` with the current number.
- Then, it enters a nested loop (`while` loop) to find consecutive numbers by checking if the next number is exactly one greater than the current number.
- If consecutive numbers are found, the loop continues to increment `i` and check the next number.

##### 4. Creating Summary Range:

- Once the end of the consecutive range is found (`end`), the code checks whether the range contains only one number (`begin == end`).
- If the range contains only one number, it's added as a single string to the `output` array.
- If the range contains multiple consecutive numbers, it's added as a range string (`begin -> end`) to the `output` array.

##### 5. Returning Output:

- After processing all numbers, the function returns the `output` array containing summary ranges.

#### Techniques used:

- Looping through an array.
- Nested `while` loop to find consecutive ranges.

#### Algorithm Complexity:

- Time Complexity:  $O(n)$ , where  $n$  is the number of elements in the input array. Each element is processed exactly once.
- Space Complexity:  $O(1)$ , excluding the space used for the output array since the output array size is directly proportional to the input array size.

This code efficiently generates summary ranges from a sorted array of numbers.

- Go back

## Summation of primes

### Summation of primes

```
function sumOfPrimes(n: number): number {

    const isPrime: boolean[] = new Array(n).fill(true);
    isPrime[0] = false;
    isPrime[1] = false;

    for (let i = 2; i <= Math.sqrt(n); i++) {
        if (isPrime[i]) {
            for (let j = i * i; j < n; j += i) {
                isPrime[j] = false;
            }
        }
    }

    let sum = 0;
    for (let i = 2; i < n; i++) {
        if (isPrime[i]) {
            sum += i;
        }
    }

    return sum;
}
```

#### Description:

##### 1. Initialize Prime Array:

- Create an array `isPrime` to track whether each number up to  $n - 1$  is prime.
- Set the values at index 0 and 1 to `false` since 0 and 1 are not prime numbers.

##### 2. Sieve of Eratosthenes:

- Use the Sieve of Eratosthenes algorithm to mark non-prime numbers in the `isPrime` array.
- Iterate through numbers starting from 2 up to the square root of  $n$ :
  - If the current number is marked as prime (`isPrime[i]` is `true`):

```
* Mark all multiples of the current number as not prime by setting isPrime[j] to false,  
where j ranges from i * i to n - 1 in increments of i.
```

### 3. Sum Prime Numbers:

- Initialize a variable `sum` to 0 to store the sum of prime numbers.
- Iterate through numbers from 2 to `n - 1`:
  - If the current number is prime (`isPrime[i]` is `true`), add it to the `sum`.

### 4. Return the Sum:

- Return the final sum of prime numbers.

The function uses the Sieve of Eratosthenes algorithm to efficiently find and mark prime numbers up to a given limit `n` and then calculates the sum of those prime numbers.

## Swap Nodes in Pairs

### Swap Nodes in Pairs

```
/**  
 * Definition for singly-linked list.  
 * class ListNode {  
 *     val: number  
 *     next: ListNode | null  
 *     constructor(val?: number, next?: ListNode | null) {  
 *         this.val = (val===undefined ? 0 : val)  
 *         this.next = (next===undefined ? null : next)  
 *     }  
 * }  
 */  
  
function swapPairs(head: ListNode | null): ListNode | null {  
    const dummy = new ListNode(0);  
  
    dummy.next = head;  
    let current = dummy;  
    while (current.next && current.next.next) {  
        let first = current.next;  
        let second = current.next.next;  
        first.next = second.next;  
        current.next = second;  
        current.next.next = first;  
        current = current.next.next;  
    }  
  
    return dummy.next;  
}
```

#### Description:

##### 1. Input and Output:

- The `swapPairs` function takes the head of a singly-linked list as input and returns the head of the modified list after swapping adjacent nodes.

## 2. Dummy Node:

- The code starts by creating a dummy node (`dummy`) and setting its `next` pointer to the input `head`.

## 3. Swapping Pairs:

- The `while` loop iterates through the list while there are at least two nodes available for swapping (`current.next` and `current.next.next`).
- Inside the loop, it creates references to the first (`first`) and second (`second`) nodes of the pair.
- It then updates the `next` pointers to perform the swap:
  - `first.next` points to the node after `second`.
  - `current.next` points to `second`.
  - `second.next` points to `first`.
- Finally, it moves `current` two steps ahead to the next pair's position.

## 4. Returning Modified List:

- After processing all pairs, the function returns the modified list's head, which is `dummy.next`.

### Techniques used:

- Linked list manipulation.
- Using a dummy node.
- Iterative traversal with pointer manipulation.

### Algorithm Complexity:

- Time Complexity:  $O(n)$ , where  $n$  is the number of nodes in the linked list. Each node is processed and swapped once.
- Space Complexity:  $O(1)$ , as only a constant amount of extra space is used, regardless of the size of the input linked list.

This code efficiently swaps adjacent nodes in a singly-linked list using a constant amount of extra space.

- Go back

### Symmetric difference

## Symmetric difference

```
export const symmetricDifference = (...args: any) => [
  ...new Set(
    args.reduce((arr1: any, arr2: any) => [
      ...arr1.filter((e: any) => !arr2.includes(e)),
      ...arr2.filter((e: any) => !arr1.includes(e)),
    ])
  ),
];
```

## Description:

### 1. Input and Output:

- The `symmetricDifference` function takes multiple arrays as input using the rest parameter `...args`.
- It returns an array containing the symmetric difference of all input arrays.

### 2. Reduce and Filter:

- The `args.reduce` function is used to iterate through each pair of arrays and compute their symmetric difference.
- The symmetric difference of two arrays is calculated by:
  - Filtering elements that are in the first array but not in the second array.
  - Filtering elements that are in the second array but not in the first array.
- The results of both filter operations are concatenated using the spread operator.

### 3. Using Set:

- The `new Set()` constructor is used to create a Set containing the concatenated symmetric differences.
- A Set is chosen to eliminate duplicates from the result.

### 4. Returning Result:

- The spread operator (...) is used to convert the Set back into an array.

## Techniques used:

1. Rest parameter to accept a variable number of arguments.
2. Using `reduce` to iterate and accumulate results.
3. Using `filter` to compute symmetric differences.
4. Using a `Set` to eliminate duplicates.
5. Using the spread operator to convert a Set back to an array.

## Example:

```
const result = symmetricDifference([1, 2, 3], [2, 3, 4], [3, 4, 5]);
console.log(result); // Output: [1, 5]
```

This code efficiently computes the symmetric difference of multiple arrays using JavaScript's array manipulation techniques.

- Go back

## Symmetric Tree

## Symmetric Tree

```

/**
 * Definition for a binary tree node.
 * class TreeNode {
 *     val: number
 *     left: TreeNode | null
 *     right: TreeNode | null
 *     constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null) {
 *         this.val = (val===undefined ? 0 : val)
 *         this.left = (left===undefined ? null : left)
 *         this.right = (right===undefined ? null : right)
 *     }
 * }
 */

function isSymmetric(root: TreeNode | null): boolean {
    return root == null || isMirror(root.left, root.right);
}

function isMirror(node1: TreeNode, node2: TreeNode): boolean {
    if (node1 === null && node2 === null) return true;
    if (node1 === null || node2 === null) return false;
    if (node1.val !== node2.val) return false;

    return isMirror(node1.left, node2.right) && isMirror(node1.right, node2.left);
}

```

### Description:

#### 1. Input and Output:

- The `isSymmetric` function takes the root of a binary tree as input.
- It returns `true` if the binary tree is symmetric (mirrored), otherwise `false`.

#### 2. Base Case and Recursion:

- The base case of the recursion is when `node1` and `node2` are both `null`, indicating that the nodes being compared are at the same positions in both subtrees.
- The base case also covers scenarios where one of the nodes is `null` (asymmetric) or when their values don't match (asymmetric).

#### 3. Checking Mirror Symmetry:

- The `isSymmetric` function first checks if the given binary tree is `null`. If it is, it's considered symmetric.
- The main recursion is done through the `isMirror` function, which takes two nodes as parameters.
- It checks if the current nodes (`node1` and `node2`) are symmetric and then recursively checks their subtrees' mirror symmetry.

#### 4. Techniques used:

- Recursive approach to check mirror symmetry.
- Handling base cases for `null` nodes and non-matching values.
- Utilizing short-circuiting to optimize the code.

### Example:

```
// Example of creating a symmetric binary tree
const root = new TreeNode(1);
root.left = new TreeNode(2, new TreeNode(3), new TreeNode(4));
root.right = new TreeNode(2, new TreeNode(4), new TreeNode(3));

const result = isSymmetric(root); // true
console.log(result); // Output: true
```

This code effectively checks if a given binary tree is symmetric by comparing the subtrees' mirror symmetry.

- Go back

### Teemo Attacking

## Teemo Attacking

```
function findPoisonedDuration(timeSeries: number[], duration: number): number {
    if (timeSeries.length === 0) {
        return 0;
    }

    let totalDuration = 0;
    let prevAttackEnd = timeSeries[0] + duration;

    for (let i = 1; i < timeSeries.length; i++) {
        const currentAttackEnd = timeSeries[i] + duration;
        totalDuration += Math.min(currentAttackEnd - prevAttackEnd, duration);
        prevAttackEnd = currentAttackEnd;
    }

    return totalDuration + duration;
}
```

### Description:

#### 1. Input and Output:

- The `findPoisonedDuration` function takes two parameters: an array `timeSeries` representing the times of attacks and an integer `duration` representing the duration of each attack.
- It returns the total poisoned duration caused by the attacks.

#### 2. Base Case and Initialization:

- If there are no attacks (the `timeSeries` array is empty), the function immediately returns 0, as there's no poison duration.

#### 3. Calculating Total Duration:

- The loop iterates through the `timeSeries` array starting from the second attack (`i = 1`).

- For each attack, it calculates the end time of the current attack by adding the attack time to the attack duration. This is stored in `currentAttackEnd`.

#### 4. Incrementing Total Duration:

- It then calculates the poisoned duration caused by the current attack using `Math.min(currentAttackEnd - prevAttackEnd, duration)`. This ensures that the duration doesn't exceed the actual attack duration.
- The calculated poisoned duration is added to the `totalDuration`.

#### 5. Updating Previous Attack End:

- The `prevAttackEnd` is updated with the `currentAttackEnd` so that it's ready for the next iteration.

#### 6. Final Calculation:

- After the loop, the code returns the `totalDuration` plus the `duration` of the last attack. This is done to account for the full duration of the last attack.

#### Techniques used:

- Looping through arrays and tracking indices.
- Basic mathematical calculations for calculating attack end times and durations.
- Efficient use of the `Math.min` function to calculate poisoned durations without exceeding attack durations.
- Handling base cases for empty input arrays.

#### Example:

```
const timeSeries = [1, 4, 7];
const duration = 3;
const result = findPoisonedDuration(timeSeries, duration);
console.log(result); // Output: 9
```

This code efficiently calculates the total poisoned duration caused by a series of attacks with given durations.

- Go back

#### Tenth Line

### Tenth Line

```
# Read from the file file.txt and output the tenth line to stdout.

head -n 10 file.txt | tail -n +10
```

The shell command `head -n 10 file.txt | tail -n +10` is used to display lines from a file in a specific range. Let's break down how this command works:

1. `head -n 10 file.txt`: This part of the command uses the `head` command to display the first 10 lines of the file named `file.txt`. The `-n` option specifies the number of lines to display.
2. `|`: This is known as a pipe operator. It takes the output from the command on its left and uses it as the input for the command on its right.
3. `tail -n +10`: This part of the command uses the `tail` command to display lines from the 10th line onwards. The `-n` option specifies the number of lines to display, and the `+10` means starting from the 10th line.

In summary, the combination of `head -n 10 file.txt` extracts the first 10 lines from the file, and then the output of this is piped to `tail -n +10`, which displays lines starting from the 10th line of the extracted lines. This effectively shows lines 10 to the end from the original file.

### Third Maximum Number

## Third Maximum Number

```
function thirdMax(nums: number[]): number {
  const output = [...new Set(nums)];
  output.sort((a, b) => b - a);
  return output[2] !== undefined ? output[2] : output[0];
}
```

- **Input:** An array of integers (`nums`).
- **Output:** The third maximum number in the array.

### 1. Create a Set for Uniqueness:

- A `Set` is used to eliminate duplicate numbers from the input array.
- This ensures that only unique values are considered further.

### 2. Sort Unique Values:

- The unique values in the `Set` are sorted in descending order using the `.sort()` function.
- Custom comparator `(a, b) => b - a` is used to sort in descending order.

### 3. Return Third Maximum or Maximum Value:

- Check if the third maximum value exists (`output[2]`):
  - If yes, return the third maximum value.
  - If not, return the maximum value (`output[0]`).

### Techniques Used:

- **Set:** Used to store unique numbers and remove duplicates.
- **Sorting:** Sorts unique numbers in descending order.
- **Ternary Operator:** Used to conditionally choose between the third maximum and maximum value.

### Algorithm:

1. Use a **Set** to remove duplicate numbers from **nums**.
2. Sort the unique values in descending order.
3. Return either the third maximum value (if exists) or the maximum value from the sorted array.

### Time Complexity:

- Removing duplicates using a **Set**:  $O(n)$
- Sorting unique values:  $O(n \log n)$
- Accessing third maximum value or maximum value:  $O(1)$

Overall time complexity:  $O(n \log n)$

### Space Complexity:

- Space for the **Set** to store unique numbers:  $O(n)$
- Space for the sorted array:  $O(n)$

Overall space complexity:  $O(n)$

- Go back

### Time conversion

## Time conversion

```
function timeConversion(s: string): string {
  const strArr = s.split(":");
  const modifier = strArr[2].slice(-2);
  let hours = strArr[0];
  if (modifier === "PM" && strArr[0] !== "12") {
    hours = +strArr[0] + 12 + "";
  }
  if (modifier === "AM" && strArr[0] === "12") {
    hours = "00";
  }
  return `${hours}:${strArr[1]}:${strArr[2].slice(0, -2)}`;
}
```

- **Input:** A time string in 12-hour format (**s**).
- **Output:** The corresponding time in 24-hour format.

### 1. Split the Time String:

- Split the input time string **s** using the colon (:) delimiter.
- Create an array **strArr** containing hours, minutes, and seconds.

### 2. Extract the Modifier (AM/PM):

- Get the last two characters of the seconds component to determine if it's "AM" or "PM".
- Store the modifier in the **modifier** variable.

### 3. Convert Hours for PM:

- If the modifier is "PM" and the hours are not already 12, add 12 to the hours.
- Convert the hours to a number using +, add 12, and convert it back to a string.

### 4. Convert Hours for AM:

- If the modifier is "AM" and the hours are 12, set the hours to "00".

### 5. Compose and Return the New Time String:

- Construct a new time string using the modified hours, original minutes, and seconds (excluding the modifier).

### Techniques Used:

- **String Splitting:** The input time string is split into an array.
- **Conditional Statements:** Used to handle AM and PM cases and modify the hours accordingly.
- **String Manipulation:** Creating the final time string by composing different parts.

### Algorithm:

1. Split the input time string into an array of hours, minutes, and seconds.
2. Determine the AM/PM modifier.
3. Convert hours according to the modifier.
4. If AM and hours are 12, set hours to "00".
5. Compose the new time string using the modified hours, original minutes, and seconds.

### Time Complexity:

- String splitting and manipulation: O(1)

### Space Complexity:

- Space for the strArr array: O(1)
- Other variables: O(1)

Overall space complexity: O(1)

- Go back

### Times function in JS

## Times function in JS

```
const times = (func, n) => {
  Array.from(Array(n)).forEach(() => {
    func();
  });
};

times(() => {
  randomFunction();
}, 3);
```

- **Input:**
  - `func`: A function to be executed `n` times.
  - `n`: The number of times the `func` function should be executed.
- **Output:** None (as it performs actions but doesn't return any value).

#### 1. Define the `times` Function:

- Create a function named `times` that takes two arguments: `func` and `n`.

#### 2. Execute the Function `n` Times:

- Create an array of length `n` using `Array.from(Array(n))`.
- Use the `forEach` method to iterate over the array and execute the `func` function for each iteration.
- The `randomFunction()` is called within each iteration, simulating the execution of a random function.

#### Techniques Used:

- **Higher-Order Function:** The `times` function takes another function as an argument and executes it a specified number of times.
- **Array Iteration:** Using the `forEach` method to iterate over an array.

#### Algorithm:

1. Define the `times` function that takes `func` and `n` as arguments.
2. Create an array of length `n` using `Array.from(Array(n))`.
3. Iterate over the array using the `forEach` method.
4. Inside each iteration, call the `func` function.

#### Time Complexity:

- The time complexity of the `times` function is  $O(n)$  due to the iteration.

#### Space Complexity:

- The space complexity is  $O(n)$  due to the array created using `Array.from(Array(n))`.
- Go back

#### Ugly Number

## Ugly Number

```
function isUgly(n: number): boolean {
  if (n == 1) return true;
  if (n <= 0) return false;
  if (n % 2 == 0) return isUgly(n / 2);
  else if (n % 3 == 0) return isUgly(n / 3);
  else if (n % 5 == 0) return isUgly(n / 5);
  else return false;
}
```

- **Input:**
    - **n**: An integer value to determine whether it's an ugly number or not.
  - **Output:**
    - A boolean value indicating whether **n** is an ugly number or not.
1. **Check for Base Cases:**
    - If **n** is equal to 1, return **true**, as 1 is considered an ugly number by definition.
    - If **n** is less than or equal to 0, return **false**, as negative or zero values are not considered ugly numbers.
  2. **Check Divisibility:**
    - Check if **n** is divisible by 2 using the modulo operator **%**. If it is, call the **isUgly** function recursively with **n / 2**.
    - If **n** is not divisible by 2, check if it's divisible by 3 or 5 in a similar manner and call the **isUgly** function recursively with **n / 3** or **n / 5** respectively.
  3. **Return Result:**
    - If **n** is divisible by 2, 3, or 5 and the recursive call for the corresponding divisor returns **true**, then return **true**, indicating that **n** is an ugly number.
    - If none of the conditions are met, return **false**, indicating that **n** is not an ugly number.

#### Techniques Used:

- **Recursion:** The function calls itself with smaller values of **n** to determine whether it's an ugly number or not.

#### Algorithm:

1. Check the base cases:
  - If **n** is 1, return **true**.
  - If **n** is less than or equal to 0, return **false**.
2. Check divisibility:
  - If **n** is divisible by 2, recursively call **isUgly(n / 2)**.
  - If **n** is not divisible by 2, check divisibility by 3 and 5 in the same way.
3. Return the result:
  - If any of the divisibility conditions hold and the corresponding recursive call returns **true**, return **true**.
  - If none of the conditions hold, return **false**.

#### Time Complexity:

- The time complexity depends on the prime factors of **n**, and in the worst case, the function makes multiple recursive calls. The time complexity is proportional to the number of prime factors of **n**.

#### Space Complexity:

- The space complexity is determined by the depth of the recursion stack. In the worst case, the recursion depth could be proportional to **n**. Therefore, the space complexity is **O(n)**.
- Go back

## Valid Anagram

### Valid Anagram

```
function isAnagram(s: string, t: string): boolean {
    if (s.length !== t.length) return false;

    let hashTable = {};

    for (let i = 0; i < s.length; i++) {
        if (!hashTable[s[i]]) hashTable[s[i]] = 0;
        if (!hashTable[t[i]]) hashTable[t[i]] = 0;
        hashTable[s[i]]++;
        hashTable[t[i]]--;
    }

    for (let key in hashTable) {
        if (hashTable[key] != 0) return false;
    }

    return true;
}
```

- **Input:**

- **s**: A string representing the first input string.
- **t**: A string representing the second input string.

- **Output:**

- A boolean value indicating whether **s** and **t** are anagrams of each other.

1. **Check Lengths:**

- If the lengths of strings **s** and **t** are not equal, return **false**. Anagrams must have the same length.

2. **Create a Hash Table:**

- Initialize an empty **hashTable** object to store the frequency of characters in the strings.

3. **Count Characters in Both Strings:**

- Iterate through the characters of both strings using the same index **i**.
- For each character **s[i]** and **t[i]**, increment the corresponding count in the **hashTable**.
- If the character doesn't exist in the **hashTable**, initialize its count to 1.

4. **Check Character Frequencies:**

- After counting characters for both strings, iterate through the keys in the **hashTable**.
- If any character's count is not equal to 0, return **false**, as it means the frequencies of characters in **s** and **t** don't match.

5. **Return Result:**

- If all characters' frequencies match and there is no character with a non-zero count in the `hashTable`, return `true`, indicating that `s` and `t` are anagrams of each other.

### Techniques Used:

- **Hash Table:** A hash table is used to store the frequency of characters in the strings.

### Algorithm:

1. Check if the lengths of strings `s` and `t` are equal. If not, return `false`.
2. Create an empty `hashTable` object.
3. Count the frequency of characters in both strings:
  - Iterate through the characters of both strings using the same index `i`.
  - If the character doesn't exist in the `hashTable`, initialize its count to 1.
  - If the character exists, increment its count.
4. Check character frequencies:
  - Iterate through the keys in the `hashTable`.
  - If any character's count is not equal to 0, return `false`.
5. If all characters' frequencies match, return `true`.

### Time Complexity:

- The time complexity is  $O(n)$ , where  $n$  is the length of the strings. The code iterates through both strings once and iterates through the keys in the hash table once.

### Space Complexity:

- The space complexity is  $O(1)$  since the hash table stores characters, and the size of the hash table is bounded by the number of distinct characters in the input strings.

### Valid parentheses

## Valid parentheses

```
function isValid(s: string): boolean {
  const temp = [];

  for (let i = 0; i < s.length; i++) {
    if (s[i] == "(" || s[i] == "{" || s[i] == "[") {
      temp.push(s[i]);
    } else if (s[i] == ")" && temp.length && temp[temp.length - 1] === "(") {
      temp.pop();
    } else if (s[i] == "]" && temp.length && temp[temp.length - 1] === "[") {
      temp.pop();
    } else if (s[i] == "}" && temp.length && temp[temp.length - 1] === "{") {
```

```

        temp.pop();
    } else {
        return false;
    }
}

return temp.length === 0;
}

```

- **Input:**

- **s:** A string containing parentheses and braces.

- **Output:**

- A boolean value indicating whether the parentheses and braces in the string are balanced.

1. **Initialization:**

- Initialize an empty array **temp** to act as a stack to track opening brackets.

2. **Iterate Through the String:**

- Iterate through each character **s[i]** in the given string **s**.

3. **Handle Opening Brackets:**

- If the current character is an opening bracket ('(', '{', or '['), push it onto the **temp** stack.

4. **Handle Closing Brackets:**

- If the current character is a closing bracket (')', '}', or ']'):
  - Check if the **temp** stack is not empty and the top of the stack matches the corresponding opening bracket for the current closing bracket.
  - If the condition is met, pop the top element from the **temp** stack.
  - If the condition is not met, return **false** as the brackets are not balanced.

5. **Handle Other Characters:**

- If the current character is not an opening or closing bracket, return **false** as the string contains invalid characters.

6. **Check Stack:**

- After processing all characters, check if the **temp** stack is empty.
- If the stack is empty, return **true** (balanced brackets).
- If the stack is not empty, return **false** (unbalanced brackets).

### **Techniques Used:**

- **Stack:** A stack data structure is used to keep track of opening brackets.

### **Algorithm:**

1. Initialize an empty stack `temp`.
2. Iterate through each character `s[i]` in the input string `s`.
3. If the current character is an opening bracket ('(', '{', or '['), push it onto the stack.
4. If the current character is a closing bracket (')', '}', or ']'), check if the stack is not empty and the top element of the stack matches the corresponding opening bracket. If yes, pop the top element from the stack. If not, return `false`.
5. If the current character is neither an opening nor a closing bracket, return `false`.
6. After processing all characters, check if the stack is empty. If yes, return `true` (balanced brackets). If not, return `false` (unbalanced brackets).

#### Time Complexity:

- The time complexity is  $O(n)$ , where  $n$  is the length of the input string. The code iterates through the string once.

#### Space Complexity:

- The space complexity is  $O(n)$  in the worst case, where  $n$  is the length of the input string. The stack can store all opening brackets in the worst case.

#### Valid Perfect Square

## Valid Perfect Square

```
function isPerfectSquare(num: number): boolean {
  let i = 1;
  while (num > 0) {
    num -= i;
    i += 2;
  }
  return num == 0;
}
```

- **Input:**

- `num`: A non-negative integer.

- **Output:**

- A boolean value indicating whether the given number is a perfect square.

1. **Initialization:**

- Initialize a variable `i` with the value 1. This variable will be used to generate consecutive odd numbers.

2. **Loop to Subtract Odd Numbers:**

- Enter a loop while `num` is greater than 0.
- In each iteration, subtract the current value of `i` from `num`.
- Increment `i` by 2 in each iteration to get the next odd number (1, 3, 5, ...).

### 3. Check for Perfect Square:

- After the loop, check if `num` is equal to 0.
- If `num` is 0, return `true` indicating that the input number is a perfect square.
- If `num` is not 0, return `false` indicating that the input number is not a perfect square.

### Techniques Used:

- **Mathematical Pattern:** The code uses a mathematical pattern based on odd numbers to determine whether the input number is a perfect square.

### Algorithm:

1. Initialize `i` to 1.
2. Enter a loop while `num` is greater than 0.
  - Subtract the current value of `i` from `num`.
  - Increment `i` by 2 to get the next odd number.
3. After the loop, check if `num` is 0.
  - If `num` is 0, return `true` as the input number is a perfect square.
  - If `num` is not 0, return `false` as the input number is not a perfect square.

### Time Complexity:

- The time complexity of this code is  $O(\sqrt{n})$ , where  $n$  is the input number. This is because the loop runs until `num` becomes 0, and in the worst case, `num` can be reduced to 0 by subtracting at most  $\sqrt{n}$  odd numbers.

### Space Complexity:

- The space complexity is  $O(1)$  as only a constant amount of extra space is used for the variable `i`.

### Valid Phone Numbers

## Valid Phone Numbers

```
# Read from the file file.txt and output all valid phone numbers to stdout.

grep -E "^(\\([0-9]{3}\\) | [0-9]{3}\\-)[0-9]{3}\\-[0-9]{4}$" file.txt
```

The command uses the `grep` utility with the `-E` flag to enable extended regular expressions. It searches for lines in the file "file.txt" that match a specific pattern, which corresponds to phone numbers in a specific format.

The regular expression `^(\\([0-9]{3}\\) | [0-9]{3}\\-)[0-9]{3}\\-[0-9]{4}$` is used to match phone numbers in the format "###-###-####" or "(###) ####". Here's the breakdown of the regular expression:

- `^`: Anchors the start of the line.
- `(`: Matches an opening parenthesis.
- `[0-9]{3}`: Matches exactly three digits (0-9).
- `)`: Matches a closing parenthesis.
- `|`: Alternation operator, matches either the pattern before or after it.
- `[0-9]{3}-`: Matches three digits followed by a hyphen.
- `[0-9]{3}-`: Matches another three digits followed by a hyphen.
- `[0-9]{4}`: Matches four digits.
- `$`: Anchors the end of the line.

The command searches each line in "file.txt" and displays only the lines that match the specified phone number pattern. This pattern is commonly used to match valid US phone numbers in different formats.

- Go back

### Validate pin

## Validate pin

```
const validatePin = (pin: string): boolean => {
  if (pin.length !== 4 && pin.length !== 6) return false;
  return /^[0-9]{4}$|^[0-9]{6}$/.test(pin);
};
```

- **Input:**
  - `pin`: A string representing a PIN code.
- **Output:**
  - A boolean value indicating whether the given PIN is valid.

### 1. Length Validation:

- Check if the length of the input `pin` is either 4 or 6.
- If the length is not 4 or 6, return `false`, indicating that the PIN is not valid.

### 2. Regular Expression Validation:

- Use a regular expression `/^[0-9]{4}$|^[0-9]{6}$/.gm` to validate the PIN:
  - `^[0-9]{4}$`: Matches exactly 4 digits.
  - `|`: OR operator.
  - `^[0-9]{6}$`: Matches exactly 6 digits.
  - `/gm`: Flags for global and multiline matching.

### 3. Test the Regular Expression:

- Use the `.test(pin)` method of the regular expression object to test if the `pin` matches the pattern.
- Return the result of the test (either `true` if the PIN matches the pattern, or `false` if it doesn't).

### Techniques Used:

- **Regular Expressions:** The code uses regular expressions to validate the PIN format.

#### Algorithm:

1. Check if the length of the input `pin` is either 4 or 6.
  - If not, return `false`.
2. Use the regular expression `/^ [0-9]{4} $| ^ [0-9]{6} $/gm` to validate the PIN format.
3. Test the regular expression using `.test(pin)` and return the result.

#### Time Complexity:

- The time complexity of this code is  $O(1)$  as both length validation and regular expression matching are constant time operations.

#### Space Complexity:

- The space complexity is  $O(1)$  as the code uses a constant amount of extra space for the regular expression and variables.

### Word Pattern

## Word Pattern

```
function wordPattern(pattern: string, s: string): boolean {
  const strArr = s.split(" ");

  if (strArr.length != pattern.length) {
    return false;
  }

  let hash = {};

  for (let i = 0; i < strArr.length; i++) {
    if (hash[pattern[i]]) {
      if (hash[pattern[i]] !== strArr[i]) {
        return false;
      }
    } else {
      if (Object.values(hash).indexOf(strArr[i]) !== -1) {
        return false;
      } else {
        hash[pattern[i]] = strArr[i];
      }
    }
  }

  return true;
}
```

- **Inputs:**
    - **pattern**: A string representing the pattern.
    - **s**: A string representing the input string.
  - **Output:**
    - A boolean value indicating whether the given **pattern** matches the structure of the words in string **s**.
- 1. Split Input String:**
    - Split the string **s** into an array of words using the space character (' ') as the delimiter.
    - Store this array in the variable **strArr**.
  - 2. Length Check:**
    - Check if the length of **strArr** is equal to the length of the **pattern**.
    - If the lengths are not equal, return **false**, indicating that the pattern cannot match the words.
  - 3. Pattern Mapping:**
    - Create an empty object **hash** to store the mapping of characters in the **pattern** to words in **strArr**.
    - Loop through each index **i** from 0 to the length of **strArr**:
      - If **pattern[i]** already exists in **hash**:
        - \* If the stored value does not match **strArr[i]**, return **false**.
      - If **pattern[i]** does not exist in **hash**:
        - \* Check if the value **strArr[i]** is already mapped to any character in the **hash** using `Object.values(hash).indexOf(strArr[i])`.
        - \* If it is, return **false**, as one word cannot be mapped to multiple characters.
        - \* If not, create a mapping between **pattern[i]** and **strArr[i]** in the **hash**.
  - 4. Pattern Mapping Successful:**
    - If the loop completes without returning **false**, return **true**, indicating that the pattern successfully matches the words in the input string.

#### Techniques Used:

- **Hash Table (Object):** The code uses an object (**hash**) to store the mapping between characters in the pattern and words in the input string.
- **Array Manipulation:** The code splits the input string into an array of words using the **split()** method.

#### Algorithm:

1. Split the input string into an array of words (**strArr**).
2. Check if the lengths of **strArr** and **pattern** are equal. If not, return **false**.
3. Loop through each index **i** from 0 to the length of **strArr**:
  - If **pattern[i]** exists in **hash**:
    - If the stored value does not match **strArr[i]**, return **false**.

- If `pattern[i]` does not exist in `hash`:
    - Check if `strArr[i]` is already mapped to any character in `hash`. If yes, return `false`.
    - Otherwise, create a mapping in `hash`.
4. If the loop completes without returning `false`, return `true`.

#### Time Complexity:

- The time complexity of this code is  $O(N)$ , where  $N$  is the length of the input string `s`.

#### Space Complexity:

- The space complexity is  $O(K)$ , where  $K$  is the number of unique characters in the pattern. In the worst case, all characters in the pattern are unique, so the space complexity is  $O(N)$ , where  $N$  is the length of the pattern.

#### Word search

## Word search

```

function exist(board: string[][] , word: string): boolean {
    const row = board.length;
    const col = board[0].length;
    let index = 0;
    for (let i = 0; i < row; i++) {
        for (let j = 0; j < col; j++) {
            if (backtrack(board, word, i, j, index, row, col)) {
                return true;
            }
        }
    }
    return false;
}

function backtrack(board, word, i, j, index, row, col) {
    if (index === word.length) return true;
    if (
        i < 0 ||
        j < 0 ||
        i === row ||
        j === col ||
        board[i][j] !== word[index] ||
        board[i][j] === "#"
    )
        return false;

    let t = board[i][j];
    board[i][j] = "#";

    let top = backtrack(board, word, i - 1, j, index + 1, row, col);

```

```

let right = backtrack(board, word, i, j + 1, index + 1, row, col);
let bottom = backtrack(board, word, i + 1, j, index + 1, row, col);
let left = backtrack(board, word, i, j - 1, index + 1, row, col);

board[i][j] = t;

return top || right || bottom || left;
}

```

- **Inputs:**

- **board:** A 2D array of characters representing the board.
- **word:** The target word to search for in the board.

- **Output:**

- A boolean value indicating whether the given **word** exists in the **board**.

1. **Board Dimensions:**

- Get the number of rows (**row**) and columns (**col**) of the board.

2. **Main Function - exist:**

- Initialize **index** to keep track of the current character being checked in the **word**.
- Loop through each cell in the **board**:
  - For each cell at (**i**, **j**), call the **backtrack** function to search for the **word** starting from this cell.
  - If **backtrack** returns **true**, the **word** is found, so return **true**.

3. **Backtracking Function - backtrack:**

- **Base Cases:**
  - If **index** reaches the length of the **word**, return **true** as the word has been completely found.
  - If (**i**, **j**) is out of bounds or the character at (**i**, **j**) does not match the current character in the **word**, or the cell has been visited ('#'), return **false**.

4. **Mark Cell as Visited:**

- Store the value of **board[i][j]** in **t** and replace it with '#' to mark the cell as visited.

5. **Recursion:**

- Recursively check the four possible directions (top, right, bottom, left) from the current cell using **backtrack**. Pass the updated **index** and the new (**i**, **j**) coordinates.
- If any of the recursive calls returns **true**, the word is found, so return **true**.

6. **Restore Cell Value:**

- After exploring all possible directions, restore the original value of **board[i][j]**.

7. **Return:**

- Return the result of logical OR operation between the four direction checks. If any of them returns **true**, then **true** is returned to the previous recursive call.

## 8. Final Output:

- If none of the cell-based `backtrack` calls in the `exist` function returns `true`, then return `false`, indicating that the word was not found in the board.

## Techniques Used:

- **Backtracking:** The algorithm explores all possible paths in the board while considering constraints and choices.
- **2D Array Traversal:** The code uses nested loops to traverse through the cells of the 2D board.

## Algorithm:

1. Get the dimensions of the board.
2. Loop through each cell in the board using nested loops:
  - For each cell  $(i, j)$ , call the `backtrack` function to search for the word starting from this cell.
  - If `backtrack` returns `true`, return `true` as the word is found.
3. The `backtrack` function performs the recursive backtracking search for the word:
  - Base cases are checked first to terminate the recursion.
  - If the base cases are not met, the current cell is marked as visited and the search continues recursively in the four possible directions.
  - After exploring all directions, the cell value is restored, and the result of direction checks is returned.
4. The `exist` function returns `false` if no cell-based `backtrack` call returns `true`.

## Time Complexity:

- The worst-case time complexity is  $O(N \cdot M \cdot 4^k)$ , where  $N$  and  $M$  are the dimensions of the board and  $k$  is the length of the word. This is because the algorithm tries all possible paths in the board, and each path can have at most  $4^k$  branches.

## Space Complexity:

- The space complexity is  $O(k)$ , where  $k$  is the length of the word. This is due to the recursive call stack depth.