DSA SHORT NOTES

1. Array

📌1️⃣Array Definition

An **array** is a collection of elements stored in contiguous memory locations, indexed starting from **0**.

📌 Declaration & Initialization:

```java
int[] arr = new int[5];  // Declaration of an array of size 5
int[] arr2 = {1, 2, 3, 4, 5};  // Initialization with values
```

📌2️⃣Key Array Operations & Syntax

Operation	Syntax (Java)	Time Complexity
Accessing an Element	arr[i]	O(1)
Updating an Element	arr[i] = newValue;	O(1)
Finding Length	arr.length	O(1)
Traversing an Array	for(int i = 0; i < arr.length; i++)	O(n)
Linear Search	for(int i = 0; i < arr.length; i++) if(arr[i] == target) return i;	O(n)
Binary Search (Sorted)	Arrays.binarySearch(arr, target);	O(log n)
Sorting an Array	Arrays.sort(arr);	O(n log n)
Reversing an Array	Collections.reverse(Arrays.asList(arr));	O(n)
Copying an Array	int[] newArr = Arrays.copyOf(arr, arr.length);	O(n)

📌3️⃣Array Patterns to Remember

🔹 Sliding Window (For Subarrays & Contiguous Elements)

Used for problems that involve subarrays of a fixed/variable size.

📌 **Use Case:** Maximum sum subarray of size k.

```java
int sum = 0;
for (int i = 0; i < k; i++) sum += arr[i];  // First window
for (int i = k; i < arr.length; i++) {
    sum += arr[i] - arr[i - k];  // Slide window
}
```

🔹 **Two Pointer Approach (For Sorted Arrays & Pair Problems)**

Used to avoid extra loops for finding pairs or subarrays.

```
int left = 0, right = arr.length - 1;
while (left < right) {
    int sum = arr[left] + arr[right];
    if (sum == target) return new int[]{left, right};
    else if (sum < target) left++;
    else right--;
}
```

📌 **Use Case:** Finding a pair with a given sum.

🔹 **Kadane's Algorithm (For Maximum Sum Subarray)**

```
int maxSum = Integer.MIN_VALUE, currSum = 0;
for (int num : arr) {
    currSum = Math.max(num, currSum + num);
    maxSum = Math.max(maxSum, currSum);
}
```

📌 **Use Case:** Maximum sum contiguous subarray.

🔹 **Prefix Sum (For Range Sum Queries in O(1))**

```
int[] prefixSum = new int[arr.length + 1];
for (int i = 1; i <= arr.length; i++) prefixSum[i] = prefixSum[i - 1] + arr[i - 1];
```

📌 **Use Case:** Fast computation of subarray sums.

🔹 **Hashing for Frequency Count (📌 Use Case:** Counting occurrences of numbers.**)**

```
Map<Integer, Integer> freq = new HashMap<>();
for (int num : arr) freq.put(num, freq.getOrDefault(num, 0) + 1);
```

📌5️⃣ Time Complexity Recap

Operation	Best Case	Worst Case
Access	O(1)	O(1)
Search (Linear Search)	O(1)	O(n)
Search (Binary Search - Sorted)	O(1)	O(log n)
Insertion (End)	O(1)	O(1)
Insertion (Middle/Start)	O(n)	O(n)
Deletion (End)	O(1)	O(1)
Deletion (Middle/Start)	O(n)	O(n)
Sorting	O(n log n)	O(n log n)

2. String

📌1️⃣String Definition

A **String** is a sequence of characters stored as an array but is immutable in languages like Java.

📌 Declaration & Initialization:

```
String s = "hello";  // String literal
String s2 = new String("hello");  // Using new keyword
```

📌2️⃣Key String Operations & Syntax

Operation	Syntax (Java)	Time Complexity
Get Length	s.length()	O(1)
Access Character	s.charAt(i)	O(1)
Concatenation	s1 + s2 / s.concat(s2)	O(n)
Substring	s.substring(i, j)	O(j - i)
Convert to Char Array	s.toCharArray()	O(n)
Check Equality	s1.equals(s2)	O(n)
Compare Strings	s1.compareTo(s2)	O(n)
Reverse a String	new StringBuilder(s).reverse().toString()	O(n)
Change Case	s.toUpperCase() / s.toLowerCase()	O(n)
Find Index of a Character	s.indexOf('a')	O(n)
Trim Whitespaces	s.trim()	O(n)
Replace Characters	s.replace('a', 'b')	O(n)
Split a String	s.split(" ")	O(n)

📌3️⃣String Patterns to Remember

🔹 Two Pointer Approach (For Palindromes & Reversing)

Used to compare characters from both ends efficiently.

📌 **Use Case:** Check if a string is a palindrome.

```
int left = 0, right = s.length() - 1;
while (left < right) {
    if (s.charAt(left) != s.charAt(right)) return false;
    left++; right--;
}
return true;
```

🔹 **Sliding Window (For Substrings & Anagrams)**

Used when we need to check substring properties dynamically.

```
int[] freq = new int[26];
for (int i = 0; i < k; i++) freq[s.charAt(i) - 'a']++;  // First window
for (int i = k; i < s.length(); i++) {
    freq[s.charAt(i) - 'a']++;
    freq[s.charAt(i - k) - 'a']--;
}
```

📌 **Use Case:** Finding anagrams or longest unique substring.

=>Used to count occurrences of characters.
📌 **Use Case:** Finding first unique character.

```
Map<Character, Integer> freq = new HashMap<>();
for (char ch : s.toCharArray()) freq.put(ch, freq.getOrDefault(ch, 0) + 1);
```

🔹 **KMP Algorithm (For Pattern Matching)**

Efficient for searching substrings in O(n). 📌 **Use Case:** Checking if a pattern exists in a string.

```
int[] lps = new int[pattern.length()];
int i = 1, len = 0;
while (i < pattern.length()) {
    if (pattern.charAt(i) == pattern.charAt(len)) {
        lps[i++] = ++len;
    } else if (len > 0) {
        len = lps[len - 1];
    } else {
        lps[i++] = 0;
    }
}
```

📌5️⃣ **Time Complexity Recap**

Operation	Best Case	Worst Case
Access	O(1)	O(1)
Search	O(1)	O(n)
Concatenation	O(1)	O(n)
Substring Extraction	O(1)	O(n)
Reversal	O(n)	O(n)
Sorting	O(n log n)	O(n log n)

3.Hashing

📌1️⃣What is Hashing?

Hashing is a technique used to map **keys** to **values** efficiently using a **hash function**. It helps in quick **search, insert, and delete** operations.

📌 Common Hashing Data Structures:

1. **Hash Table** - Stores key-value pairs using an array + linked list (chaining).
2. **Hash Map** - Stores key-value pairs (e.g., `HashMap` in Java).
3. **Hash Set** - Stores unique elements.
4. **Bloom Filters** - Probabilistic hash structure used for membership checking.

📌2️⃣Key Hashing Operations & Syntax

Operation	Syntax (Java)	Time Complexity
Insert Key-Value Pair	map.put(key, value);	O(1) (Avg)
Access Value by Key	map.get(key);	O(1) (Avg)
Check if Key Exists	map.containsKey(key);	O(1)
Remove Key	map.remove(key);	O(1)
Get All Keys	map.keySet();	O(n)
Get All Values	map.values();	O(n)
Iterate Through HashMap	for (Map.Entry<K, V> entry : map.entrySet())	O(n)
Insert into HashSet	set.add(value);	O(1)
Check Presence in HashSet	set.contains(value);	O(1)
Delete from HashSet	set.remove(value);	O(1)

📌3️⃣Hashing Patterns to Remember

🔹 Hashing for Frequency Count

Used to count occurrences of elements.

```
Map<Integer, Integer> freq = new HashMap<>();
for (int num : arr) freq.put(num, freq.getOrDefault(num, 0) + 1);
```

📌 **Use Case:** Finding duplicates, majority element, anagrams.

🔹 Hashing for Finding Pairs (Two Sum Problem)

Used to find pairs in O(n) instead of O(n²).

```
Map<Integer, Integer> map = new HashMap<>();
for (int num : arr) {
    if (map.containsKey(target - num)) return new int[]{map.get(target - num), num};
    map.put(num, num);
}
```

📌 **Use Case:** Finding a pair that sums to a target.

🔹 Hashing for Anagram Detection

Used to check if two strings have the same character frequencies.

```
Map<Character, Integer> freq = new HashMap<>();
for (char ch : s1.toCharArray()) freq.put(ch, freq.getOrDefault(ch, 0) + 1);
for (char ch : s2.toCharArray()) {
    if (!freq.containsKey(ch) || freq.get(ch) == 0) return false;
    freq.put(ch, freq.get(ch) - 1);
}
return true;
```

📌 **Use Case:** Checking if two strings are anagrams.

📌6️⃣ Time Complexity Recap

Operation	Best Case	Worst Case
Insert	O(1)	O(n) (for worst-case collision)
Search	O(1)	O(n)
Delete	O(1)	O(n)
Sorting Keys	O(n log n)	O(n log n)

4. Sliding Window

📌1️⃣Definition

The **Sliding Window** technique is used for **efficiently processing contiguous subarrays or substrings** of a given size or dynamically adjusting window size to solve problems involving **arrays or strings**.

- Helps reduce **time complexity** from **O(N²) → O(N)**
- Used in **subarray sum, longest substring, maximum/minimum in a subarray** problems.

📌2️⃣Types of Sliding Window

1. **Fixed-size window** → Window size remains constant (e.g., "Find max sum of K elements").
2. **Variable-size window** → Window expands/contracts based on conditions (e.g., "Find the longest substring with at most K distinct characters").

📌3️⃣Fixed-Size Sliding Window Implementation

📌 **Problem**: Find the **maximum sum of K consecutive elements** in an array.

✅ Approach:

1. Use **two pointers**: `left` (start) and `right` (end).
2. Maintain a **window sum** of size `K`.
3. Slide the window by **adding right element and removing left element**.

```java
int maxSum(int[] arr, int k) {
    int n = arr.length, maxSum = 0, windowSum = 0;

    // Compute sum of first window
    for (int i = 0; i < k; i++) {
        windowSum += arr[i];
    }

    maxSum = windowSum;

    // Slide the window
    for (int i = k; i < n; i++) {
        windowSum += arr[i] - arr[i - k];  // Add new element, remove old element
        maxSum = Math.max(maxSum, windowSum);
    }

    return maxSum;
}
```

📌4️⃣Variable-Size Sliding Window Implementation

📌 **Problem**: Find the **length of the longest subarray with sum ≤ S**.

✅ Approach:

1. **Expand the window** by adding elements to the right.
2. **Shrink** window from the left if sum exceeds S.
3. Keep track of the **maximum window size**.

```java
int longestSubarray(int[] arr, int s) {
    int left = 0, sum = 0, maxLength = 0;

    for (int right = 0; right < arr.length; right++) {
        sum += arr[right];

        while (sum > s) {  // Shrink window if sum exceeds S
            sum -= arr[left];
            left++;
        }

        maxLength = Math.max(maxLength, right - left + 1);
    }

    return maxLength;
}
```

5. Two Pointer

📌1️⃣Definition

The **Two Pointer** technique is an optimization method used to **process pairs of elements** in an array or string by using **two pointers** instead of nested loops.

- Helps reduce **time complexity** from **O(N²) → O(N)**.
- Used for **searching, sorting, merging, partitioning, and subarray problems**.

📌2️⃣Types of Two Pointer Approach

1. **Opposite Direction Pointers**
    - Used when dealing with **sorted arrays**.
    - Common for **finding pairs (e.g., sum problems, palindromes, two-sum in sorted array)**.
2. **Same Direction Pointers**
    - Used for **traversing and modifying elements** efficiently.
    - Common in **merging, removing duplicates, and longest subarray problems**.

📌3️⃣Opposite Direction Two Pointer Example

📌 **Problem**: Check if a given string is a **palindrome**.

✅ Approach

1. Use two pointers:
    - **Left pointer (`l`)** at the beginning.
    - **Right pointer (`r`)** at the end.
2. Compare characters at both pointers.
3. Move pointers inward until they meet.

```
boolean isPalindrome(String s) {
    int l = 0, r = s.length() - 1;

    while (l < r) {
        if (s.charAt(l) != s.charAt(r)) return false;
        l++;
        r--;
    }

    return true;
}
```

✔ **Time Complexity**: O(N)
✔ **Space Complexity**: O(1)

📌4️⃣ Same Direction Two Pointer Example

📌 **Problem**: Remove duplicates from a **sorted array** in-place and return the new length.

✅ Approach

1. Use two pointers:
    - `i` **(slow pointer)** tracks the last unique element.
    - `j` **(fast pointer)** scans for new unique elements.
2. When `arr[j] ≠ arr[i]`, move the unique element forward.

```java
int removeDuplicates(int[] arr) {
    if (arr.length == 0) return 0;

    int i = 0;  // Slow pointer

    for (int j = 1; j < arr.length; j++) {
        if (arr[j] != arr[i]) {
            i++;
            arr[i] = arr[j];  // Move unique element forward
        }
    }

    return i + 1;  // Length of unique elements
}
```

✔ **Time Complexity**: O(N)
✔ **Space Complexity**: O(1)

📌6️⃣ Key Points to Remember

- **Two Pointers optimize O(N²) problems to O(N)**.
- **Use opposite pointers for sum/palindrome problems**.
- **Use same-direction pointers for merging and modifying**.
- **Works best with sorted arrays and contiguous subarrays**.

6.Stack

📌1️⃣What is a Stack?

A **stack** is a **LIFO (Last-In, First-Out)** data structure where elements are inserted (pushed) and removed (popped) from the same end (the top).

📌 Key Applications of Stacks:
✔️ Function call management (Recursion)
✔️ Undo/Redo operations
✔️ Expression evaluation (postfix, infix, prefix)
✔️ Backtracking (e.g., maze solving, DFS)
✔️ Parentheses matching

📌2️⃣Stack Operations & Syntax

Operation	Syntax (Java - Stack)	Time Complexity
Push (Insert at Top)	stack.push(value);	O(1)
Pop (Remove Top)	stack.pop();	O(1)
Peek (Top Element)	stack.peek();	O(1)
Check if Empty	stack.isEmpty();	O(1)
Size of Stack	stack.size();	O(1)

📌 In Java (Using `Stack` class):

```
Stack<Integer> stack = new Stack<>();
stack.push(10);
stack.push(20);
System.out.println(stack.pop());  // 20
System.out.println(stack.peek()); // 10
```

📌3️⃣Stack Implementations

1️⃣Using Arrays (Fixed Size)

- Simple but has a predefined size limitation.

2️⃣Using Linked List (Dynamic Size)

- No size limitation, but requires extra memory for pointers.

📌4️⃣ Stack Patterns

🔹 **Monotonic Stack** → Used in problems where elements need to be processed in increasing or decreasing order.
- 🔹 **Min Stack** → Stack that supports retrieving the minimum element in O(1).
- 🔹 **Two Stacks in One Array** → Efficient way to use limited space.
- 🔹 **Stack Implementation using Queues** → Stack using two queues.
- 🔹 **Queue Implementation using Stacks** → Queue using two stacks.

📌6️⃣ Time Complexity Recap

Operation	Best Case	Worst Case
Push	O(1)	O(1)
Pop	O(1)	O(1)
Peek	O(1)	O(1)
Search	O(n)	O(n)

7.Queue

📌1️⃣ Definition

A **queue** is a **FIFO (First-In, First-Out)** data structure where elements are inserted at the **rear** and removed from the **front**.
Queues are useful for managing tasks in **scheduling**, **buffers**, and scenarios where order of processing matters.

📌 Key Applications:
✔️ Task Scheduling (OS and CPU scheduling)
✔️ BFS (Breadth-First Search)
✔️ Print Queue/Job Processing
✔️ Order Processing Systems (e.g., Bank Queue)
✔️ Circular Buffers

📌2️⃣ Queue Operations & Syntax

Operation	Syntax (Java - Queue Interface)	Time Complexity
Enqueue (Insert at Rear)	queue.offer(value);	O(1)
Dequeue (Remove from Front)	queue.poll();	O(1)
Peek (Front Element)	queue.peek();	O(1)
Check if Empty	queue.isEmpty();	O(1)
Size of Queue	queue.size();	O(1)

📌 In Java (Using LinkedList as Queue):

```java
import java.util.Queue;
import java.util.LinkedList;

public class QueueExample {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(10); // Enqueue
        queue.offer(20);
        System.out.println(queue.poll());  // Dequeue -> 10
        System.out.println(queue.peek());  // Peek -> 20
    }
}
```

📌3️⃣ Types of Queues

🔹1️⃣ Simple Queue

Basic queue where **elements** are inserted at the **rear** and removed from the **front**.

🔹2️⃣ Circular Queue

In a circular queue, the **rear** and **front** pointers wrap around when they reach the end. This helps in utilizing the entire array space efficiently.

🔹3️⃣ Priority Queue

A **priority queue** stores elements in order of **priority**, rather than the order of insertion. A higher priority element is always dequeued before a lower priority element.

- **Min-Heap**: The element with the lowest value has the highest priority.
- **Max-Heap**: The element with the highest value has the highest priority.

🔹4️⃣ Deque (Double-Ended Queue)

A **deque** allows adding and removing elements from **both ends** (front and rear). It's often used for problems like sliding window, or when you need access to both ends.

📌5️⃣ Time Complexity Recap

Operation	Time Complexity
Enqueue (Insert at Rear)	O(1)
Dequeue (Remove from Front)	O(1)
Peek (Front Element)	O(1)
Search	O(n)

8. Linked List

📌1️⃣Definition

A **linked list** is a linear **data structure** where elements (called **nodes**) are stored in **non-contiguous memory** locations. Each node contains:

1. **Data** - The value stored.
2. **Next (pointer/reference)** - A reference to the next node in the sequence.

📌 **Key Applications:**
✔️ Dynamic Memory Allocation
✔️ Implementing Queues, Stacks, Graphs, and Hash Tables
✔️ Memory-Efficient Data Structures (e.g., when frequent insertions/deletions are required)

📌1️⃣Definition

A linked list is a linear data structure where elements (nodes) are stored at different memory locations and are connected using pointers. It allows efficient insertions and deletions but has slower access times compared to arrays**.**

📌2️⃣Types of Linked Lists

1. **Singly Linked List** → Each node points to the next node.
2. **Doubly Linked List** → Each node has pointers to both the next and previous nodes.
3. **Circular Linked List** → The last node connects back to the first node.
4. **Circular Doubly Linked List** → A doubly linked list where the last node links to the first node.

📌3️⃣Node Structure (Java)

A linked list consists of nodes where each node has:

- **Data** (value stored in the node)
- **Pointer** (reference to the next node)

```java
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;
        this.next = null;
    }
}
```

📌4️⃣Basic Operations

4.1 Insert at Head

```
ListNode insertAtHead(ListNode head, int val) {
    ListNode newNode = new ListNode(val);
    newNode.next = head;
    return newNode;
}
```

4.2 Insert at Tail

```
ListNode insertAtTail(ListNode head, int val) {
    ListNode newNode = new ListNode(val);
    if (head == null) return newNode;

    ListNode temp = head;
    while (temp.next != null) {
        temp = temp.next;
    }
    temp.next = newNode;
    return head;
}
```

4.3 Delete a Node by Value

```
ListNode deleteNode(ListNode head, int val) {
    if (head == null) return null;
    if (head.val == val) return head.next;

    ListNode temp = head;
    while (temp.next != null && temp.next.val != val) {
        temp = temp.next;
    }
    if (temp.next != null) {
        temp.next = temp.next.next;
    }
    return head;
}
```

4.4 Search a Node

```
boolean search(ListNode head, int key) {
    while (head != null) {
        if (head.val == key) return true;
        head = head.next;
    }
    return false;
}
```

📌3️⃣ Operations on Linked List

Operation	Syntax	Time Complexity
Insertion (at front)	addFirst(value);	O(1)
Insertion (at end)	addLast(value);	O(n) (for singly) or O(1) (for doubly)
Insertion (at specific position)	add(index, value);	O(n)
Deletion (from front)	removeFirst();	O(1)
Deletion (from end)	removeLast();	O(n) (for singly) or O(1) (for doubly)
Deletion (from specific position)	remove(index);	O(n)
Search (by value)	contains(value);	O(n)
Traversal	printList();	O(n)

📌6️⃣ Time Complexity Recap

Operation	Time Complexity
Insertion (at front)	O(1)
Insertion (at end)	O(n) (for singly) or O(1) (for doubly)
Insertion (at specific position)	O(n)
Deletion (from front)	O(1)
Deletion (from end)	O(n) (for singly) or O(1) (for doubly)
Deletion (from specific position)	O(n)
Search (by value)	O(n)
Traversal	O(n)

9. Binary Search

📌1️⃣ Definition

Binary Search is an efficient algorithm for searching an element in a **sorted array** by repeatedly dividing the search interval in half.

📌2️⃣ Binary Search Operations

Operation	Syntax (Java)	Time Complexity
Binary Search	binarySearch(arr, target)	O(log n)
Binary Search (Recursive)	binarySearch(arr, left, right, target)	O(log n)

Recursive Method Syntax (Java)

```java
public class BinarySearch {
    public static int binarySearch(int[] arr, int left, int right, int target) {
        if (left > right) return -1;

        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] > target) return binarySearch(arr, left, mid - 1, target);
        return binarySearch(arr, mid + 1, right, target);
    }
}
```

📌3️⃣ Binary Search Variants

1. **Finding the First Occurrence**
   Modify the algorithm to continue searching on the left even after finding the target.

2. **Finding the Last Occurrence**
   Modify the algorithm to continue searching on the right even after finding the target.

3. **Finding the Closest Element**
   Adjust the search space to return the closest element if the target is not found.

📌4️⃣ Time Complexity

- **Best Case:** O(1) (Target is at mid)
- **Average & Worst Case:** O(log n)

10.Tree

📌1️⃣Definition

A **tree** is a hierarchical data structure consisting of nodes connected by edges. The top node is called the **root**, and each node contains a value or data and references (or links) to its children.

Basic Terminology:

- **Node:** An element of the tree containing data.
- **Root:** The topmost node.
- **Child:** A node directly connected to another node when moving away from the root.
- **Parent:** A node directly connected to another node when moving towards the root.
- **Leaf:** A node with no children.
- **Height:** The length of the longest path from the root to a leaf.
- **Depth:** The length of the path from the root to a node.
- **Subtree:** A tree consisting of a node and its descendants.

📌3️⃣Tree Node Structure (Java)

```java
class TreeNode {
    int val;
    TreeNode left, right;

    TreeNode(int val) {
        this.val = val;
        this.left = this.right = null;
    }
}
```

📌2️⃣Types of Trees

🔹1️⃣Binary Tree

Each node has at most **two children** (left and right).

🔹2️⃣Binary Search Tree (BST)

A **binary tree** where:

- Left subtree contains only nodes with **smaller values**.
- Right subtree contains only nodes with **larger values**.

🔹 3️⃣ AVL Tree (Self-balancing BST)

A **balanced binary search tree** where the difference between the heights of left and right subtrees is at most **1**.

🔹 4️⃣ Heap

A **binary tree** that satisfies the **heap property**:

- **Max-Heap:** Parent nodes are greater than or equal to their children.
- **Min-Heap:** Parent nodes are smaller than or equal to their children.

📌 3️⃣ **Tree Operations**

Operation	Syntax	Time Complexity
Insertion	insert(root, value);	O(log n) (for BST)
Deletion	delete(root, value);	O(log n) (for BST)
Search	search(root, value);	O(log n) (for BST)
Traversal	inOrderTraversal(root);	O(n)
Preorder	preOrderTraversal(root);	O(n)
Postorder	postOrderTraversal(root);	O(n)
Level Order	levelOrderTraversal(root);	O(n)

📌 4️⃣ Tree Traversals

1. **Inorder Traversal** (Left, Root, Right)
    - Used for **BSTs** to get nodes in **ascending order**.
2. **Syntax:**

```
void inOrder(Node root) {
    if (root != null) {
        inOrder(root.left);
        System.out.print(root.data + " ");
        inOrder(root.right);
    }
}
```

Preorder Traversal (Root, Left, Right)

- Used to **copy** a tree or create a **prefix expression**.

Syntax:

```
void preOrder(Node root) {
    if (root != null) {
        System.out.print(root.data + " ");
        preOrder(root.left);
        preOrder(root.right);
    }
}
```

Postorder Traversal (Left, Right, Root)

- Used for **deletion** or **postfix expressions**.

Syntax:

```
void postOrder(Node root) {
    if (root != null) {
        postOrder(root.left);
        postOrder(root.right);
        System.out.print(root.data + " ");
    }
}
```

Level Order Traversal (Breadth-first)

- Traverse the tree level by level from top to bottom.

Syntax:

```
void levelOrder(Node root) {
    if (root == null) return;
    Queue<Node> queue = new LinkedList<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        Node temp = queue.poll();
        System.out.print(temp.data + " ");
        if (temp.left != null) queue.add(temp.left);
        if (temp.right != null) queue.add(temp.right);
    }
}
```

📌5️⃣ Time Complexity for Operations

Operation	Time Complexity
Search	O(log n) (BST)
Insertion	O(log n) (BST)
Deletion	O(log n) (BST)
Traversal	O(n)
Level Order	O(n)

📌7️⃣ Common Tree Patterns

- **Balanced Trees:** AVL trees, Red-Black trees (balancing on insertion and deletion).
- **Complete Trees:** A binary tree in which every level, except possibly the last, is completely filled.
- **Height of a Tree:** Calculated from the root to the deepest leaf.

11.Trie

📌1️⃣ Definition

A **Trie** (or **Prefix Tree**) is a tree-like data structure that stores a dynamic set of strings, where each node represents a **character** of a string. It is primarily used for **fast string search** and **prefix matching**.

Key Terminology:

- **Root:** The starting point of the trie, usually empty.
- **Node:** Each node represents a **character** of the string.
- **Edge:** A link between two nodes.
- **Leaf:** A node representing the end of a string.
- **Word:** A complete string represented by the path from the root to a leaf node.

📌2️⃣ Trie Operations

Operation	Syntax	Time Complexity
Insert	insert(root, word)	O(m)
Search	search(root, word)	O(m)
StartsWith	startsWith(root, prefix)	O(m)
Delete	delete(root, word)	O(m)

Where m is the length of the string (word or prefix).

📌3️⃣ Trie Node Structure (Java)

```java
class TrieNode {
    TrieNode[] children = new TrieNode[26]; // For lowercase English letters
    boolean isEndOfWord = false;  // True if this node represents the end of a word
}

class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }
```

📌4️⃣ Trie Operations

🔹 Insert Operation

To insert a word, start from the root node and traverse through the trie. If a node does not exist for a character, create a new node. Mark the last node of the word as `isEndOfWord = true`.

🔹 Search Operation

To search for a word, follow the same steps as in the insert operation. If a node for a character does not exist, return `false`. After traversing all characters, check if the last node is marked as `isEndOfWord = true`.

🔹 StartsWith Operation

To check if a prefix exists, traverse the trie similarly to the search operation but do not require the last node to be `isEndOfWord`.

🔹 Delete Operation

To delete a word, follow the search path to find the node corresponding to the last character of the word. Then mark `isEndOfWord = false`. If that node has no children, delete the node. This operation is more complicated than insert/search due to potential cleanup of empty nodes.

📌5️⃣ Time Complexity

- **Insert Operation:** O(m) where m is the length of the word.
- **Search Operation:** O(m) where m is the length of the word.
- **StartsWith Operation:** O(m) where m is the length of the prefix.
- **Delete Operation:** O(m) where m is the length of the word.

📌7️⃣ Space Complexity

The space complexity of a trie depends on the number of words and their lengths. For `n` words, each of length `m`, the worst-case space complexity is O(n * m), because each node stores `26` pointers (for each lowercase letter).

12.Heap/Priority Queue

📌1️⃣ Definition

A **Heap** is a special tree-based data structure that satisfies the **heap property**:

- **Max Heap:** The value of each node is greater than or equal to the values of its children.
- **Min Heap:** The value of each node is less than or equal to the values of its children.

A **Priority Queue** is an abstract data type that supports operations to insert elements and remove the element with the **highest priority**. It can be implemented using a heap, where:

- **Max Priority Queue:** Removes the largest element.
- **Min Priority Queue:** Removes the smallest element.

📌2️⃣ Heap Operations

Operation	Syntax	Time Complexity
Insert	insert(heap, element)	O(log n)
Extract-Max	extractMax(heap)	O(log n)
Extract-Min	extractMin(heap)	O(log n)
Peek (Max/Min)	peek(heap)	O(1)
Heapify	heapify(arr)	O(n)
Delete	delete(heap, index)	O(log n)

Where n is the number of elements in the heap.

📌3️⃣ Heap Node Structure (Java)

```
class MaxHeap {
    int[] heap;
    int size;

    public MaxHeap(int capacity) {
        heap = new int[capacity];
        size = 0;
    }
```

📌4️⃣ Priority Queue Operations (Java)

Max Priority Queue (Using a Max Heap)

```
PriorityQueue<Integer> maxQueue = new PriorityQueue<>(Collections.reverseOrder());
maxQueue.add(10);
maxQueue.add(20);
maxQueue.add(5);

int max = maxQueue.poll(); // Removes and returns the max element (20)
```

Min Priority Queue (Using a Min Heap)

```
PriorityQueue<Integer> minQueue = new PriorityQueue<>();
minQueue.add(10);
minQueue.add(20);
minQueue.add(5);

int min = minQueue.poll(); // Removes and returns the min element (5)
```

📌5️⃣ Time Complexity of Operations

Operation	Time Complexity
Insert	O(log n)
Extract-Max/Min	O(log n)
Peek (Max/Min)	O(1)
Heapify	O(n)
Delete	O(log n)

📌7️⃣ Use Cases of Heaps/Priority Queues

- **Dijkstra's Algorithm** (for shortest path).
- **Huffman Encoding** (compression algorithms).
- **Task Scheduling** (priority-based task execution).
- **K-th Largest/Smallest Element** in an array.
- **Real-time Data Streaming** (median finding).

13. Intervals

📌1️⃣ Definition

An **interval** is a range of values, typically used to represent a continuous sequence of numbers. In the context of algorithms and data structures, intervals are often represented as pairs of integers, such as `[start, end]`.

Types of Intervals:

- **Closed Interval**: `[start, end]` includes both endpoints.
- **Open Interval**: `(start, end)` excludes both endpoints.
- **Half-Open Interval**: `[start, end)` or `(start, end]` includes one endpoint but not the other.

📌2️⃣ Interval Operations

Operation	Syntax	Time Complexity
Merge Intervals	mergeIntervals(intervals)	O(n log n)
Insert Interval	insertInterval(intervals, newInterval)	O(n)
Interval Intersection	intervalIntersection(intervals1, intervals2)	O(n + m)
Interval Removal/Deletion	removeInterval(intervals, toRemove)	O(n)
Overlapping Intervals Check	hasOverlap(intervals)	O(n)

Where n is the number of intervals.

📌3️⃣ Merge Intervals

Problem: Given a collection of intervals, merge all overlapping intervals.

Algorithm:

1. **Sort** the intervals by the start time.
2. Traverse through the sorted intervals, merging intervals that overlap.
3. If two intervals don't overlap, add the current interval to the result.

📌4 Insert Interval

Insert a new interval into a list of non-overlapping intervals and return the new list of intervals.

Algorithm:

1. **Sort** the intervals (if not already sorted).
2. Find the position to insert the new interval.
3. Merge any overlapping intervals.

📌5 Interval Intersection

Given two lists of intervals, return the intersection of these two intervals.

Algorithm:

1. Sort both intervals by their start times.
2. Use two pointers to traverse through both intervals.
3. If there's an overlap, add the intersection to the result.

📌6 Time Complexity

Operation	Time Complexity
Merge Intervals	O(n log n)
Insert Interval	O(n)
Interval Intersection	O(n + m)
Interval Removal	O(n)
Overlapping Check	O(n)

Where n and m are the number of intervals in the respective lists.

📌7 Interval Overlap Check

Problem: Check if two intervals overlap.

Algorithm:

- Two intervals [a1, a2] and [b1, b2] overlap if:
  - a1 <= b2 and b1 <= a2

📌8 Use Cases of Intervals

- **Scheduling Problems**: Finding overlapping schedules or gaps between schedules.
- **Calendar Applications**: Merging events that overlap.
- **Range Queries**: Searching for data within a specified range.
- **Resource Allocation**: Ensuring resources are allocated within time slots without conflicts.

14. Greedy Algorithms

📌1️⃣Definition

A **Greedy Algorithm** is an approach for solving optimization problems by making a sequence of choices, each of which looks the best at the moment. The idea is to choose the local optimum at each step with the hope of finding the global optimum.

Greedy algorithms are often used when a problem can be broken down into stages where decisions at each stage are made based on local optimal choices.

📌2️⃣Characteristics of Greedy Algorithms

1. **Greedy Choice Property**: A global optimum can be arrived at by selecting a local optimum.
2. **Optimal Substructure**: A problem has optimal substructure if an optimal solution to the problem contains optimal solutions to the subproblems.

Note: Greedy algorithms do not always guarantee an optimal solution, but for certain problems (like Huffman Coding, Dijkstra's Algorithm), they are proven to give the best result.

📌3️⃣Greedy Algorithm Steps

1. **Initialize**: Set the initial solution to the problem.
2. **Iterate**: For each step, select the next local optimal choice.
3. **Check**: Evaluate the selected solution.
4. **Repeat**: If the solution is not complete, repeat the process with the next choice.
5. **Terminate**: When the solution is complete, return the solution.

📌4️⃣Greedy Algorithm Operations/Applications

Operation	Description	Time Complexity
Activity Selection	Select the maximum number of non-overlapping activities.	O(n log n)
Fractional Knapsack	Maximize value with a given weight capacity using fractional items.	O(n log n)
Huffman Coding	Build an optimal prefix code for encoding.	O(n log n)
Dijkstra's Shortest Path Algorithm	Find the shortest path in a graph.	O(V log V + E)
Prim's Algorithm	Find the minimum spanning tree of a graph.	O(V log V + E)

Where n is the number of items, V is the number of vertices, and E is the number of edges.

📌5️⃣ Greedy Algorithm Examples

1. Activity Selection Problem

Given a set of activities with start and finish times, select the maximum number of non-overlapping activities.

Algorithm:

1. Sort the activities based on their finish times.
2. Iterate through the activities, selecting those that start after the last selected one end

```java
public static void activitySelection(int[] start, int[] end) {
    int n = start.length;
    List<int[]> activities = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        activities.add(new int[] {start[i], end[i]});
    }

    // Sort activities by end time
    activities.sort(Comparator.comparingInt(a -> a[1]));

    int lastEndTime = -1;
    for (int[] activity : activities) {
        if (activity[0] >= lastEndTime) {
            System.out.println("Activity: (" + activity[0] + ", " + activity[1] + ")");
            lastEndTime = activity[1];
        }
    }
}
```

2. Fractional Knapsack Problem

Given a set of items, each with a weight and value, and a knapsack with a given capacity, determine the maximum value that can be obtained by filling the knapsack with fractions of items.

Algorithm:

1. Calculate the value-to-weight ratio for each item.
2. Sort the items by the value-to-weight ratio in descending order.
3. Add items (or their fractions) to the knapsack until it is full.

4. Dijkstra’s Algorithm (Greedy for Shortest Path)

This algorithm finds the shortest path between a source node and all other nodes in a graph with non-negative weights.

Algorithm:

1. Initialize the distance to the source node as 0 and all other nodes as infinity.
2. Use a priority queue to always choose the next node with the smallest tentative distance.
3. Update the distance of the neighbors based on the chosen node.

Algorithm	Time Complexity
Activity Selection	O(n log n)
Fractional Knapsack	O(n log n)
Huffman Coding	O(n log n)
Dijkstra’s Algorithm	O(V log V + E)
Prim’s Algorithm	O(V log V + E)

Where n is the number of activities/items and V, E are the vertices and edges of the graph, respectively.

📌7️⃣ Use Cases of Greedy Algorithms

- **Resource Allocation** (e.g., Knapsack problem).
- **Pathfinding Algorithms** (e.g., Dijkstra's and Prim's algorithm).
- **Huffman Coding** for data compression.
- **Scheduling Problems** (e.g., Activity Selection).
- **Currency Denominations** (e.g., finding the least number of coins needed).

15. Recursion

📌1️⃣Definition

Recursion is a programming technique where a function calls itself to solve a smaller instance of the same problem. It breaks a problem into smaller subproblems until it reaches a base case, which does not require further recursion.

📌2️⃣Base and Recursive Cases

Every recursive function should have:

- **Base Case**: The condition that stops the recursion. Without this, the function would call itself indefinitely.
- **Recursive Case**: The part where the function calls itself with a modified argument to move closer to the base case.

📌3️⃣Recursion Syntax

In most programming languages, the general structure of a recursive function is:

```
returnType functionName(parameters) {
    if (base condition) {
        return base case result;
    }
    // Recursive step
    return functionName(modified parameters);
}
```

📌4️⃣Common Recursion Patterns

4.1 Factorial Calculation

A classic example of recursion is calculating the factorial of a number n:

Factorial Formula:

```
n! = n * (n-1)!
```

For `n = 0`, `0! = 1` is the base case.

```
int factorial(int n) {
    if (n == 0) {
        return 1; // Base case
    }
    return n * factorial(n - 1); // Recursive case
}
```

4.2 Fibonacci Sequence

The Fibonacci sequence is another classic example, where each term is the sum of the two preceding ones.

Fibonacci Formula:

```
int fibonacci(int n) {
    if (n <= 1) {
        return n; // Base cases
    }
    return fibonacci(n - 1) + fibonacci(n - 2); // Recursive case
}
```

📌6️⃣ Key Points to Remember

- **Base Case**: Ensure that every recursive function has a well-defined base case to terminate the recursion.
- **Recursive Case**: Ensure that each recursive call moves closer to the base case.
- **Memory Consumption**: Recursive calls consume stack space, so avoid excessive recursion depth in some cases.
- **Debugging**: Recursive solutions can be difficult to debug; visualize recursion tree if necessary.

The time complexity of a recursive function depends on the number of recursive calls and the work done in each call.

- **Factorial**: O(n)
- **Fibonacci**: O(2^n) (Exponential without memoization)
- **Binary Search**: O(log n)
- **Tree Traversals**: O(n) (where n is the number of nodes in the tree)

📌8️⃣ Space Complexity

The space complexity of a recursive solution depends on the depth of the recursion tree (the maximum number of function calls on the call stack). For most recursive algorithms:

- **Factorial**: O(n) (in worst case)
- **Fibonacci (naive)**: O(n)
- **Binary Search**: O(log n)
- **Tree Traversal**: O(h), where h is the height of the tree

📌9️⃣ Recursion vs Iteration

- **Recursion**: Can be simpler and more intuitive for problems like tree traversal, backtracking, or divide and conquer algorithms.
- **Iteration**: Generally more efficient in terms of space, as it doesn't add new stack frames.

16. Backtracking

📌1️⃣ Definition

Backtracking is an algorithmic technique for solving problems by incrementally building solutions and abandoning a solution as soon as it is determined that it cannot possibly lead to a valid solution.

It is a form of **depth-first search (DFS)** where, when a problem cannot be solved any further, it **backtracks** to the previous step and tries another possibility.

📌2️⃣ Backtracking Approach

1. **Choice**: Make a choice at each step.
2. **Constraint**: Check if the current solution satisfies the problem's constraints.
3. **Goal**: If the solution meets the goal, record it.
4. **Backtrack**: If no valid solution is found, backtrack to the previous choice and try another path.
5. **Repeat** until all paths are explored.

📌3️⃣ Operations/Applications of Backtracking

Operation	Description	Time Complexity
Subset Generation	Generate all subsets of a set.	O(2^n)
Permutations	Generate all possible permutations of a set.	O(n!)
N-Queens Problem	Place N queens on an N x N chessboard so that no two queens threaten each other.	O(N!)
Sudoku Solver	Solve a given Sudoku puzzle.	O(9^81)
Hamiltonian Path	Find a path in a graph that visits each node exactly once.	O(N!)

Where n is the size of the input (e.g., number of elements in a set, or number of nodes).

📌4️⃣ Backtracking Example Problems

1. N-Queens Problem

The N-Queens problem is a classic problem where the goal is to place N queens on an NxN chessboard such that no two queens threaten each other.

Algorithm:

1. Try placing a queen on each row.
2. For each column in the row, check if placing a queen leads to a solution.
3. If placing a queen does not lead to a valid configuration, backtrack and try another column or row.

2. Subset Generation

Given a set of numbers, generate all possible subsets (the power set).

Algorithm:

1. Start from an empty subset.
2. For each element, either include it in the subset or exclude it.
3. Use recursion to generate all possible combinations.

📌5️⃣ Time Complexity of Backtracking

Problem	Time Complexity
Subset Generation	O(2^n)
Permutations	O(n!)
N-Queens Problem	O(N!)
Sudoku Solver	O(9^81)
Hamiltonian Path	O(N!)

Where n is the size of the set (for subsets or permutations) or the size of the grid (for Sudoku).

📌6️⃣ Use Cases of Backtracking

- **Puzzle Solving**: Sudoku, crosswords, and N-Queens.
- **Combinatorial Problems**: Generating subsets, permutations, combinations.
- **Constraint Satisfaction Problems**: Solving problems that require satisfying certain constraints (e.g., Sudoku).
- **Graph Traversal**: Finding paths, Hamiltonian cycles.

17. Graphs

📌1️⃣Definition

A **Graph** is a collection of nodes (vertices) connected by edges (arcs). Graphs are used to model relationships between pairs of objects, like social networks, web pages, cities, and much more.

There are two primary types of graphs:

- **Directed Graph (Digraph)**: Each edge has a direction, going from one vertex to another.
- **Undirected Graph**: Edges have no direction, meaning the relationship between the vertices is mutual.

📌2️⃣Graph Terminology

- **Vertex (V)**: A node in the graph.
- **Edge (E)**: A connection between two vertices.
- **Adjacent Vertices**: Two vertices connected by an edge.
- **Degree**: The number of edges incident to a vertex.
    - **In-degree** (for directed graphs): The number of edges coming into a vertex.
    - **Out-degree** (for directed graphs): The number of edges going out of a vertex.
- **Path**: A sequence of edges that connect a sequence of vertices.
- **Cycle**: A path that starts and ends at the same vertex without repeating any other vertices.
- **Connected Graph**: A graph in which there is a path between every pair of vertices.
- **Disconnected Graph**: A graph where at least one pair of vertices is not connected by a path.

📌3️⃣Types of Graphs

- **Weighted Graph**: Each edge has an associated weight or cost.
- **Unweighted Graph**: Edges do not have weights.
- **Complete Graph**: A graph in which there is an edge between every pair of vertices.
- **Bipartite Graph**: A graph whose vertex set can be divided into two disjoint sets, such that no two vertices within the same set are adjacent.
- **Tree**: A connected graph with no cycles.
- **Forest**: A collection of disjoint trees.

📌4️⃣ Graph Representation

Graphs can be represented in two common ways:

1. **Adjacency Matrix**:
    - A 2D array of size `V x V`, where each element indicates whether there is an edge between the corresponding pair of vertices.
    - **Space Complexity**: O(V²).
2. **Example (for an undirected graph)**:

```
int[][] adjMatrix = new int[V][V];
adjMatrix[u][v] = 1;  // Edge from vertex u to vertex v
adjMatrix[v][u] = 1;  // Edge from vertex v to vertex u (since it's undirected)
```

Adjacency List:

- A list of lists, where each list at index `i` contains all the vertices adjacent to vertex `i`.
- **Space Complexity**: O(V + E), where E is the number of edges.

Example:

```
List<List<Integer>> adjList = new ArrayList<>();
for (int i = 0; i < V; i++) {
    adjList.add(new ArrayList<>());
}
adjList.get(u).add(v);  // Edge from u to v
adjList.get(v).add(u);  // Edge from v to u (for undirected graph)
```

📌5️⃣ Graph Traversal

1. Depth-First Search (DFS)

DFS is a traversal technique that explores as far as possible along each branch before backtracking.

- **Time Complexity**: O(V + E) for both directed and undirected graphs.
- **Space Complexity**: O(V) due to the recursion stack.

Java Code (DFS using Recursion):

```java
private static void dfs(int v, List<List<Integer>> adjList, boolean[] visited) {
    visited[v] = true;
    System.out.print(v + " ");

    for (int neighbor : adjList.get(v)) {
        if (!visited[neighbor]) {
            dfs(neighbor, adjList, visited);
        }
    }
}
```

2. Breadth-First Search (BFS)

BFS explores the graph level by level, starting from a source node and visiting all of its neighbors before moving on to their neighbors.

- **Time Complexity**: O(V + E).
- **Space Complexity**: O(V) for the queue.

Java Code (BFS using Queue):

```java
private static void bfs(int start, List<List<Integer>> adjList) {
    int V = adjList.size();
    boolean[] visited = new boolean[V];
    Queue<Integer> queue = new LinkedList<>();

    visited[start] = true;
    queue.offer(start);

    while (!queue.isEmpty()) {
        int node = queue.poll();
        System.out.print(node + " ");

        for (int neighbor : adjList.get(node)) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                queue.offer(neighbor);
            }
        }
    }
}
```

📌6️⃣ Graph Algorithms

1. Dijkstra's Shortest Path Algorithm (for Weighted Graphs)

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative weights.

- **Time Complexity**: O((V + E) log V) using a priority queue.
- **Space Complexity**: O(V).

2. Bellman-Ford Algorithm (for Weighted Graphs)

This algorithm computes the shortest paths from a source vertex to all other vertices, even with negative weights.

- **Time Complexity**: O(V * E).
- **Space Complexity**: O(V).

3. Floyd-Warshall Algorithm (for All-Pairs Shortest Paths)

Floyd-Warshall computes shortest paths between all pairs of vertices in a weighted graph.

- **Time Complexity**: $O(V^3)$.
- **Space Complexity**: $O(V^2)$.

4. Kruskal's Algorithm (for Minimum Spanning Tree)

Kruskal's algorithm finds a minimum spanning tree for a connected, undirected graph.

- **Time Complexity**: O(E log E).
- **Space Complexity**: O(E).

5. Prim's Algorithm (for Minimum Spanning Tree)

Prim's algorithm also finds a minimum spanning tree but grows the tree by adding vertices one by one.

- **Time Complexity**: $O(V^2)$ using an adjacency matrix, O(E + V log V) with a priority queue.
- **Space Complexity**: O(V).

📌7️⃣ **Time Complexity of Common Graph Operations**

Operation	Time Complexity
DFS	O(V + E)
BFS	O(V + E)
Dijkstra's Algorithm	O((V + E) log V)
Bellman-Ford	O(V * E)
Floyd-Warshall	$O(V^3)$
Kruskal's Algorithm	O(E log E)
Prim's Algorithm	$O(V^2)$ or O(E + V log V)

Where V is the number of vertices and E is the number of edges.

📌2️⃣ Advanced Graph Algorithms

2.1 Maximum Flow Problem - Ford-Fulkerson Algorithm

The **Maximum Flow** problem involves finding the maximum flow from a source node to a sink node in a flow network.

- **Ford-Fulkerson Algorithm** is the most common approach to solving the maximum flow problem.
- It uses augmenting paths to find the maximum flow in a graph.

Key Concepts

- **Residual Graph**: A graph showing remaining capacities of edges after some flow has been pushed through.
- **Augmenting Path**: A path from source to sink that can carry additional flow.

Time Complexity:

- O(max flow × number of edges), though it depends on how augmenting paths are chosen.

2.2 Shortest Path Algorithms (Advanced)

Dijkstra's Algorithm (Optimized)

- **Dijkstra's algorithm** can be optimized using **priority queues (min-heaps)**.
- **Time Complexity**: O((V + E) log V) using a priority queue.

For **negative weights**, Dijkstra doesn't work, and other algorithms must be used.

Bellman-Ford Algorithm (for Negative Weights)

The **Bellman-Ford** algorithm computes shortest paths from a source to all other vertices, even in graphs with negative edge weights.

- **Time Complexity**: O(V * E).
- **Can detect negative weight cycles**.

Johnson's Algorithm (for All-Pairs Shortest Path)

This algorithm works by first running **Bellman-Ford** to reweight the edges, making all edge weights non-negative, then running **Dijkstra** for each vertex.

- **Time Complexity**: $O(V^2 \log V + VE)$ (for sparse graphs).

2.3 Topological Sorting

Topological Sorting is a linear ordering of vertices in a Directed Acyclic Graph (DAG) such that for every directed edge (u, v), vertex u comes before vertex v in the ordering.

- **Applications**: Task scheduling, compilation ordering.
- **Approach**: Can be done using **DFS** or **Kahn's Algorithm** (BFS-based).

Time Complexity: O(V + E)

2.4 Articulation Points and Bridges

- **Articulation Points**: A vertex whose removal increases the number of connected components in a graph.
- **Bridges**: An edge whose removal increases the number of connected components in a graph.

These concepts are important in network design (for identifying critical points).

DFS-based approach:

- Use **DFS** to detect articulation points and bridges by tracking discovery and low values of vertices.

📌3️⃣ Advanced Graph Representation

3.1 Adjacency Matrix (for Dense Graphs)

- **Space Complexity**: $O(V^2)$ (useful for dense graphs).
- **Operations**: Checking if there's an edge between two vertices can be done in constant time O(1).

3.2 Adjacency List (for Sparse Graphs)

- **Space Complexity**: O(V + E) (useful for sparse graphs).
- **Operations**: Neighbors of a vertex can be accessed in O(k) time, where k is the number of neighbors.

3.3 Incidence Matrix

An **Incidence Matrix** represents a graph by indicating the edges incident to the vertices.

- **Space Complexity**: O(V * E)
- Useful for edge-centric algorithms like **maximum flow**.

📌4️⃣ Important Graph Theorems

4.1 Euler's Theorem (for Eulerian Circuits)

- **Eulerian Circuit**: A cycle that visits every edge of the graph exactly once.
- **Conditions**: A connected graph has an Eulerian Circuit if and only if every vertex has an even degree.

4.2 Fleury's Algorithm (for Eulerian Path/Circuit)

- An algorithm to find an Eulerian path or circuit in a graph, based on conditions for edge traversal.

📌5️⃣ Applications of Advanced Graphs

1. **Network Flow**: Used in transportation, communication, and data networks to model the flow of resources.
2. **Matching Problems**: Bipartite graph matching for job assignments, marriage problems, etc.
3. **Social Networks**: Analyzing relationships, influence, and connectivity.

4. **Recommendation Systems**: Collaborative filtering and graph-based recommendation engines.
5. **Route Planning**: Shortest path algorithms applied in navigation systems (Google Maps).
6. **Network Reliability**: Articulation points and bridges to determine critical points in communication networks.
7. **Cycle Detection**: Used in scheduling and deadlock detection.

📌6 Time Complexity Summary for Advanced Graph Algorithms

Algorithm	Time Complexity
Ford-Fulkerson (Edmonds-Karp)	$O(VE^2)$
Hopcroft-Karp (Bipartite Matching)	$O(\sqrt{V} * E)$
Dijkstra's Algorithm	$O((V + E) \log V)$
Bellman-Ford Algorithm	$O(V * E)$
Floyd-Warshall Algorithm	$O(V^3)$
Kosaraju's Algorithm (SCC)	$O(V + E)$
Topological Sort (DFS)	$O(V + E)$

18.Dynamic Programming

📌1 Definition

Dynamic Programming (DP) is a method used for solving complex problems by breaking them down into simpler subproblems. It is an optimization technique used to avoid redundant calculations by storing the results of subproblems and reusing them when needed.

DP is used for optimization problems where we need to find the best solution under given constraints, and where solutions to subproblems overlap.

📌2 Key Concepts

2.1 Optimal Substructure

A problem has **optimal substructure** if the optimal solution of the problem can be constructed efficiently from optimal solutions of its subproblems.

2.2 Overlapping Subproblems

A problem has **overlapping subproblems** if the problem can be broken down into subproblems that are solved multiple times.

📌3️⃣ DP Approaches

3.1 Top-Down Approach (Memoization)

- **Memoization** involves solving the problem recursively, but storing the results of subproblems to avoid repeated work.
- When a subproblem is encountered for the first time, it is solved and stored; if it's encountered again, the stored result is used.

Time Complexity: O(n) (depends on the number of unique subproblems)

Space Complexity: O(n) (for storing results)

Example (Fibonacci):

```
int fib(int n, int[] memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];
    memo[n] = fib(n-1, memo) + fib(n-2, memo);
    return memo[n];
}

int[] memo = new int[100];
Arrays.fill(memo, -1);
System.out.println(fib(10, memo));
```

3.2 Bottom-Up Approach (Tabulation)

- **Tabulation** involves solving the problem iteratively, starting from the base case and building up to the final solution.
- The solutions to all subproblems are stored in a table (array) in a bottom-up manner.

Time Complexity: O(n) (for iterating through the table)

Space Complexity: O(n) (for the table)

Example (Fibonacci):

```
int fib(int n) {
    int[] dp = new int[n+1];
    dp[0] = 0;
    dp[1] = 1;
    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
```

📌4️⃣Common DP Problems

4.1 Fibonacci Sequence

- **Problem**: Find the nth Fibonacci number.
- **Recursive Relation**: `F(n) = F(n-1) + F(n-2)`

4.2 Knapsack Problem

- **Problem**: Given weights and values of items, determine the maximum value that can be obtained by selecting items such that their total weight does not exceed the knapsack capacity.
- **Recursive Relation**:

```
dp[i][w] = max(dp[i-1][w], dp[i-1][w - weight[i]] + value[i])
```

Time Complexity: O(n * W), where `n` is the number of items, and `W` is the capacity of the knapsack.

4.3 Longest Common Subsequence (LCS)

- **Problem**: Given two sequences, find the length of the longest subsequence common to both sequences.
- **Recursive Relation**:

```
dp[i][j] =
  1 + dp[i-1][j-1] if sequence1[i] == sequence2[j]
  max(dp[i-1][j], dp[i][j-1]) otherwise
```

Time Complexity: O(m * n), where `m` and `n` are the lengths of the two sequences.

4.4 Longest Increasing Subsequence (LIS)

- **Problem**: Find the longest subsequence of a given sequence such that all elements are sorted in increasing order.
- **Recursive Relation**:

```
dp[i] = max(dp[i], dp[j] + 1) if arr[i] > arr[j]
```

Time Complexity: O(n²) (for the dynamic programming approach)

📌5️⃣ DP Patterns to Remember

5.1 0/1 Knapsack Pattern

This pattern is useful in problems where we are given a set of items and need to make a decision (select or reject) based on certain constraints (e.g., weight, value).

- **Recursive Relation**:

```
dp[i][w] = max(dp[i-1][w], dp[i-1][w - weight[i]] + value[i])
```

5.2 Unbounded Knapsack Pattern

Similar to 0/1 Knapsack but with the difference that each item can be taken any number of times (unbounded).

- **Recursive Relation**:

```
dp[i][w] = max(dp[i-1][w], dp[i][w - weight[i]] + value[i])
```

5.3 Longest Common Subsequence Pattern

Used in problems where we are asked to find common subsequences (e.g., LCS, Edit Distance).

- **Recursive Relation**:

```
dp[i][j] =
  1 + dp[i-1][j-1] if str1[i] == str2[j]
  max(dp[i-1][j], dp[i][j-1]) otherwise
```

📌6️⃣ Time Complexity of Common DP Problems

Problem	Time Complexity
Fibonacci Sequence	O(n)
Knapsack Problem	O(n * W)
Longest Common Subsequence (LCS)	O(m * n)
Longest Increasing Subsequence (LIS)	$O(n^2)$
Coin Change Problem	O(n * amount)

📌7️⃣When to Use Dynamic Programming

- When the problem involves **optimal substructure** and **overlapping subproblems**.
- When brute force recursion would result in **repetitive calculations**.
- DP is ideal for **optimization problems**, such as maximizing profit, minimizing cost, or counting distinct ways to achieve a result.

📌8️⃣Space Optimization in DP

- If only the current and previous states are needed, you can optimize space from **O(n * W)** to **O(W)** (or similar), which helps to reduce memory usage.

Example (1D DP Optimization):

```
int[] dp = new int[W + 1];  // 1D DP array for knapsack
```


19. 2-D Dynamic Programming

📌1️⃣Definition

2-D Dynamic Programming (DP) involves using a 2-dimensional table (usually a 2D array) to store the results of overlapping subproblems. This approach is useful for problems that involve two sets of parameters, often representing two dimensions of state.

2-D DP is typically used when the problem involves two varying parameters or indices. Common problems include:

- Finding the **Longest Common Subsequence** (LCS).
- **2-D Knapsack Problem**.
- **Matrix Chain Multiplication**.
- **Edit Distance** (Levenshtein Distance).
- **Word Break Problem**.

📌3️⃣Common 2-D DP Problems

3.1 Longest Common Subsequence (LCS)

- **Problem**: Given two sequences, find the length of the longest subsequence common to both sequences.

Recursive Relation:

```
dp[i][j] =
  dp[i-1][j-1] + 1 if str1[i-1] == str2[j-1]
  max(dp[i-1][j], dp[i][j-1]) if str1[i-1] != str2[j-1]
```

Time Complexity: O(m * n), where `m` and `n` are the lengths of the two sequences.

Example (LCS):

```java
public int lcs(String str1, String str2) {
    int m = str1.length();
    int n = str2.length();
    int[][] dp = new int[m + 1][n + 1];

    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (str1.charAt(i - 1) == str2.charAt(j - 1)) {
                dp[i][j] = dp[i - 1][j - 1] + 1;
            } else {
                dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }
    }

    return dp[m][n];
}
```

3.3 Matrix Chain Multiplication

- **Problem**: Given a sequence of matrices, find the most efficient way to multiply them together. The problem is to find the minimum number of scalar multiplications needed to multiply the chain.

Recursive Relation:

```
dp[i][j] =
```

```
min(dp[i][k] + dp[k+1][j] + (dimensions[i-1] * dimensions[k] * dimensions[j]))
```

Time Complexity: $O(n^3)$, where n is the number of matrices.

📌4️⃣ General 2-D DP Approach

4.1 Table Initialization

For most DP problems, the 2D table is initialized to zero or a base value (e.g., `dp[0][0] = 0`).

4.2 Recurrence Relation

The recurrence relation should be carefully constructed based on the problem, as it determines how the values in the DP table are updated.

4.3 Backtracking

Once the table is filled, the solution is typically found by backtracking from the last cell (`dp[m][n]` for LCS) to trace back the path to the optimal solution.

📌5️⃣ 2-D DP Patterns to Remember

5.1 Path Problems (e.g., Grid Problems)

Many problems involve finding the number of unique paths from the top-left to the bottom-right of a grid. These types of problems can be solved using 2-D DP.

Example (Unique Paths in a Grid):

```java
public int uniquePaths(int m, int n) {
    int[][] dp = new int[m][n];

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (i == 0 || j == 0) {
                dp[i][j] = 1;
            } else {
                dp[i][j] = dp[i-1][j] + dp[i][j-1];
            }
        }
    }

    return dp[m-1][n-1];
}
```

📌6️⃣ **Time Complexity of Common 2-D DP Problems**

Problem	Time Complexity
Longest Common Subsequence (LCS)	O(m * n)
Edit Distance (Levenshtein Distance)	O(m * n)
Matrix Chain Multiplication	$O(n^3)$
2-D Knapsack Problem	O(n * W)
Unique Paths in Grid	O(m * n)

📌7️⃣ Space Optimization in 2-D DP

You can optimize space by using a **1-D DP array** in some problems where only the current and previous rows are needed. This reduces space complexity from **O(m * n)** to **O(n)** or **O(m)**, depending on the problem.

20. Bit Manipulation

📌1️⃣ Definition

Bit manipulation involves operations that directly manipulate bits (0s and 1s) of binary numbers. Bitwise operations are used to perform tasks like checking bits, setting bits, toggling bits, and shifting bits. It is commonly used for optimization, low-level programming, and problem-solving in competitive programming.

📌2️⃣ **Common Bitwise Operators**

Operator	Description	Syntax
AND (&)	Bitwise AND: returns 1 if both bits are 1.	a & b
**OR (`)	`)**	Bitwise OR: returns 1 if at least one bit is 1.
XOR (^)	Bitwise XOR: returns 1 if bits are different.	a ^ b
NOT (~)	Bitwise NOT: inverts the bits.	~a
Left Shift (<<)	Shifts bits to the left by a given number of positions (multiplies by 2^n).	a << n
Right Shift (>>)	Shifts bits to the right by a given number of positions (divides by 2^n).	a >> n

📌3️⃣ Key Operations & Concepts

3.1 Checking if a Number is Odd or Even

To check if a number is odd or even, use the **AND** operator (&) with 1.

- **Even**: `n & 1 == 0`
- **Odd**: `n & 1 == 1`

📌6️⃣ Time Complexity of Common Bit Manipulation Operations

Operation	Time Complexity
AND (&)	O(1)
**OR (`	`)**
XOR (^)	O(1)
NOT (~)	O(1)
Left Shift (<<)	O(1)
Right Shift (>>)	O(1)
Counting Set Bits	O(number of bits)
Checking Power of Two	O(1)

NEVER LOOK FOR THE SOLUTION, ALWAYS THINK FOR THE SOLUTION! - NISHCHAL
FOLLOW ME ON IG: https://www.instagram.com/codewithnishchal/