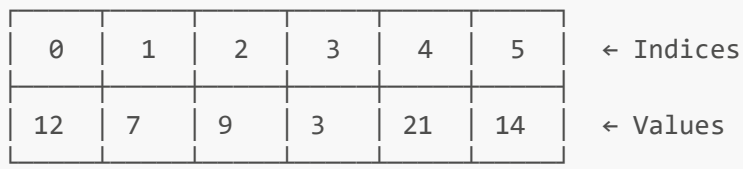
Data Structures and Algorithms

Table of Contents

Arrays

Conceptual Explanation

Arrays are the most fundamental data structure, consisting of elements stored in contiguous memory locations. In Java, arrays have a fixed size once created.



Java Implementation

```
// Declaration and initialization
int[] array = new int[5]; // Creates an array of size 5
int[] populatedArray = {1, 2, 3, 4, 5}; // Creates and initializes

// Accessing elements - O(1)
int value = array[2]; // Access 3rd element (index 2)
```

```
// Modifying elements - O(1)
array[2] = 10; // Sets 3rd element to 10

// Dynamic arrays using ArrayList
import java.util.ArrayList;

ArrayList<Integer> dynamicArray = new ArrayList<>();
dynamicArray.add(1); // Adds element at the end
dynamicArray.get(0); // Access element
dynamicArray.set(0, 10); // Modify element
dynamicArray.remove(0); // Remove element
dynamicArray.size(); // Get size
```

Time and Space Complexity

- **Access**: O(1) - Constant time access by index
- **Search**: O(n) - Linear search in unsorted array, O(log n) in sorted array
- **Insertion/Deletion**:
  - At the end (ArrayList with space): O(1) amortized
  - At a specific position: O(n) due to shifting elements
- **Space**: O(n) where n is the size of the array

Common Variations and Applications

1. **Dynamic Arrays** (ArrayList in Java): Automatically resize
2. **Multi-dimensional Arrays**: Matrix representations
3. **Jagged Arrays**: Arrays of arrays with different lengths
4. **Circular Arrays**: Used for efficient queue implementation

Typical Interview Questions

1. Two Sum

Problem: Find two numbers in an array that add up to a target value.

Solution:

```
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();

    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[] { map.get(complement), i };
        }
        map.put(nums[i], i);
    }

    return new int[] {};
}
```

Time Complexity: O(n) **Space Complexity**: O(n)

2. Maximum Subarray

Problem: Find the contiguous subarray with the largest sum.

Solution (Kadane's Algorithm):

```
public int maxSubArray(int[] nums) {
    int maxSoFar = nums[0];
    int maxEndingHere = nums[0];

    for (int i = 1; i < nums.length; i++) {
        // Either extend the previous subarray or start a new one
        maxEndingHere = Math.max(nums[i], maxEndingHere + nums[i]);
        maxSoFar = Math.max(maxSoFar, maxEndingHere);
    }

    return maxSoFar;
}
```

Time Complexity: O(n) **Space Complexity**: O(1)

3. Container With Most Water

Problem: Find two lines that together with the x-axis form a container that holds the most water.

Solution:

```
public int maxArea(int[] height) {
    int maxWater = 0;
    int left = 0;
    int right = height.length - 1;

    while (left < right) {
        // Calculate water area
        int width = right - left;
        int h = Math.min(height[left], height[right]);
        maxWater = Math.max(maxWater, width * h);

        // Move pointers
        if (height[left] < height[right]) {
            left++;
        } else {
            right--;
        }
    }
```

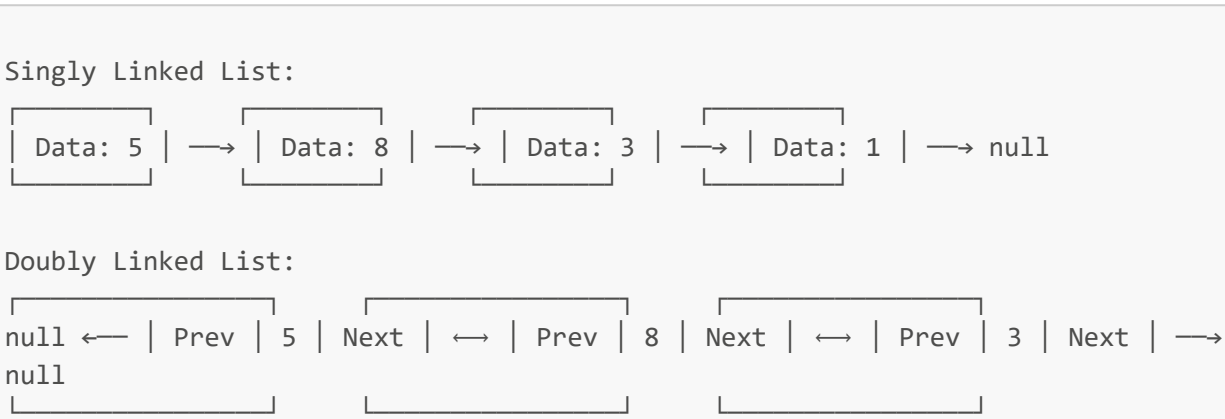
```
        return maxWater;
    }
```

Time Complexity: O(n) **Space Complexity**: O(1)

Linked Lists

Conceptual Explanation

Linked lists are linear data structures where elements (nodes) are not stored in contiguous memory locations. Each node contains data and a reference to the next node.



Java Implementation

```
// Singly Linked List
class ListNode {
    int val;
    ListNode next;

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

class SinglyLinkedList {
    ListNode head;

    // Insert at beginning - O(1)
    public void insertAtBeginning(int value) {
        ListNode newNode = new ListNode(value);
        newNode.next = head;
        head = newNode;
    }

    // Insert at end - O(n)
    public void insertAtEnd(int value) {
        ListNode newNode = new ListNode(value);
```

```
        if (head == null) {
            head = newNode;
            return;
        }

        ListNode current = head;
        while (current.next != null) {
            current = current.next;
        }

        current.next = newNode;
    }

    // Delete a node with given value - O(n)
    public void delete(int value) {
        if (head == null) return;

        if (head.val == value) {
            head = head.next;
            return;
        }

        ListNode current = head;
        while (current.next != null && current.next.val != value) {
            current = current.next;
        }

        if (current.next != null) {
            current.next = current.next.next;
        }
    }

    // Search for a value - O(n)
    public boolean search(int value) {
        ListNode current = head;

        while (current != null) {
            if (current.val == value) {
                return true;
            }
            current = current.next;
        }

        return false;
    }
}

// Doubly Linked List Node
class DoublyListNode {
    int val;
    DoublyListNode prev;
    DoublyListNode next;
```

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

Time and Space Complexity

- **Access**: O(n) - Need to traverse the list
- **Search**: O(n) - Need to traverse the list
- **Insertion**:
  - At beginning: O(1)
  - At end: O(n) for singly linked list, O(1) if we maintain a tail pointer
  - At specific position: O(n) to find the position, O(1) to insert
- **Deletion**:
  - At beginning: O(1)
  - At end: O(n) for singly linked list, O(1) for doubly linked list with tail pointer
  - At specific position: O(n) to find the position, O(1) to delete
- **Space**: O(n) where n is the number of nodes

Common Variations and Applications

1. **Singly Linked List**: Each node points to the next node
2. **Doubly Linked List**: Each node points to both next and previous nodes
3. **Circular Linked List**: Last node points back to the first node
4. **Skip List**: Multiple layers of linked lists for faster searching

Typical Interview Questions

1. Reverse a Linked List

Problem: Reverse a singly linked list.

Solution:

```
public ListNode reverseList(ListNode head) {
    ListNode prev = null;
    ListNode current = head;

    while (current != null) {
        ListNode nextTemp = current.next;
        current.next = prev;
        prev = current;
        current = nextTemp;
    }

    return prev;
}
```

Time Complexity: O(n) **Space Complexity**: O(1)

2. Detect Cycle in a Linked List

Problem: Determine if a linked list has a cycle.

Solution:

```
public boolean hasCycle(ListNode head) {
    if (head == null || head.next == null) {
        return false;
    }

    ListNode slow = head;
    ListNode fast = head;

    while (fast != null && fast.next != null) {
        slow = slow.next;          // Move one step
        fast = fast.next.next;     // Move two steps

        if (slow == fast) {
            return true;           // Cycle detected
        }
    }

    return false;                  // No cycle
}
```

Time Complexity: O(n) **Space Complexity**: O(1)

3. Merge Two Sorted Lists

Problem: Merge two sorted linked lists into one sorted linked list.

Solution:

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    // Create a dummy head node
    ListNode dummy = new ListNode(0);
    ListNode tail = dummy;

    while (l1 != null && l2 != null) {
        if (l1.val < l2.val) {
            tail.next = l1;
            l1 = l1.next;
        } else {
            tail.next = l2;
            l2 = l2.next;
        }
```

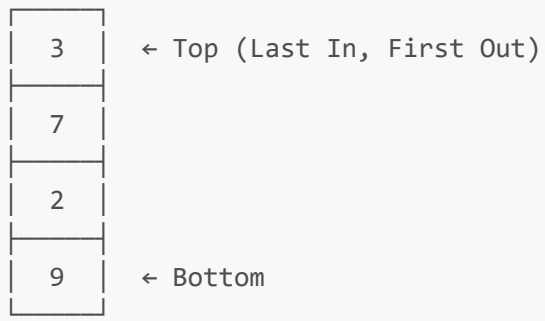
```
            tail = tail.next;
        }

        // Attach remaining nodes
        if (l1 != null) {
            tail.next = l1;
        } else {
            tail.next = l2;
        }

        return dummy.next;
    }
```

Time Complexity: O(n + m) where n and m are lengths of the lists **Space Complexity**: O(1)

Stacks

Conceptual Explanation

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. Elements can only be inserted or removed from the top of the stack.



Java Implementation

```
// Using built-in Stack class
import java.util.Stack;

Stack<Integer> stack = new Stack<>();
stack.push(5);    // Add to top
stack.pop();      // Remove from top
stack.peek();     // View top without removing
stack.isEmpty();  // Check if empty
stack.size();     // Get size

// Custom stack implementation using an array
class ArrayStack {
    private int[] array;
    private int top;
    private int capacity;
```

```
    public ArrayStack(int capacity) {
        this.array = new int[capacity];
        this.top = -1;
        this.capacity = capacity;
    }

    // Add element to the top of stack
    public void push(int value) {
        if (isFull()) {
            throw new RuntimeException("Stack is full");
        }
        array[++top] = value;
    }

    // Remove element from top of stack
    public int pop() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        return array[top--];
    }

    // Return top element without removing
    public int peek() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        return array[top];
    }

    public boolean isEmpty() {
        return top == -1;
    }

    public boolean isFull() {
        return top == capacity - 1;
    }

    public int size() {
        return top + 1;
    }
}

// Stack using a linked list
class LinkedListStack {
    private class Node {
        int data;
        Node next;

        Node(int data) {
            this.data = data;
        }
    }
```

```
    private Node top;
    private int size;

    public void push(int value) {
        Node newNode = new Node(value);
        newNode.next = top;
        top = newNode;
        size++;
    }

    public int pop() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        int data = top.data;
        top = top.next;
        size--;
        return data;
    }

    public int peek() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        return top.data;
    }

    public boolean isEmpty() {
        return top == null;
    }

    public int size() {
        return size;
    }
}
```

Time and Space Complexity

- **Push**: O(1)
- **Pop**: O(1)
- **Peek**: O(1)
- **Search**: O(n) - Need to pop elements to search
- **Space**: O(n) where n is the number of elements

Common Variations and Applications

1. **Expression Evaluation**: Evaluating arithmetic expressions
2. **Balancing Symbols**: Checking for balanced parentheses
3. **Function Call Stack**: Managing function calls in programming
4. **Undo Mechanism**: Implementing undo functionality

5. **DFS Implementation**: Implementing depth-first search for graphs

Typical Interview Questions

1. Valid Parentheses

Problem: Determine if a string of parentheses is valid.

Solution:

```
public boolean isValid(String s) {
    Stack<Character> stack = new Stack<>();

    for (char c : s.toCharArray()) {
        if (c == '(' || c == '[' || c == '{') {
            stack.push(c);
        } else {
            if (stack.isEmpty()) {
                return false;
            }

            char top = stack.pop();
            if ((c == ')' && top != '(') ||
                (c == ']' && top != '[') ||
                (c == '}' && top != '{')) {
                return false;
            }
        }
    }

    return stack.isEmpty();
}
```

Time Complexity: O(n) **Space Complexity**: O(n)

2. Evaluate Reverse Polish Notation

Problem: Evaluate an expression in Reverse Polish Notation (postfix).

Solution:

```
public int evalRPN(String[] tokens) {
    Stack<Integer> stack = new Stack<>();

    for (String token : tokens) {
        if (token.equals("+") || token.equals("-") ||
            token.equals("*") || token.equals("/")) {
            int b = stack.pop();
            int a = stack.pop();
```

```
            switch (token) {
                case "+":
                    stack.push(a + b);
                    break;
                case "-":
                    stack.push(a - b);
                    break;
                case "*":
                    stack.push(a * b);
                    break;
                case "/":
                    stack.push(a / b);
                    break;
            }
        } else {
            stack.push(Integer.parseInt(token));
        }
    }

    return stack.pop();
}
```

Time Complexity: O(n) **Space Complexity**: O(n)

3. Min Stack

Problem: Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Solution:

```
class MinStack {
    private Stack<Integer> stack;
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<>();
        minStack = new Stack<>();
    }

    public void push(int val) {
        stack.push(val);

        // Update minimum
        if (minStack.isEmpty() || val <= minStack.peek()) {
            minStack.push(val);
        }
    }

    public void pop() {
        if (stack.isEmpty()) {
            return;
```

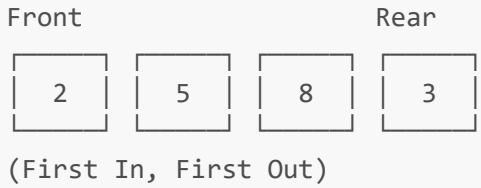
```
        }

        // If popped element is the minimum, remove from minStack too
        if (stack.peek().equals(minStack.peek())) {
            minStack.pop();
        }

        stack.pop();
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return minStack.peek();
    }
}
```

Time Complexity: O(1) for all operations **Space Complexity**: O(n)

Queues

Conceptual Explanation

A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. Elements are inserted at the rear (enqueue) and removed from the front (dequeue).



Java Implementation

```
// Using built-in Queue interface
import java.util.LinkedList;
import java.util.Queue;

Queue<Integer> queue = new LinkedList<>();
queue.add(5);      // Add to rear
queue.remove();    // Remove from front
queue.peek();      // View front without removing
queue.isEmpty();   // Check if empty
queue.size();      // Get size

// Custom queue implementation using an array
class ArrayQueue {
```

```java
    private int[] array;
    private int front;
    private int rear;
    private int capacity;
    private int size;

    public ArrayQueue(int capacity) {
        this.array = new int[capacity];
        this.front = 0;
        this.rear = -1;
        this.capacity = capacity;
        this.size = 0;
    }

    // Add element to the rear of queue
    public void enqueue(int value) {
        if (isFull()) {
            throw new RuntimeException("Queue is full");
        }
        rear = (rear + 1) % capacity;
        array[rear] = value;
        size++;
    }

    // Remove element from front of queue
    public int dequeue() {
        if (isEmpty()) {
            throw new RuntimeException("Queue is empty");
        }
        int data = array[front];
        front = (front + 1) % capacity;
        size--;
        return data;
    }

    // Return front element without removing
    public int peek() {
        if (isEmpty()) {
            throw new RuntimeException("Queue is empty");
        }
        return array[front];
    }

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

    public boolean isFull() {
        return size == capacity;
    }

    public int size() {
        return size;
    }
```

```
}

// Priority Queue
import java.util.PriorityQueue;

// Min Priority Queue (default)
PriorityQueue<Integer> minPQ = new PriorityQueue<>();
minPQ.add(5);       // Add element
minPQ.poll();       // Remove and return smallest element
minPQ.peek();       // View smallest element without removing

// Max Priority Queue (using custom comparator)
PriorityQueue<Integer> maxPQ = new PriorityQueue<>((a, b) -> b - a);
maxPQ.add(5);       // Add element
maxPQ.poll();       // Remove and return largest element
maxPQ.peek();       // View largest element without removing
```

Time and Space Complexity

- **Enqueue (add)**: O(1) for basic queue, O(log n) for priority queue
- **Dequeue (remove)**: O(1) for basic queue, O(log n) for priority queue
- **Peek**: O(1)
- **Search**: O(n) - Need to dequeue elements to search
- **Space**: O(n) where n is the number of elements

Common Variations and Applications

1. **Circular Queue**: Efficiently utilizes memory by reusing spaces
2. **Priority Queue**: Elements are served based on priority, not arrival time
3. **Deque (Double-Ended Queue)**: Can insert and remove from both ends
4. **BFS Implementation**: Implementing breadth-first search for graphs
5. **Task Scheduling**: Managing task execution order

Typical Interview Questions

1. Implement a Queue using Stacks

Problem: Implement a queue using only stacks.

Solution:

```
class MyQueue {
    private Stack<Integer> stack1; // For pushing elements
    private Stack<Integer> stack2; // For popping elements

    public MyQueue() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }
```

```
    // Push element x to the back of queue
    public void push(int x) {
        stack1.push(x);
    }

    // Remove the element from the front of queue and return it
    public int pop() {
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }

    // Get the front element
    public int peek() {
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.peek();
    }

    // Return whether the queue is empty
    public boolean empty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }
}
```

Time Complexity:

- Push: O(1)
- Pop: Amortized O(1) (worst case O(n))
- Peek: Amortized O(1) (worst case O(n)) **Space Complexity**: O(n)

2. Sliding Window Maximum

Problem: Find the maximum element in each sliding window of size k.

Solution:

```
public int[] maxSlidingWindow(int[] nums, int k) {
    if (nums == null || nums.length == 0) {
        return new int[0];
    }

    int n = nums.length;
    int[] result = new int[n - k + 1];
```

```
    // Use a deque to store indices of potential maximum elements
    Deque<Integer> deque = new ArrayDeque<>();

    for (int i = 0; i < n; i++) {
        // Remove elements outside the current window
        while (!deque.isEmpty() && deque.peekFirst() < i - k + 1) {
            deque.pollFirst();
        }

        // Remove smaller elements as they won't be the maximum
        while (!deque.isEmpty() && nums[deque.peekLast()] < nums[i]) {
            deque.pollLast();
        }

        // Add current element's index
        deque.offerLast(i);

        // Add maximum to result if we have a complete window
        if (i >= k - 1) {
            result[i - k + 1] = nums[deque.peekFirst()];
        }
    }

    return result;
}
```

Time Complexity: O(n) **Space Complexity**: O(k)

3. Task Scheduler

Problem: Schedule tasks with cooling intervals to minimize completion time.

Solution:

```
public int leastInterval(char[] tasks, int n) {
    // Count frequency of each task
    int[] frequencies = new int[26];
    for (char task : tasks) {
        frequencies[task - 'A']++;
    }

    // Sort frequencies in ascending order
    Arrays.sort(frequencies);

    // Get the maximum frequency
    int maxFreq = frequencies[25];

    // Calculate idle slots needed: (maxFreq - 1) * n
    int idleSlots = (maxFreq - 1) * n;

    // Fill idle slots with other tasks
```

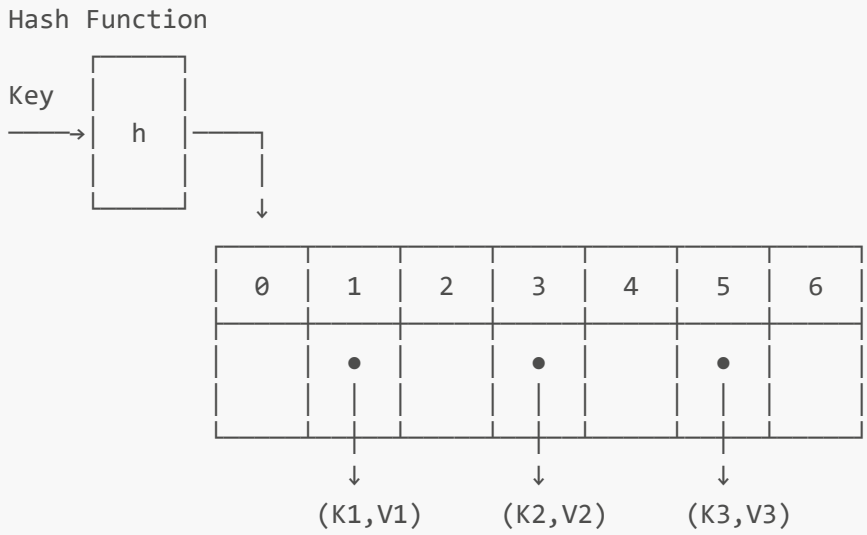
```
        for (int i = 24; i >= 0 && frequencies[i] > 0; i--) {
            // Min between maxFreq-1 and actual frequency
            idleSlots -= Math.min(maxFreq - 1, frequencies[i]);
        }

        // If we have enough tasks to fill all idle slots
        idleSlots = Math.max(0, idleSlots);

        return tasks.length + idleSlots;
    }
```

Time Complexity: O(n) **Space Complexity**: O(1)

Hash Tables

Conceptual Explanation

A hash table (or hash map) is a data structure that implements an associative array abstract data type, mapping keys to values. It uses a hash function to compute an index into an array of buckets, from which the desired value can be found.



Java Implementation

```
// Using built-in HashMap
import java.util.HashMap;

HashMap<String, Integer> map = new HashMap<>();
map.put("key", 42);           // Add key-value pair
map.get("key");               // Retrieve value
map.containsKey("key");       // Check if key exists
map.remove("key");            // Remove key-value pair
map.size();                   // Get number of entries
map.keySet();                 // Get set of keys
map.values();                 // Get collection of values
```

```
map.entrySet();               // Get set of key-value pairs

// Custom hash table implementation
class SimpleHashTable<K, V> {
    private static class Entry<K, V> {
        K key;
        V value;
        Entry<K, V> next;

        Entry(K key, V value) {
            this.key = key;
            this.value = value;
        }
    }

    private static final int DEFAULT_CAPACITY = 16;
    private Entry<K, V>[] buckets;
    private int size;

    @SuppressWarnings("unchecked")
    public SimpleHashTable() {
        buckets = new Entry[DEFAULT_CAPACITY];
        size = 0;
    }

    // Hash function to determine bucket index
    private int hash(K key) {
        return key == null ? 0 : Math.abs(key.hashCode() % buckets.length);
    }

    // Add or update key-value pair
    public void put(K key, V value) {
        int index = hash(key);

        // If bucket is empty, create new entry
        if (buckets[index] == null) {
            buckets[index] = new Entry<>(key, value);
            size++;
            return;
        }

        // Check if key already exists in the bucket
        Entry<K, V> current = buckets[index];
        while (current != null) {
            if ((key == null && current.key == null) ||
                (key != null && key.equals(current.key))) {
                current.value = value;  // Update existing key
                return;
            }
            if (current.next == null) {
                break;
            }
            current = current.next;
        }
```

```
        // Key does not exist, add as new entry
        current.next = new Entry<>(key, value);
        size++;
    }

    // Retrieve value for given key
    public V get(K key) {
        int index = hash(key);

        Entry<K, V> current = buckets[index];
        while (current != null) {
            if ((key == null && current.key == null) ||
                (key != null && key.equals(current.key))) {
                return current.value;
            }
            current = current.next;
        }

        return null;  // Key not found
    }

    // Check if key exists
    public boolean containsKey(K key) {
        int index = hash(key);

        Entry<K, V> current = buckets[index];
        while (current != null) {
            if ((key == null && current.key == null) ||
                (key != null && key.equals(current.key))) {
                return true;
            }
            current = current.next;
        }

        return false;
    }

    // Remove key-value pair
    public V remove(K key) {
        int index = hash(key);

        if (buckets[index] == null) {
            return null;
        }

        // If key is at the head of the bucket
        if ((key == null && buckets[index].key == null) ||
            (key != null && key.equals(buckets[index].key))) {
            V value = buckets[index].value;
            buckets[index] = buckets[index].next;
            size--;
            return value;
        }
```

```
        // Search for key in the chain
        Entry<K, V> current = buckets[index];
        while (current.next != null) {
            if ((key == null && current.next.key == null) ||
                (key != null && key.equals(current.next.key))) {
                V value = current.next.value;
                current.next = current.next.next;
                size--;
                return value;
            }
            current = current.next;
        }

        return null;  // Key not found
    }

    public int size() {
        return size;
    }

    public boolean isEmpty() {
        return size == 0;
    }
}
```

Time and Space Complexity

- **Insert (put)**: O(1) average case, O(n) worst case
- **Lookup (get)**: O(1) average case, O(n) worst case
- **Delete (remove)**: O(1) average case, O(n) worst case
- **Space**: O(n) where n is the number of key-value pairs

Common Variations and Applications

1. **HashMap**: General purpose key-value storage
2. **HashSet**: Collection of unique elements
3. **LinkedHashMap**: Maintains insertion order
4. **ConcurrentHashMap**: Thread-safe for concurrent access
5. **Used for caching, indexing, and deduplication**

Typical Interview Questions

1. Two Sum

Problem: Find two numbers in an array that add up to a target value.

Solution:

```
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();

    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[] { map.get(complement), i };
        }
        map.put(nums[i], i);
    }

    return new int[] {};
}
```

Time Complexity: O(n) **Space Complexity**: O(n)

2. Group Anagrams

Problem: Group strings that are anagrams of each other.

Solution:

```
public List<List<String>> groupAnagrams(String[] strs) {
    Map<String, List<String>> map = new HashMap<>();

    for (String str : strs) {
        // Convert string to char array and sort
        char[] chars = str.toCharArray();
        Arrays.sort(chars);
        String key = new String(chars);

        // Add string to the group of its sorted form
        map.putIfAbsent(key, new ArrayList<>());
        map.get(key).add(str);
    }

    return new ArrayList<>(map.values());
}
```

Time Complexity: O(n _ k log k) where n is the number of strings and k is the max length of a string **Space Complexity**: O(n _ k)

3. LRU Cache

Problem: Implement an LRU (Least Recently Used) cache with get and put operations in O(1) time.

Solution:

```
class LRUCache {
    private class Node {
        int key;
        int value;
        Node prev;
        Node next;

        Node(int key, int value) {
            this.key = key;
            this.value = value;
        }
    }

    private Map<Integer, Node> cache;
    private int capacity;
    private Node head;  // Most recently used
    private Node tail;  // Least recently used

    public LRUCache(int capacity) {
        this.capacity = capacity;
        this.cache = new HashMap<>();
        this.head = new Node(0, 0);
        this.tail = new Node(0, 0);
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        if (!cache.containsKey(key)) {
            return -1;
        }

        // Move node to head (most recently used)
        Node node = cache.get(key);
        removeNode(node);
        addToHead(node);

        return node.value;
    }

    public void put(int key, int value) {
        // Update existing key
        if (cache.containsKey(key)) {
            Node node = cache.get(key);
            node.value = value;
            removeNode(node);
            addToHead(node);
            return;
        }

        // Add new key
        Node newNode = new Node(key, value);
        cache.put(key, newNode);
```

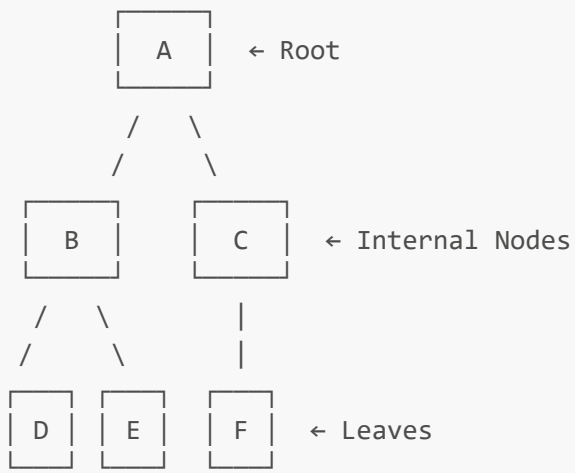
```
            addToHead(newNode);

            // Remove LRU if capacity is exceeded
            if (cache.size() > capacity) {
                Node lru = tail.prev;
                removeNode(lru);
                cache.remove(lru.key);
            }
        }

        private void removeNode(Node node) {
            node.prev.next = node.next;
            node.next.prev = node.prev;
        }

        private void addToHead(Node node) {
            node.next = head.next;
            node.prev = head;
            head.next.prev = node;
            head.next = node;
        }
    }
```

Time Complexity: O(1) for both get and put operations **Space Complexity**: O(capacity)

Trees

Conceptual Explanation

A tree is a hierarchical data structure consisting of nodes, where each node has a value and references to child nodes. The topmost node is called the root, and nodes with no children are called leaves.



Java Implementation

```java
// Basic tree node
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int val) {
        this.val = val;
    }
}

// Binary Search Tree implementation
class BinarySearchTree {
    private TreeNode root;

    // Insert value into the BST
    public void insert(int value) {
        root = insertRecursive(root, value);
    }

    private TreeNode insertRecursive(TreeNode root, int value) {
        if (root == null) {
            return new TreeNode(value);
        }

        if (value < root.val) {
            root.left = insertRecursive(root.left, value);
        } else if (value > root.val) {
            root.right = insertRecursive(root.right, value);
        }

        return root;
    }

    // Search for a value in the BST
    public boolean search(int value) {
        return searchRecursive(root, value);
    }

    private boolean searchRecursive(TreeNode root, int value) {
        if (root == null) {
            return false;
        }

        if (root.val == value) {
            return true;
        }

        if (value < root.val) {
            return searchRecursive(root.left, value);
        } else {
            return searchRecursive(root.right, value);
        }
```

```
    }

    // Delete a value from the BST
    public void delete(int value) {
        root = deleteRecursive(root, value);
    }

    private TreeNode deleteRecursive(TreeNode root, int value) {
        if (root == null) {
            return null;
        }

        if (value < root.val) {
            root.left = deleteRecursive(root.left, value);
        } else if (value > root.val) {
            root.right = deleteRecursive(root.right, value);
        } else {
            // Node with only one child or no child
            if (root.left == null) {
                return root.right;
            } else if (root.right == null) {
                return root.left;
            }

            // Node with two children
            // Get inorder successor (smallest in right subtree)
            root.val = minValue(root.right);

            // Delete the inorder successor
            root.right = deleteRecursive(root.right, root.val);
        }

        return root;
    }

    private int minValue(TreeNode root) {
        int minValue = root.val;
        while (root.left != null) {
            minValue = root.left.val;
            root = root.left;
        }
        return minValue;
    }

    // Tree traversals
    public void inorderTraversal() {
        inorderTraversalRecursive(root);
        System.out.println();
    }

    private void inorderTraversalRecursive(TreeNode root) {
        if (root != null) {
            inorderTraversalRecursive(root.left);
            System.out.print(root.val + " ");
```

```
            inorderTraversalRecursive(root.right);
        }
    }

    public void preorderTraversal() {
        preorderTraversalRecursive(root);
        System.out.println();
    }

    private void preorderTraversalRecursive(TreeNode root) {
        if (root != null) {
            System.out.print(root.val + " ");
            preorderTraversalRecursive(root.left);
            preorderTraversalRecursive(root.right);
        }
    }

    public void postorderTraversal() {
        postorderTraversalRecursive(root);
        System.out.println();
    }

    private void postorderTraversalRecursive(TreeNode root) {
        if (root != null) {
            postorderTraversalRecursive(root.left);
            postorderTraversalRecursive(root.right);
            System.out.print(root.val + " ");
        }
    }

    // Level order traversal (BFS)
    public void levelOrderTraversal() {
        if (root == null) {
            return;
        }

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

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

            if (node.left != null) {
                queue.offer(node.left);
            }

            if (node.right != null) {
                queue.offer(node.right);
            }
        }
        System.out.println();
    }
}
```

```java
// AVL Tree Node
class AVLNode {
    int val;
    AVLNode left;
    AVLNode right;
    int height;

    AVLNode(int val) {
        this.val = val;
        this.height = 1;
    }
}

// AVL Tree implementation (self-balancing BST)
class AVLTree {
    private AVLNode root;

    // Get height of a node
    private int height(AVLNode node) {
        if (node == null) {
            return 0;
        }
        return node.height;
    }

    // Get balance factor
    private int getBalance(AVLNode node) {
        if (node == null) {
            return 0;
        }
        return height(node.left) - height(node.right);
    }

    // Right rotation
    private AVLNode rightRotate(AVLNode y) {
        AVLNode x = y.left;
        AVLNode T2 = x.right;

        // Rotation
        x.right = y;
        y.left = T2;

        // Update heights
        y.height = Math.max(height(y.left), height(y.right)) + 1;
        x.height = Math.max(height(x.left), height(x.right)) + 1;

        return x;
    }

    // Left rotation
    private AVLNode leftRotate(AVLNode x) {
        AVLNode y = x.right;
        AVLNode T2 = y.left;
```

```
        // Rotation
        y.left = x;
        x.right = T2;

        // Update heights
        x.height = Math.max(height(x.left), height(x.right)) + 1;
        y.height = Math.max(height(y.left), height(y.right)) + 1;

        return y;
    }

    // Insert node
    public void insert(int value) {
        root = insertRecursive(root, value);
    }

    private AVLNode insertRecursive(AVLNode node, int value) {
        // Perform standard BST insert
        if (node == null) {
            return new AVLNode(value);
        }

        if (value < node.val) {
            node.left = insertRecursive(node.left, value);
        } else if (value > node.val) {
            node.right = insertRecursive(node.right, value);
        } else {
            // Duplicate values not allowed
            return node;
        }

        // Update height of this node
        node.height = Math.max(height(node.left), height(node.right)) + 1;

        // Get balance factor
        int balance = getBalance(node);

        // Left Left Case
        if (balance > 1 && value < node.left.val) {
            return rightRotate(node);
        }

        // Right Right Case
        if (balance < -1 && value > node.right.val) {
            return leftRotate(node);
        }

        // Left Right Case
        if (balance > 1 && value > node.left.val) {
            node.left = leftRotate(node.left);
            return rightRotate(node);
        }
```

```
        // Right Left Case
        if (balance < -1 && value < node.right.val) {
            node.right = rightRotate(node.right);
            return leftRotate(node);
        }

        return node;
    }
}
```

Time and Space Complexity

- **Binary Search Tree (balanced)**:
  - **Search**: O(log n)
  - **Insert**: O(log n)
  - **Delete**: O(log n)
- **Binary Search Tree (unbalanced/worst case)**:
  - **Search**: O(n)
  - **Insert**: O(n)
  - **Delete**: O(n)
- **AVL Tree / Red-Black Tree**:
  - **Search**: O(log n)
  - **Insert**: O(log n)
  - **Delete**: O(log n)
- **Space**: O(n) for all tree structures

Common Variations and Applications

1. **Binary Tree**: Each node has at most two children
2. **Binary Search Tree (BST)**: Left child < parent < right child
3. **AVL Tree**: Self-balancing BST using height balance
4. **Red-Black Tree**: Self-balancing BST using color properties
5. **B-Tree**: Generalization of BST that allows more than two children
6. **Heap**: Complete binary tree with heap property
7. **Trie**: Used for efficient string operations

Typical Interview Questions

1. Validate Binary Search Tree

Problem: Determine if a binary tree is a valid binary search tree.

Solution:

```
public boolean isValidBST(TreeNode root) {
    return isValidBSTHelper(root, null, null);
}
```

```
private boolean isValidBSTHelper(TreeNode node, Integer lower, Integer upper) {
    if (node == null) {
        return true;
    }

    int val = node.val;

    // Check if node's value violates lower bound
    if (lower != null && val <= lower) {
        return false;
    }

    // Check if node's value violates upper bound
    if (upper != null && val >= upper) {
        return false;
    }

    // Check right subtree with current value as lower bound
    if (!isValidBSTHelper(node.right, val, upper)) {
        return false;
    }

    // Check left subtree with current value as upper bound
    if (!isValidBSTHelper(node.left, lower, val)) {
        return false;
    }

    return true;
}
```

Time Complexity: O(n) **Space Complexity**: O(h) where h is the height of the tree

2. Lowest Common Ancestor in a Binary Tree

Problem: Find the lowest common ancestor of two nodes in a binary tree.

Solution:

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    // Base case
    if (root == null || root == p || root == q) {
        return root;
    }

    // Search in left and right subtrees
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);

    // If both nodes found in different subtrees, current node is LCA
    if (left != null && right != null) {
        return root;
```

```
        }

        // Otherwise, return the non-null subtree result
        return (left != null) ? left : right;
    }
```

Time Complexity: O(n) **Space Complexity**: O(h) where h is the height of the tree

3. Maximum Path Sum in Binary Tree

Problem: Find the maximum path sum in a binary tree.

Solution:

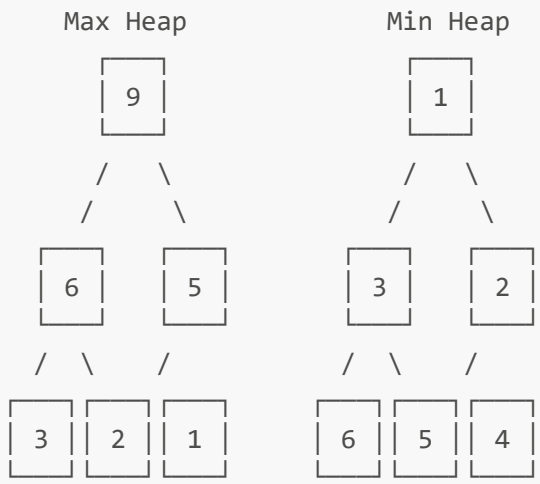
```
    public int maxPathSum(TreeNode root) {
        int[] maxSum = new int[1];
        maxSum[0] = Integer.MIN_VALUE;

        maxPathSumHelper(root, maxSum);

        return maxSum[0];
    }

    private int maxPathSumHelper(TreeNode node, int[] maxSum) {
        if (node == null) {
            return 0;
        }

        // Compute max path sum for left subtree
        int leftSum = Math.max(0, maxPathSumHelper(node.left, maxSum));

        // Compute max path sum for right subtree
        int rightSum = Math.max(0, maxPathSumHelper(node.right, maxSum));

        // Max path through current node
        int pathSum = node.val + leftSum + rightSum;

        // Update global maximum
        maxSum[0] = Math.max(maxSum[0], pathSum);

        // Return max path from this node to either left or right
        return node.val + Math.max(leftSum, rightSum);
    }
```

Time Complexity: O(n) **Space Complexity**: O(h) where h is the height of the tree

Heaps

Conceptual Explanation

A heap is a specialized tree-based data structure that satisfies the heap property. In a max heap, for any given node, the value of the node is greater than or equal to the values of its children. In a min heap, the value of the node is less than or equal to the values of its children.



Java Implementation

```java
// Using built-in PriorityQueue
import java.util.PriorityQueue;

// Min Heap (default)
PriorityQueue<Integer> minHeap = new PriorityQueue<>();
minHeap.add(5);         // Add element
minHeap.poll();         // Remove and return the minimum element
minHeap.peek();         // View the minimum element without removing
minHeap.size();         // Get number of elements

// Max Heap (using custom comparator)
PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);
maxHeap.add(5);         // Add element
maxHeap.poll();         // Remove and return the maximum element
maxHeap.peek();         // View the maximum element without removing

// Custom MinHeap implementation
class MinHeap {
    private int[] heap;
    private int size;
    private int capacity;

    public MinHeap(int capacity) {
        this.capacity = capacity;
        this.size = 0;
        this.heap = new int[capacity];
    }

    // Get parent index
    private int parent(int i) {
```

```
        return (i - 1) / 2;
    }

    // Get left child index
    private int leftChild(int i) {
        return 2 * i + 1;
    }

    // Get right child index
    private int rightChild(int i) {
        return 2 * i + 2;
    }

    // Swap two elements
    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }

    // Insert a new element
    public void insert(int value) {
        if (size == capacity) {
            throw new RuntimeException("Heap is full");
        }

        // Insert at the end
        heap[size] = value;
        size++;

        // Fix the min heap property
        int i = size - 1;
        while (i > 0 && heap[i] < heap[parent(i)]) {
            swap(i, parent(i));
            i = parent(i);
        }
    }

    // Heapify the subtree rooted at index i
    private void heapify(int i) {
        int smallest = i;
        int left = leftChild(i);
        int right = rightChild(i);

        if (left < size && heap[left] < heap[smallest]) {
            smallest = left;
        }

        if (right < size && heap[right] < heap[smallest]) {
            smallest = right;
        }

        if (smallest != i) {
            swap(i, smallest);
```

```
                heapify(smallest);
            }
        }

        // Extract the minimum element
        public int extractMin() {
            if (size <= 0) {
                throw new RuntimeException("Heap is empty");
            }

            if (size == 1) {
                size--;
                return heap[0];
            }

            int root = heap[0];
            heap[0] = heap[size - 1];
            size--;
            heapify(0);

            return root;
        }

        // Peek at the minimum element
        public int peek() {
            if (size <= 0) {
                throw new RuntimeException("Heap is empty");
            }
            return heap[0];
        }

        // Get size
        public int size() {
            return size;
        }

        // Check if empty
        public boolean isEmpty() {
            return size == 0;
        }
    }
```

Time and Space Complexity

- **Insert**: O(log n)
- **Extract Min/Max**: O(log n)
- **Peek (Get Min/Max)**: O(1)
- **Build Heap**: O(n)
- **Space**: O(n) where n is the number of elements

Common Variations and Applications

1. **Min Heap**: Root is the minimum element
2. **Max Heap**: Root is the maximum element
3. **Binomial Heap**: Collection of binomial trees
4. **Fibonacci Heap**: More efficient for decrease key operations
5. **Applications**: Priority queues, heap sort, graph algorithms, scheduling

Typical Interview Questions

1. Kth Largest Element in an Array

Problem: Find the kth largest element in an unsorted array.

Solution:

```
public int findKthLargest(int[] nums, int k) {
    // Use min heap of size k
    PriorityQueue<Integer> minHeap = new PriorityQueue<>();

    for (int num : nums) {
        minHeap.offer(num);

        // Keep only k largest elements
        if (minHeap.size() > k) {
            minHeap.poll();
        }
    }

    // The min element in the heap is the kth largest
    return minHeap.peek();
}
```

Time Complexity: O(n log k) **Space Complexity**: O(k)

2. Merge K Sorted Lists

Problem: Merge k sorted linked lists into one sorted list.

Solution:

```
public ListNode mergeKLists(ListNode[] lists) {
    if (lists == null || lists.length == 0) {
        return null;
    }

    // Min heap to store nodes
    PriorityQueue<ListNode> minHeap = new PriorityQueue<>((a, b) -> a.val -
b.val);

    // Add the first node from each list to heap
```

```
        for (ListNode list : lists) {
            if (list != null) {
                minHeap.offer(list);
            }
        }

        ListNode dummy = new ListNode(0);
        ListNode current = dummy;

        while (!minHeap.isEmpty()) {
            ListNode node = minHeap.poll();
            current.next = node;
            current = current.next;

            if (node.next != null) {
                minHeap.offer(node.next);
            }
        }

        return dummy.next;
    }
```

Time Complexity: O(N log k) where N is the total number of nodes and k is the number of lists **Space Complexity**: O(k)

3. Top K Frequent Elements

Problem: Find the k most frequent elements in an array.

Solution:

```
    public int[] topKFrequent(int[] nums, int k) {
        // Count frequency of each number
        Map<Integer, Integer> freqMap = new HashMap<>();
        for (int num : nums) {
            freqMap.put(num, freqMap.getOrDefault(num, 0) + 1);
        }

        // Min heap based on frequency
        PriorityQueue<int[]> minHeap = new PriorityQueue<>((a, b) -> a[1] - b[1]);

        // Process each entry
        for (Map.Entry<Integer, Integer> entry : freqMap.entrySet()) {
            int num = entry.getKey();
            int count = entry.getValue();

            minHeap.offer(new int[] {num, count});

            // Keep only k most frequent
            if (minHeap.size() > k) {
                minHeap.poll();
```

```
        }
    }

    // Build result array
    int[] result = new int[k];
    for (int i = k - 1; i >= 0; i--) {
        result[i] = minHeap.poll()[0];
    }

    return result;
}
```

Time Complexity: O(n log k) **Space Complexity**: O(n)

Graphs

Conceptual Explanation

A graph is a non-linear data structure consisting of vertices (or nodes) and edges that connect these vertices. Graphs can be directed (edges have direction) or undirected, and weighted (edges have values) or unweighted.

```
Undirected Graph          Directed Graph
    A --- B                  A → B
    |     |                  ↑   ↓
    |     |                  |   |
    C --- D                  C ← D
```

Java Implementation

```
// Graph representation using adjacency list
class Graph {
    private int V;  // Number of vertices
    private LinkedList<Integer>[] adjList;  // Adjacency list

    @SuppressWarnings("unchecked")
    public Graph(int v) {
        this.V = v;
        adjList = new LinkedList[v];
        for (int i = 0; i < v; i++) {
            adjList[i] = new LinkedList<>();
        }
    }

    // Add edge for undirected graph
    public void addEdge(int src, int dest) {
        adjList[src].add(dest);
        adjList[dest].add(src);  // For undirected graph
    }
```

```java
    // Add edge for directed graph
    public void addDirectedEdge(int src, int dest) {
        adjList[src].add(dest);
    }

    // Breadth-First Search
    public void bfs(int startVertex) {
        boolean[] visited = new boolean[V];
        Queue<Integer> queue = new LinkedList<>();

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

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

            for (int neighbor : adjList[vertex]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.offer(neighbor);
                }
            }
        }
        System.out.println();
    }

    // Depth-First Search
    public void dfs(int startVertex) {
        boolean[] visited = new boolean[V];
        dfsUtil(startVertex, visited);
        System.out.println();
    }

    private void dfsUtil(int vertex, boolean[] visited) {
        visited[vertex] = true;
        System.out.print(vertex + " ");

        for (int neighbor : adjList[vertex]) {
            if (!visited[neighbor]) {
                dfsUtil(neighbor, visited);
            }
        }
    }

    // Check if graph has a cycle (undirected)
    public boolean hasCycle() {
        boolean[] visited = new boolean[V];

        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                if (hasCycleUtil(i, visited, -1)) {
                    return true;
```

```
                }
            }
        }

        return false;
    }

    private boolean hasCycleUtil(int vertex, boolean[] visited, int parent) {
        visited[vertex] = true;

        for (int neighbor : adjList[vertex]) {
            if (!visited[neighbor]) {
                if (hasCycleUtil(neighbor, visited, vertex)) {
                    return true;
                }
            } else if (neighbor != parent) {
                return true;
            }
        }

        return false;
    }

    // Dijkstra's algorithm for shortest paths
    public void dijkstra(int start) {
        int[] dist = new int[V];
        boolean[] visited = new boolean[V];

        // Initialize distances
        for (int i = 0; i < V; i++) {
            dist[i] = Integer.MAX_VALUE;
        }
        dist[start] = 0;

        // Find shortest path for all vertices
        for (int count = 0; count < V - 1; count++) {
            // Find minimum distance vertex
            int u = minDistance(dist, visited);
            visited[u] = true;

            // Update distances of adjacent vertices
            for (int v = 0; v < V; v++) {
                if (!visited[v] && adjList[u].contains(v) &&
                    dist[u] != Integer.MAX_VALUE &&
                    dist[u] + 1 < dist[v]) {
                    dist[v] = dist[u] + 1;
                }
            }
        }

        // Print distances
        System.out.println("Vertex \t Distance from Source");
        for (int i = 0; i < V; i++) {
            System.out.println(i + " \t " + dist[i]);
```

```
        }
    }

    private int minDistance(int[] dist, boolean[] visited) {
        int min = Integer.MAX_VALUE;
        int minIndex = -1;

        for (int v = 0; v < V; v++) {
            if (!visited[v] && dist[v] <= min) {
                min = dist[v];
                minIndex = v;
            }
        }

        return minIndex;
    }

    // Topological Sort (for directed acyclic graphs)
    public void topologicalSort() {
        Stack<Integer> stack = new Stack<>();
        boolean[] visited = new boolean[V];

        for (int i = 0; i < V; i++) {
            if (!visited[i]) {
                topologicalSortUtil(i, visited, stack);
            }
        }

        // Print topological order
        while (!stack.isEmpty()) {
            System.out.print(stack.pop() + " ");
        }
        System.out.println();
    }

    private void topologicalSortUtil(int vertex, boolean[] visited, Stack<Integer>
stack) {
        visited[vertex] = true;

        for (int neighbor : adjList[vertex]) {
            if (!visited[neighbor]) {
                topologicalSortUtil(neighbor, visited, stack);
            }
        }

        stack.push(vertex);
    }

    // Weighted graph representation
    private class WeightedGraph {
        private int V;
        private List<List<Edge>> adjList;

        private class Edge {
```

```
            int dest;
            int weight;

            Edge(int dest, int weight) {
                this.dest = dest;
                this.weight = weight;
            }
        }

        public WeightedGraph(int v) {
            this.V = v;
            adjList = new ArrayList<>(v);
            for (int i = 0; i < v; i++) {
                adjList.add(new ArrayList<>());
            }
        }

        public void addEdge(int src, int dest, int weight) {
            adjList.get(src).add(new Edge(dest, weight));
        }
    }
}
```

Time and Space Complexity

- **Adjacency List**:
  - **Space**: O(V + E) where V is number of vertices and E is number of edges
  - **Add Edge**: O(1)
  - **Check if edge exists**: O(degree(V))
  - **Find all neighbors**: O(degree(V))
- **Adjacency Matrix**:
  - **Space**: $O(V^2)$
  - **Add Edge**: O(1)
  - **Check if edge exists**: O(1)
  - **Find all neighbors**: O(V)
- **Common Algorithms**:
  - **BFS**: O(V + E)
  - **DFS**: O(V + E)
  - **Dijkstra's**: $O(V^2)$ with array, O((V + E) log V) with priority queue
  - **Bellman-Ford**: O(V·E)
  - **Floyd-Warshall**: $O(V^3)$
  - **Topological Sort**: O(V + E)
  - **Minimum Spanning Tree (Kruskal's)**: O(E log E)
  - **Minimum Spanning Tree (Prim's)**: $O(V^2)$ with array, O(E log V) with priority queue

Common Variations and Applications

1. **Directed Graphs**: Edges have direction
2. **Undirected Graphs**: Edges have no direction

3. **Weighted Graphs**: Edges have values/weights
4. **Connected Graphs**: All vertices are connected
5. **Complete Graphs**: Every vertex is connected to every other vertex
6. **Bipartite Graphs**: Vertices can be divided into two disjoint sets
7. **DAG (Directed Acyclic Graph)**: Directed graph with no cycles
8. **Applications**: Social networks, web pages, maps, scheduling, recommendation systems

Typical Interview Questions

1. Number of Islands

Problem: Count the number of islands in a 2D grid.

Solution:

```
public int numIslands(char[][] grid) {
    if (grid == null || grid.length == 0) {
        return 0;
    }

    int rows = grid.length;
    int cols = grid[0].length;
    int count = 0;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (grid[i][j] == '1') {
                count++;
                dfs(grid, i, j, rows, cols);
            }
        }
    }

    return count;
}

private void dfs(char[][] grid, int i, int j, int rows, int cols) {
    if (i < 0 || i >= rows || j < 0 || j >= cols || grid[i][j] != '1') {
        return;
    }

    // Mark as visited
    grid[i][j] = '0';

    // Check all four directions
    dfs(grid, i + 1, j, rows, cols);
    dfs(grid, i - 1, j, rows, cols);
    dfs(grid, i, j + 1, rows, cols);
    dfs(grid, i, j - 1, rows, cols);
}
```

Time Complexity: O(m × n) where m is the number of rows and n is the number of columns **Space Complexity**: O(m × n) in worst case

2. Course Schedule

Problem: Determine if it's possible to finish all courses given prerequisites.

Solution:

```
public boolean canFinish(int numCourses, int[][] prerequisites) {
    // Create adjacency list
    List<List<Integer>> graph = new ArrayList<>();
    for (int i = 0; i < numCourses; i++) {
        graph.add(new ArrayList<>());
    }

    // Add edges
    for (int[] prereq : prerequisites) {
        graph.get(prereq[1]).add(prereq[0]);
    }

    // 0 = unvisited, 1 = visiting, 2 = visited
    int[] visited = new int[numCourses];

    // Check for cycles using DFS
    for (int i = 0; i < numCourses; i++) {
        if (visited[i] == 0) {
            if (hasCycle(graph, visited, i)) {
                return false;
            }
        }
    }

    return true;
}

private boolean hasCycle(List<List<Integer>> graph, int[] visited, int course) {
    visited[course] = 1;  // Mark as visiting

    for (int next : graph.get(course)) {
        if (visited[next] == 1) {
            return true;  // Cycle detected
        }

        if (visited[next] == 0) {
            if (hasCycle(graph, visited, next)) {
                return true;
            }
        }
    }

    visited[course] = 2;  // Mark as visited
```

```
        return false;
    }
```

Time Complexity: O(V + E) where V is number of courses and E is number of prerequisites **Space Complexity**: O(V + E)

3. Network Delay Time

Problem: Find the time it takes for all nodes to receive a signal.

Solution:

```
public int networkDelayTime(int[][] times, int n, int k) {
    // Create adjacency list
    List<List<int[]>> graph = new ArrayList<>();
    for (int i = 0; i <= n; i++) {
        graph.add(new ArrayList<>());
    }

    // Add edges
    for (int[] time : times) {
        int src = time[0];
        int dest = time[1];
        int weight = time[2];
        graph.get(src).add(new int[] {dest, weight});
    }

    // Distances array
    int[] distances = new int[n + 1];
    Arrays.fill(distances, Integer.MAX_VALUE);
    distances[k] = 0;

    // Priority queue for Dijkstra's algorithm
    PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> a[1] - b[1]);
    pq.offer(new int[] {k, 0});

    while (!pq.isEmpty()) {
        int[] current = pq.poll();
        int node = current[0];
        int distance = current[1];

        if (distance > distances[node]) {
            continue;
        }

        for (int[] neighbor : graph.get(node)) {
            int nextNode = neighbor[0];
            int weight = neighbor[1];

            if (distances[node] + weight < distances[nextNode]) {
                distances[nextNode] = distances[node] + weight;
```

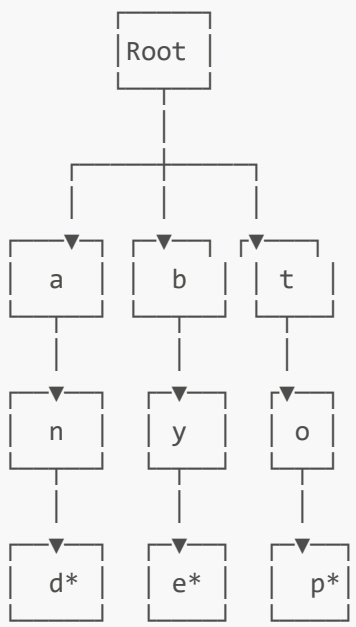
```
                    pq.offer(new int[] {nextNode, distances[nextNode]});
                }
            }
        }

        // Find maximum distance (time)
        int maxTime = 0;
        for (int i = 1; i <= n; i++) {
            if (distances[i] == Integer.MAX_VALUE) {
                return -1;  // Node unreachable
            }
            maxTime = Math.max(maxTime, distances[i]);
        }

        return maxTime;
    }
```

Time Complexity: O((V + E) log V) where V is number of nodes and E is number of edges **Space Complexity**: O(V + E)

Tries

Conceptual Explanation

A trie (pronounced "try") is a tree-like data structure used for storing a dynamic set of strings, where keys are usually strings. Each node of a trie represents a character of a string, and the path from root to a node represents a prefix of a string.



Java Implementation

```java
// Trie node
class TrieNode {
    private TrieNode[] children;
    private boolean isEndOfWord;

    public TrieNode() {
        children = new TrieNode[26];  // For lowercase English letters
        isEndOfWord = false;
    }

    public TrieNode getChild(char c) {
        return children[c - 'a'];
    }

    public void setChild(char c, TrieNode node) {
        children[c - 'a'] = node;
    }

    public boolean containsChild(char c) {
        return children[c - 'a'] != null;
    }

    public boolean isEndOfWord() {
        return isEndOfWord;
    }

    public void setEndOfWord(boolean isEndOfWord) {
        this.isEndOfWord = isEndOfWord;
    }
}

// Trie data structure
class Trie {
    private TrieNode root;

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

    // Insert a word into the trie
    public void insert(String word) {
        TrieNode current = root;

        for (char c : word.toCharArray()) {
            if (!current.containsChild(c)) {
                current.setChild(c, new TrieNode());
            }
            current = current.getChild(c);
        }

        current.setEndOfWord(true);
    }
```

```java
    // Search for a word in the trie
    public boolean search(String word) {
        TrieNode node = searchPrefix(word);
        return node != null && node.isEndOfWord();
    }

    // Check if there is any word that starts with the given prefix
    public boolean startsWith(String prefix) {
        return searchPrefix(prefix) != null;
    }

    // Helper method to find node for prefix
    private TrieNode searchPrefix(String prefix) {
        TrieNode current = root;

        for (char c : prefix.toCharArray()) {
            if (!current.containsChild(c)) {
                return null;
            }
            current = current.getChild(c);
        }

        return current;
    }

    // Delete a word from the trie
    public void delete(String word) {
        delete(root, word, 0);
    }

    private boolean delete(TrieNode current, String word, int index) {
        if (index == word.length()) {
            // Word found, but it's not the end of a word
            if (!current.isEndOfWord()) {
                return false;
            }

            current.setEndOfWord(false);

            // Check if current node has no children
            for (char c = 'a'; c <= 'z'; c++) {
                if (current.containsChild(c)) {
                    return false;
                }
            }

            return true;
        }

        char c = word.charAt(index);
        if (!current.containsChild(c)) {
            return false;
        }
```

```
            boolean shouldDeleteChild = delete(current.getChild(c), word, index + 1);

            if (shouldDeleteChild) {
                current.setChild(c, null);

                // Check if current node is end of another word
                if (current.isEndOfWord()) {
                    return false;
                }

                // Check if current node has other children
                for (char ch = 'a'; ch <= 'z'; ch++) {
                    if (current.containsChild(ch)) {
                        return false;
                    }
                }

                return true;
            }

            return false;
        }
    }
```

Time and Space Complexity

- **Insert**: O(m) where m is the length of the word
- **Search**: O(m) where m is the length of the word
- **Delete**: O(m) where m is the length of the word
- **Space**: O(n × m) where n is the number of words and m is the average length of words

Common Variations and Applications

1. **Standard Trie**: For storing and retrieving strings
2. **Compressed Trie**: Combines nodes with single children
3. **Suffix Trie**: Stores all suffixes of a string
4. **Ternary Search Trie**: Each node has three children
5. **Applications**: Autocomplete, spell checkers, IP routing, word games

Typical Interview Questions

1. Implement Trie (Prefix Tree)

Problem: Implement a trie with insert, search, and startsWith methods.

Solution:

```
class Trie {
    private TrieNode root;
```

```
    class TrieNode {
        TrieNode[] children;
        boolean isEndOfWord;

        public TrieNode() {
            children = new TrieNode[26];
            isEndOfWord = false;
        }
    }

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

    public void insert(String word) {
        TrieNode current = root;

        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                current.children[index] = new TrieNode();
            }
            current = current.children[index];
        }

        current.isEndOfWord = true;
    }

    public boolean search(String word) {
        TrieNode node = searchPrefix(word);
        return node != null && node.isEndOfWord;
    }

    public boolean startsWith(String prefix) {
        return searchPrefix(prefix) != null;
    }

    private TrieNode searchPrefix(String prefix) {
        TrieNode current = root;

        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                return null;
            }
            current = current.children[index];
        }

        return current;
    }
}
```

Time Complexity:

- Insert: O(m) where m is the length of the word
- Search: O(m) where m is the length of the word
- StartsWith: O(m) where m is the length of the prefix **Space Complexity**: O(n × m) where n is the number of words and m is the average length of words

2. Word Search II

Problem: Find all words in a board from a given dictionary.

Solution:

```
public List<String> findWords(char[][] board, String[] words) {
    List<String> result = new ArrayList<>();

    // Edge cases
    if (board == null || board.length == 0 || words == null || words.length == 0)
{
        return result;
    }

    // Build trie
    TrieNode root = buildTrie(words);

    int rows = board.length;
    int cols = board[0].length;

    // Search each cell as starting point
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            dfs(board, i, j, root, result);
        }
    }

    return result;
}

private void dfs(char[][] board, int i, int j, TrieNode node, List<String> result)
{
    int rows = board.length;
    int cols = board[0].length;

    // Check boundaries and if character exists in trie
    if (i < 0 || i >= rows || j < 0 || j >= cols ||
        board[i][j] == '#' || node.children[board[i][j] - 'a'] == null) {
        return;
    }

    char c = board[i][j];
    node = node.children[c - 'a'];

    // If we found a word
    if (node.word != null) {
```

```
            result.add(node.word);
            node.word = null;  // Avoid duplicates
        }

        // Mark cell as visited
        board[i][j] = '#';

        // Explore neighbors
        dfs(board, i + 1, j, node, result);
        dfs(board, i - 1, j, node, result);
        dfs(board, i, j + 1, node, result);
        dfs(board, i, j - 1, node, result);

        // Restore cell
        board[i][j] = c;
    }

    private TrieNode buildTrie(String[] words) {
        TrieNode root = new TrieNode();

        for (String word : words) {
            TrieNode current = root;

            for (char c : word.toCharArray()) {
                int index = c - 'a';
                if (current.children[index] == null) {
                    current.children[index] = new TrieNode();
                }
                current = current.children[index];
            }

            current.word = word;  // Store the complete word
        }

        return root;
    }

    class TrieNode {
        TrieNode[] children = new TrieNode[26];
        String word;  // Store full word at end node
    }
```

Time Complexity: O(m × n × 4^L) where m×n is the size of the board and L is the maximum length of words
Space Complexity: O(k) where k is the total number of characters in the dictionary

3. Replace Words

Problem: Replace all words in a sentence with their root prefixes.

Solution:

```java
public String replaceWords(List<String> dictionary, String sentence) {
    // Build trie
    Trie trie = new Trie();
    for (String root : dictionary) {
        trie.insert(root);
    }

    StringBuilder result = new StringBuilder();
    String[] words = sentence.split("\\s+");

    for (String word : words) {
        // Find shortest prefix
        String prefix = trie.shortestPrefix(word);
        if (prefix.isEmpty()) {
            result.append(word);
        } else {
            result.append(prefix);
        }
        result.append(" ");
    }

    return result.toString().trim();
}

class TrieNode {
    TrieNode[] children = new TrieNode[26];
    boolean isEndOfWord;
}

class Trie {
    private TrieNode root;

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

    public void insert(String word) {
        TrieNode current = root;

        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                current.children[index] = new TrieNode();
            }
            current = current.children[index];
        }

        current.isEndOfWord = true;
    }

    public String shortestPrefix(String word) {
        TrieNode current = root;
        StringBuilder prefix = new StringBuilder();
```

```
        for (char c : word.toCharArray()) {
            int index = c - 'a';

            if (current.children[index] == null) {
                return "";  // No prefix found
            }

            prefix.append(c);
            current = current.children[index];

            if (current.isEndOfWord) {
                return prefix.toString();  // Found a prefix
            }
        }

        return "";  // No prefix found
    }
}
```

Time Complexity: O(n + m) where n is the total length of the dictionary words and m is the length of the sentence **Space Complexity**: O(n)

Sorting Algorithms

Selection Sort

Conceptual Explanation

Selection sort divides the array into a sorted and an unsorted region. In each iteration, it finds the minimum element from the unsorted region and moves it to the end of the sorted region.

```
Initial:  [64, 25, 12, 22, 11]
Step 1:   [11, 25, 12, 22, 64]  (Swap 11 and 64)
Step 2:   [11, 12, 25, 22, 64]  (Swap 12 and 25)
Step 3:   [11, 12, 22, 25, 64]  (Swap 22 and 25)
Step 4:   [11, 12, 22, 25, 64]  (No swap needed)
```

Java Implementation

```
public void selectionSort(int[] arr) {
    int n = arr.length;

    for (int i = 0; i < n - 1; i++) {
        // Find minimum element in unsorted part
        int minIndex = i;
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[minIndex]) {
```

```
                minIndex = j;
            }
        }

        // Swap minimum element with first element of unsorted part
        int temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}
```

Time and Space Complexity

- **Time Complexity**: O(n^2) for all cases (best, average, worst)
- **Space Complexity**: O(1)
- **Stable**: No
- **In-place**: Yes

Insertion Sort

Conceptual Explanation

Insertion sort builds the sorted array one element at a time. It takes an element from the unsorted part and inserts it into its correct position in the sorted part.

```
Initial:  [64, 25, 12, 22, 11]
Step 1:   [25, 64, 12, 22, 11]  (Insert 25 before 64)
Step 2:   [12, 25, 64, 22, 11]  (Insert 12 before 25)
Step 3:   [12, 22, 25, 64, 11]  (Insert 22 after 12)
Step 4:   [11, 12, 22, 25, 64]  (Insert 11 before 12)
```

Java Implementation

```
public void insertionSort(int[] arr) {
    int n = arr.length;

    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;

        // Move elements greater than key one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }

        arr[j + 1] = key;
```

```
    }
}
```

Time and Space Complexity

- **Time Complexity**:
  - Best case: O(n) when array is already sorted
  - Average/Worst case: $O(n^2)$
- **Space Complexity**: O(1)
- **Stable**: Yes
- **In-place**: Yes

Bubble Sort

Conceptual Explanation

Bubble sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until no swaps are needed.

```
Initial:  [64, 25, 12, 22, 11]
Step 1:   [25, 64, 12, 22, 11]  (Swap 64 and 25)
Step 2:   [25, 12, 64, 22, 11]  (Swap 64 and 12)
Step 3:   [25, 12, 22, 64, 11]  (Swap 64 and 22)
Step 4:   [25, 12, 22, 11, 64]  (Swap 64 and 11)
Step 5:   [12, 25, 22, 11, 64]  (Swap 25 and 12)
Step 6:   [12, 22, 25, 11, 64]  (Swap 25 and 22)
Step 7:   [12, 22, 11, 25, 64]  (Swap 25 and 11)
Step 8:   [12, 11, 22, 25, 64]  (Swap 22 and 11)
Step 9:   [11, 12, 22, 25, 64]  (Swap 12 and 11)
```

Java Implementation

```java
public void bubbleSort(int[] arr) {
    int n = arr.length;
    boolean swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = false;

        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap adjacent elements
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
```

```
            }

            // If no swapping occurred in this pass, array is sorted
            if (!swapped) {
                break;
            }
        }
    }
```

Time and Space Complexity

- **Time Complexity**:
  - Best case: O(n) when array is already sorted
  - Average/Worst case: $O(n^2)$
- **Space Complexity**: O(1)
- **Stable**: Yes
- **In-place**: Yes

Merge Sort

Conceptual Explanation

Merge sort is a divide-and-conquer algorithm. It divides the array into two halves, recursively sorts them, and then merges the sorted halves.

```
Original: [38, 27, 43, 3, 9, 82, 10]

Divide:
[38, 27, 43, 3] and [9, 82, 10]
[38, 27] and [43, 3] and [9, 82] and [10]
[38] and [27] and [43] and [3] and [9] and [82] and [10]

Merge:
[27, 38] and [3, 43] and [9, 82] and [10]
[3, 27, 38, 43] and [9, 10, 82]
[3, 9, 10, 27, 38, 43, 82]
```

Java Implementation

```
public void mergeSort(int[] arr) {
    if (arr.length > 1) {
        mergeSortHelper(arr, 0, arr.length - 1);
    }
}

private void mergeSortHelper(int[] arr, int left, int right) {
```

```
        if (left < right) {
            // Find the middle point
            int mid = left + (right - left) / 2;

            // Sort first and second halves
            mergeSortHelper(arr, left, mid);
            mergeSortHelper(arr, mid + 1, right);

            // Merge the sorted halves
            merge(arr, left, mid, right);
        }
    }

    private void merge(int[] arr, int left, int mid, int right) {
        // Create temporary arrays
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int[] L = new int[n1];
        int[] R = new int[n2];

        // Copy data to temporary arrays
        for (int i = 0; i < n1; i++) {
            L[i] = arr[left + i];
        }
        for (int j = 0; j < n2; j++) {
            R[j] = arr[mid + 1 + j];
        }

        // Merge the temporary arrays
        int i = 0, j = 0, k = left;

        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k] = L[i];
                i++;
            } else {
                arr[k] = R[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements of L[] if any
        while (i < n1) {
            arr[k] = L[i];
            i++;
            k++;
        }

        // Copy remaining elements of R[] if any
        while (j < n2) {
            arr[k] = R[j];
            j++;
```

```
            k++;
        }
    }
```

Time and Space Complexity

- **Time Complexity**: O(n log n) for all cases (best, average, worst)
- **Space Complexity**: O(n)
- **Stable**: Yes
- **In-place**: No

Quick Sort

Conceptual Explanation

Quick sort is another divide-and-conquer algorithm. It picks an element as a pivot and partitions the array around the pivot such that elements smaller than the pivot are on the left and larger elements are on the right.

```
Original: [10, 80, 30, 90, 40, 50, 70]

Partition (pivot = 70):
[10, 30, 40, 50] and [70] and [80, 90]

Recursively sort subarrays:
[10, 30, 40, 50] and [80, 90]

Result:
[10, 30, 40, 50, 70, 80, 90]
```

Java Implementation

```
public void quickSort(int[] arr) {
    quickSortHelper(arr, 0, arr.length - 1);
}

private void quickSortHelper(int[] arr, int low, int high) {
    if (low < high) {
        // Partition the array
        int pivotIndex = partition(arr, low, high);

        // Sort elements before and after partition
        quickSortHelper(arr, low, pivotIndex - 1);
        quickSortHelper(arr, pivotIndex + 1, high);
    }
}
```

```
    private int partition(int[] arr, int low, int high) {
        // Choose the rightmost element as pivot
        int pivot = arr[high];

        // Index of smaller element
        int i = low - 1;

        for (int j = low; j < high; j++) {
            // If current element is smaller than the pivot
            if (arr[j] < pivot) {
                i++;

                // Swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        // Swap arr[i+1] and arr[high] (pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }
```

Time and Space Complexity

- **Time Complexity**:
  - Best/Average case: O(n log n)
  - Worst case: $O(n^2)$ when the array is already sorted
- **Space Complexity**: O(log n) for recursion stack
- **Stable**: No
- **In-place**: Yes

Heap Sort

Conceptual Explanation

Heap sort uses a binary heap data structure to sort elements. It builds a max heap from the array and repeatedly extracts the maximum element.

```
Original: [4, 10, 3, 5, 1]

Build max heap:
         10
        /  \
       5    3
      / \
```

```
     4   1

Extract max and heapify:
[1, 4, 3, 5] and [10]
[1, 4, 3] and [5, 10]
[1, 3] and [4, 5, 10]
[1] and [3, 4, 5, 10]
[] and [1, 3, 4, 5, 10]

Result:
[1, 3, 4, 5, 10]
```

Java Implementation

```java
public void heapSort(int[] arr) {
    int n = arr.length;

    // Build max heap
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // Extract elements from heap one by one
    for (int i = n - 1; i > 0; i--) {
        // Move current root to the end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

private void heapify(int[] arr, int n, int i) {
    int largest = i;  // Initialize largest as root
    int left = 2 * i + 1;
    int right = 2 * i + 2;

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // If largest is not root
    if (largest != i) {
```

```
            int temp = arr[i];
            arr[i] = arr[largest];
            arr[largest] = temp;

            // Recursively heapify the affected subtree
            heapify(arr, n, largest);
        }
    }
```

Time and Space Complexity

- **Time Complexity**: O(n log n) for all cases (best, average, worst)
- **Space Complexity**: O(1)
- **Stable**: No
- **In-place**: Yes

Counting Sort

Conceptual Explanation

Counting sort is a non-comparative sorting algorithm that works well when the range of input values is small. It counts the occurrences of each input value and uses this information to place elements in their correct positions.

```
Original: [4, 2, 2, 8, 3, 3, 1]

Count occurrences:
Index:    0  1  2  3  4  5  6  7  8
Count:    0  1  2  2  1  0  0  0  1

Calculate positions:
Index:    0  1  2  3  4  5  6  7  8
Position: 0  0  1  3  5  6  6  6  6

Place elements in output array:
[1, 2, 2, 3, 3, 4, 8]
```

Java Implementation

```
public void countingSort(int[] arr) {
    int n = arr.length;

    // Find the maximum value
    int max = arr[0];
    for (int i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
```

```
            }
        }

        // Create count array
        int[] count = new int[max + 1];

        // Count occurrences
        for (int i = 0; i < n; i++) {
            count[arr[i]]++;
        }

        // Calculate positions
        for (int i = 1; i <= max; i++) {
            count[i] += count[i - 1];
        }

        // Create output array
        int[] output = new int[n];

        // Place elements in correct positions
        for (int i = n - 1; i >= 0; i--) {
            output[count[arr[i]] - 1] = arr[i];
            count[arr[i]]--;
        }

        // Copy output array to original array
        for (int i = 0; i < n; i++) {
            arr[i] = output[i];
        }
    }
```

Time and Space Complexity

- **Time Complexity**: O(n + k) where n is the size of the array and k is the range of input values
- **Space Complexity**: O(n + k)
- **Stable**: Yes
- **In-place**: No

Radix Sort

Conceptual Explanation

Radix sort sorts integers by processing individual digits. It sorts numbers from the least significant digit to the most significant digit using a stable sort algorithm (usually counting sort).

```
Original: [170, 45, 75, 90, 802, 24, 2, 66]

Sort by least significant digit (ones place):
[170, 90, 802, 2, 24, 45, 75, 66]
```

```
Sort by second digit (tens place):
[802, 2, 24, 45, 66, 70, 75, 90]

Sort by most significant digit (hundreds place):
[2, 24, 45, 66, 75, 90, 170, 802]
```

Java Implementation

```java
public void radixSort(int[] arr) {
    // Find the maximum number to know number of digits
    int max = getMax(arr);

    // Do counting sort for every digit
    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSortByDigit(arr, exp);
    }
}

private int getMax(int[] arr) {
    int max = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    return max;
}

private void countingSortByDigit(int[] arr, int exp) {
    int n = arr.length;
    int[] output = new int[n];
    int[] count = new int[10];  // 0-9 digits

    // Store count of occurrences in count[]
    for (int i = 0; i < n; i++) {
        count[(arr[i] / exp) % 10]++;
    }

    // Change count[i] so that count[i] contains position of digit in output[]
    for (int i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }

    // Build the output array
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the output array to arr[]
    for (int i = 0; i < n; i++) {
```

```
            arr[i] = output[i];
        }
    }
```

Time and Space Complexity

- **Time Complexity**: O(d * (n + k)) where d is the number of digits, n is the size of the array, and k is the range of input values (typically 10 for decimal digits)
- **Space Complexity**: O(n + k)
- **Stable**: Yes
- **In-place**: No

Bucket Sort

Conceptual Explanation

Bucket sort divides the range of input values into equal-sized buckets. It then sorts each bucket individually (often using another sorting algorithm) and concatenates the results.

```
Original: [0.42, 0.32, 0.73, 0.12, 0.68, 0.94, 0.30, 0.87]

Distribute into buckets:
Bucket 0 (0.0-0.1): []
Bucket 1 (0.1-0.2): [0.12]
Bucket 2 (0.2-0.3): []
Bucket 3 (0.3-0.4): [0.32, 0.30]
Bucket 4 (0.4-0.5): [0.42]
Bucket 5 (0.5-0.6): []
Bucket 6 (0.6-0.7): [0.68]
Bucket 7 (0.7-0.8): [0.73]
Bucket 8 (0.8-0.9): [0.87]
Bucket 9 (0.9-1.0): [0.94]

Sort each bucket:
Bucket 3: [0.30, 0.32]

Concatenate buckets:
[0.12, 0.30, 0.32, 0.42, 0.68, 0.73, 0.87, 0.94]
```

Java Implementation

```
public void bucketSort(double[] arr) {
    int n = arr.length;

    if (n <= 0) {
        return;
    }
```

```
        // Create buckets
        List<List<Double>> buckets = new ArrayList<>(n);
        for (int i = 0; i < n; i++) {
            buckets.add(new ArrayList<>());
        }

        // Add elements to buckets
        for (int i = 0; i < n; i++) {
            int bucketIndex = (int) (n * arr[i]);
            buckets.get(bucketIndex).add(arr[i]);
        }

        // Sort each bucket
        for (int i = 0; i < n; i++) {
            Collections.sort(buckets.get(i));
        }

        // Concatenate all buckets
        int index = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < buckets.get(i).size(); j++) {
                arr[index++] = buckets.get(i).get(j);
            }
        }
    }
```

Time and Space Complexity

- **Time Complexity**:
  - Average case: O(n + n^2/k + k) where n is the size of the array and k is the number of buckets
  - Best case: O(n) when elements are uniformly distributed
  - Worst case: O(n^2) when all elements fall into one bucket
- **Space Complexity**: O(n + k)
- **Stable**: Yes (if the bucket sorting algorithm is stable)
- **In-place**: No

Searching Algorithms

Linear Search

Conceptual Explanation

Linear search sequentially checks each element of the list until it finds the target element or reaches the end of the list.

```
Array: [5, 2, 9, 1, 7, 3]
Target: 7

Step 1: Check 5, not equal to 7
```

```
Step 2: Check 2, not equal to 7
Step 3: Check 9, not equal to 7
Step 4: Check 1, not equal to 7
Step 5: Check 7, equal to 7 -> Target found at index 4
```

Java Implementation

```java
public int linearSearch(int[] arr, int target) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == target) {
            return i;  // Element found at index i
        }
    }
    return -1;  // Element not found
}
```

Time and Space Complexity

- **Time Complexity**: O(n) in worst and average case
- **Space Complexity**: O(1)

Binary Search

Conceptual Explanation

Binary search is a divide-and-conquer algorithm that works on sorted arrays. It repeatedly divides the search space in half by comparing the target value with the middle element.

```
Sorted Array: [1, 2, 3, 5, 7, 9]
Target: 7

Step 1: Compare with middle element (3)
        7 > 3, search right half [5, 7, 9]
Step 2: Compare with middle element (7)
        7 == 7 -> Target found at index 4
```

Java Implementation

```java
// Iterative implementation
public int binarySearch(int[] arr, int target) {
    int left = 0;
    int right = arr.length - 1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
```

```
            // Check if target is present at mid
            if (arr[mid] == target) {
                return mid;
            }

            // If target is greater, ignore left half
            if (arr[mid] < target) {
                left = mid + 1;
            }
            // If target is smaller, ignore right half
            else {
                right = mid - 1;
            }
        }

        return -1;  // Element not found
    }

    // Recursive implementation
    public int binarySearchRecursive(int[] arr, int target, int left, int right) {
        if (left > right) {
            return -1;  // Element not found
        }

        int mid = left + (right - left) / 2;

        // Check if target is present at mid
        if (arr[mid] == target) {
            return mid;
        }

        // If target is greater, search in right half
        if (arr[mid] < target) {
            return binarySearchRecursive(arr, target, mid + 1, right);
        }

        // If target is smaller, search in left half
        return binarySearchRecursive(arr, target, left, mid - 1);
    }
```

Time and Space Complexity

- **Time Complexity**: O(log n)
- **Space Complexity**:
  - O(1) for iterative approach
  - O(log n) for recursive approach due to call stack

Interpolation Search

Conceptual Explanation

Interpolation search is an improved variant of binary search that works best for uniformly distributed data. Instead of always checking the middle element, it estimates the position of the target value.

```
Formula for position:
pos = left + ((target - arr[left]) * (right - left)) / (arr[right] - arr[left])
```

Java Implementation

```
public int interpolationSearch(int[] arr, int target) {
    int left = 0;
    int right = arr.length - 1;

    while (left <= right && target >= arr[left] && target <= arr[right]) {
        if (left == right) {
            if (arr[left] == target) {
                return left;
            }
            return -1;
        }

        // Calculate the probable position
        int pos = left + ((target - arr[left]) * (right - left)) / (arr[right] -
arr[left]);

        if (arr[pos] == target) {
            return pos;
        }

        if (arr[pos] < target) {
            left = pos + 1;
        } else {
            right = pos - 1;
        }
    }

    return -1;  // Element not found
}
```

Time and Space Complexity

- **Time Complexity**:
  - Average case: O(log log n) for uniformly distributed data
  - Worst case: O(n) for very skewed distributions
- **Space Complexity**: O(1)

Dynamic Programming

Conceptual Explanation

Dynamic Programming (DP) is a method for solving complex problems by breaking them down into simpler subproblems. It solves each subproblem only once and stores the results to avoid redundant calculations.

Key characteristics of Dynamic Programming problems:

1. **Optimal Substructure**: An optimal solution to the problem contains optimal solutions to subproblems
2. **Overlapping Subproblems**: The same subproblems are solved multiple times

DP typically follows these steps:

1. Define the subproblems
2. Write down the recurrence relation
3. Solve the base cases
4. Implement the solution (either top-down with memoization or bottom-up with tabulation)

Example Problems

1. Fibonacci Sequence

Problem: Find the nth Fibonacci number.

Solution (Top-down with Memoization):

```
public int fib(int n) {
    Map<Integer, Integer> memo = new HashMap<>();
    return fibMemoized(n, memo);
}

private int fibMemoized(int n, Map<Integer, Integer> memo) {
    if (n <= 1) {
        return n;
    }

    if (memo.containsKey(n)) {
        return memo.get(n);
    }

    int result = fibMemoized(n - 1, memo) + fibMemoized(n - 2, memo);
    memo.put(n, result);

    return result;
}
```

Solution (Bottom-up with Tabulation):

```
public int fib(int n) {
    if (n <= 1) {
        return 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];
        }

        return dp[n];
    }
```

Time Complexity: O(n) **Space Complexity**: O(n)

2. Longest Common Subsequence

Problem: Find the length of the longest subsequence present in both strings.

Solution:

```
public int longestCommonSubsequence(String text1, String text2) {
    int m = text1.length();
    int n = text2.length();

    // Create DP table
    int[][] dp = new int[m + 1][n + 1];

    // Fill the DP table
    for (int i = 1; i <= m; i++) {
        for (int j = 1; j <= n; j++) {
            if (text1.charAt(i - 1) == text2.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];
}
```

Time Complexity: O(m _ n) where m and n are the lengths of the input strings **Space Complexity**: O(m _ n)

3. Knapsack Problem

Problem: Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value.

Solution:

```
public int knapsack(int[] values, int[] weights, int capacity) {
    int n = values.length;
    int[][] dp = new int[n + 1][capacity + 1];

    for (int i = 1; i <= n; i++) {
        for (int w = 1; w <= capacity; w++) {
            if (weights[i - 1] <= w) {
                dp[i][w] = Math.max(
                    values[i - 1] + dp[i - 1][w - weights[i - 1]],  // Include
item
                    dp[i - 1][w]  // Exclude item
                );
            } else {
                dp[i][w] = dp[i - 1][w];  // Can't include item
            }
        }
    }

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

Time Complexity: O(n _ W) where n is the number of items and W is the capacity **Space Complexity**: O(n _ W)

4. Coin Change

Problem: Find the minimum number of coins that you need to make up a given amount.

Solution:

```
public int coinChange(int[] coins, int amount) {
    // Initialize dp array with amount+1 (which is greater than max possible
coins)
    int[] dp = new int[amount + 1];
    Arrays.fill(dp, amount + 1);
    dp[0] = 0;

    for (int coin : coins) {
        for (int i = coin; i <= amount; i++) {
            dp[i] = Math.min(dp[i], dp[i - coin] + 1);
        }
    }

    return dp[amount] > amount ? -1 : dp[amount];
}
```

Time Complexity: O(amount * n) where n is the number of coin denominations **Space Complexity**: O(amount)

Greedy Algorithms

Conceptual Explanation

Greedy algorithms make locally optimal choices at each step with the hope of finding a global optimum. Unlike dynamic programming, greedy algorithms don't guarantee an optimal solution for all problems but are generally more efficient.

Key characteristics of Greedy problems:

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

Example Problems

1. Activity Selection

Problem: Select the maximum number of activities that don't overlap.

Solution:

```
public int maxActivities(int[] start, int[] finish) {
    int n = start.length;

    // Create a list of activities with start and finish times
    List<int[]> activities = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        activities.add(new int[] {start[i], finish[i]});
    }

    // Sort activities by finish time
    activities.sort((a, b) -> a[1] - b[1]);

    int count = 1;  // Select first activity
    int endTime = activities.get(0)[1];

    // Consider rest of the activities
    for (int i = 1; i < n; i++) {
        if (activities.get(i)[0] >= endTime) {
            count++;
            endTime = activities.get(i)[1];
        }
    }

    return count;
}
```

Time Complexity: O(n log n) due to sorting **Space Complexity**: O(n)

2. Fractional Knapsack

Problem: Fill a knapsack with fractions of items to maximize value.

Solution:

```
public double fractionalKnapsack(int[] values, int[] weights, int capacity) {
    int n = values.length;

    // Create array of item value/weight ratios
    Item[] items = new Item[n];
    for (int i = 0; i < n; i++) {
        items[i] = new Item(values[i], weights[i], (double) values[i] /
weights[i]);
    }

    // Sort items by value/weight ratio in descending order
    Arrays.sort(items, (a, b) -> Double.compare(b.ratio, a.ratio));

    double totalValue = 0;
    int currentWeight = 0;

    for (Item item : items) {
        // If adding the whole item doesn't exceed capacity
        if (currentWeight + item.weight <= capacity) {
            currentWeight += item.weight;
            totalValue += item.value;
        } else {
            // Take a fraction of the item
            int remainingCapacity = capacity - currentWeight;
            totalValue += item.ratio * remainingCapacity;
            break;
        }
    }

    return totalValue;
}

private class Item {
    int value;
    int weight;
    double ratio;  // value/weight ratio

    Item(int value, int weight, double ratio) {
        this.value = value;
        this.weight = weight;
        this.ratio = ratio;
    }
}
```

Time Complexity: O(n log n) due to sorting **Space Complexity**: O(n)

3. Huffman Coding

Problem: Create a variable-length prefix code for characters based on their frequencies.

Solution:

```
public class HuffmanCoding {
    // Node class for Huffman Tree
    private class Node implements Comparable<Node> {
        char ch;
        int freq;
        Node left;
        Node right;

        Node(char ch, int freq) {
            this.ch = ch;
            this.freq = freq;
        }

        Node(char ch, int freq, Node left, Node right) {
            this.ch = ch;
            this.freq = freq;
            this.left = left;
            this.right = right;
        }

        @Override
        public int compareTo(Node other) {
            return this.freq - other.freq;
        }
    }

    public Map<Character, String> buildHuffmanCodes(String text) {
        // Count frequency of each character
        Map<Character, Integer> freqMap = new HashMap<>();
        for (char c : text.toCharArray()) {
            freqMap.put(c, freqMap.getOrDefault(c, 0) + 1);
        }

        // Create a priority queue (min-heap) of nodes
        PriorityQueue<Node> pq = new PriorityQueue<>();
        for (Map.Entry<Character, Integer> entry : freqMap.entrySet()) {
            pq.offer(new Node(entry.getKey(), entry.getValue()));
        }

        // Build Huffman Tree
        while (pq.size() > 1) {
            // Remove two nodes with lowest frequency
            Node left = pq.poll();
            Node right = pq.poll();

            // Create a new internal node with these two nodes as children
            pq.offer(new Node('\0', left.freq + right.freq, left, right));
        }
```

```
        // Get the root of Huffman Tree
        Node root = pq.poll();

        // Generate Huffman codes
        Map<Character, String> huffmanCodes = new HashMap<>();
        generateCodes(root, "", huffmanCodes);

        return huffmanCodes;
    }

    private void generateCodes(Node node, String code, Map<Character, String>
huffmanCodes) {
        if (node == null) {
            return;
        }

        // If this is a leaf node (contains a character)
        if (node.left == null && node.right == null) {
            huffmanCodes.put(node.ch, code.isEmpty() ? "1" : code);
        }

        // Traverse left (add '0' to code)
        generateCodes(node.left, code + "0", huffmanCodes);

        // Traverse right (add '1' to code)
        generateCodes(node.right, code + "1", huffmanCodes);
    }
}
```

Time Complexity: O(n log n) where n is the number of unique characters **Space Complexity**: O(n)

Divide and Conquer

Conceptual Explanation

Divide and Conquer is an algorithmic paradigm that breaks a problem into smaller subproblems, solves them recursively, and then combines their solutions to solve the original problem. It consists of three main steps:

1. **Divide**: Break the problem into smaller subproblems
2. **Conquer**: Solve the subproblems recursively
3. **Combine**: Combine the solutions of subproblems to form the solution of the original problem

Example Problems

1. Merge Sort

Already covered in the sorting algorithms section.

2. Quick Sort

Already covered in the sorting algorithms section.

3. Maximum Subarray Sum

Problem: Find the contiguous subarray with the largest sum.

Solution:

```
public int maxSubArray(int[] nums) {
    return maxSubArrayHelper(nums, 0, nums.length - 1);
}

private int maxSubArrayHelper(int[] nums, int left, int right) {
    if (left == right) {
        return nums[left];
    }

    int mid = left + (right - left) / 2;

    // Find maximum subarray sum in left half
    int leftMax = maxSubArrayHelper(nums, left, mid);

    // Find maximum subarray sum in right half
    int rightMax = maxSubArrayHelper(nums, mid + 1, right);

    // Find maximum subarray sum that crosses the midpoint
    int crossMax = maxCrossingSum(nums, left, mid, right);

    // Return maximum of the three
    return Math.max(Math.max(leftMax, rightMax), crossMax);
}

private int maxCrossingSum(int[] nums, int left, int mid, int right) {
    // Find maximum sum starting from mid point and going left
    int sum = 0;
    int leftSum = Integer.MIN_VALUE;

    for (int i = mid; i >= left; i--) {
        sum += nums[i];
        leftSum = Math.max(leftSum, sum);
    }

    // Find maximum sum starting from mid+1 and going right
    sum = 0;
    int rightSum = Integer.MIN_VALUE;

    for (int i = mid + 1; i <= right; i++) {
        sum += nums[i];
        rightSum = Math.max(rightSum, sum);
    }

    // Return sum of the two halves
    return leftSum + rightSum;
}
```

Time Complexity: O(n log n) **Space Complexity**: O(log n) due to recursion stack

4. Closest Pair of Points

Problem: Find the closest pair of points in a set of points in a 2D plane.

Solution:

```
public class ClosestPair {
    private static class Point {
        double x, y;

        Point(double x, double y) {
            this.x = x;
            this.y = y;
        }
    }

    // Calculate distance between two points
    private double distance(Point p1, Point p2) {
        return Math.sqrt((p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y -
p2.y));
    }

    // Find the closest pair of points
    public double closestPair(Point[] points) {
        // Sort points by x-coordinate
        Arrays.sort(points, (a, b) -> Double.compare(a.x, b.x));

        return closestPairHelper(points, 0, points.length - 1);
    }

    private double closestPairHelper(Point[] points, int start, int end) {
        // Base case: If there are 2 or 3 points, use brute force
        if (end - start <= 2) {
            return bruteForce(points, start, end);
        }

        int mid = start + (end - start) / 2;

        // Find the middle point
        double midX = points[mid].x;

        // Recursively find the smallest distance in both halves
        double leftMin = closestPairHelper(points, start, mid);
        double rightMin = closestPairHelper(points, mid + 1, end);

        // Find the smaller of the two
        double minDist = Math.min(leftMin, rightMin);

        // Build an array strip[] that contains points close to the line of
```

```
separation
        List<Point> strip = new ArrayList<>();
        for (int i = start; i <= end; i++) {
            if (Math.abs(points[i].x - midX) < minDist) {
                strip.add(points[i]);
            }
        }

        // Sort the strip by y-coordinate
        strip.sort((a, b) -> Double.compare(a.y, b.y));

        // Find the closest points in the strip
        for (int i = 0; i < strip.size(); i++) {
            for (int j = i + 1; j < strip.size() && strip.get(j).y -
strip.get(i).y < minDist; j++) {
                minDist = Math.min(minDist, distance(strip.get(i), strip.get(j)));
            }
        }

        return minDist;
    }

    private double bruteForce(Point[] points, int start, int end) {
        double minDist = Double.MAX_VALUE;

        for (int i = start; i < end; i++) {
            for (int j = i + 1; j <= end; j++) {
                minDist = Math.min(minDist, distance(points[i], points[j]));
            }
        }

        return minDist;
    }
}
```

Time Complexity: O(n $\log^2$ n) **Space Complexity**: O(n)

Backtracking

Conceptual Explanation

Backtracking is an algorithmic technique for solving problems by trying to build a solution incrementally, one piece at a time, and removing those solutions that fail to satisfy the constraints of the problem at any point.

Key characteristics of Backtracking:

1. **Decision Space**: At each step, we make a choice from available options
2. **Constraints**: We check if the current state satisfies all constraints
3. **Goal State**: We check if we've reached a valid solution
4. **Backtrack**: If we reach a state where we can't proceed, we undo the last choice and try another option

Example Problems

1. N-Queens

Problem: Place N queens on an N×N chessboard so that no two queens threaten each other.

Solution:

```java
public List<List<String>> solveNQueens(int n) {
    List<List<String>> solutions = new ArrayList<>();
    char[][] board = new char[n][n];

    // Initialize board with empty cells
    for (int i = 0; i < n; i++) {
        Arrays.fill(board[i], '.');
    }

    backtrack(board, 0, solutions);

    return solutions;
}

private void backtrack(char[][] board, int row, List<List<String>> solutions) {
    // If we've placed queens in all rows, we've found a solution
    if (row == board.length) {
        solutions.add(constructSolution(board));
        return;
    }

    // Try placing queen in each column of the current row
    for (int col = 0; col < board.length; col++) {
        if (isValid(board, row, col)) {
            // Place queen
            board[row][col] = 'Q';

            // Recur for next row
            backtrack(board, row + 1, solutions);

            // Remove queen (backtrack)
            board[row][col] = '.';
        }
    }
}

private boolean isValid(char[][] board, int row, int col) {
    int n = board.length;

    // Check column
    for (int i = 0; i < row; i++) {
        if (board[i][col] == 'Q') {
            return false;
        }
    }
```

```
        // Check left diagonal
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 'Q') {
                return false;
            }
        }

        // Check right diagonal
        for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
            if (board[i][j] == 'Q') {
                return false;
            }
        }

        return true;
    }

    private List<String> constructSolution(char[][] board) {
        List<String> solution = new ArrayList<>();
        for (char[] row : board) {
            solution.add(new String(row));
        }
        return solution;
    }
```

Time Complexity: O(n!) **Space Complexity**: $O(n^2)$

2. Sudoku Solver

Problem: Solve a 9x9 Sudoku puzzle.

Solution:

```
public void solveSudoku(char[][] board) {
    solve(board);
}

private boolean solve(char[][] board) {
    for (int row = 0; row < 9; row++) {
        for (int col = 0; col < 9; col++) {
            // Find an empty cell
            if (board[row][col] == '.') {
                // Try placing digits 1-9
                for (char digit = '1'; digit <= '9'; digit++) {
                    if (isValid(board, row, col, digit)) {
                        // Place digit
                        board[row][col] = digit;

                        // Recursively solve the rest of the puzzle
                        if (solve(board)) {
                            return true;
```

```
                        }

                        // If placing digit didn't lead to a solution, backtrack
                        board[row][col] = '.';
                    }
                }
                // If no digit can be placed, the puzzle is unsolvable
                return false;
            }
        }
    }
    // If no empty cells are found, the puzzle is solved
    return true;
}

private boolean isValid(char[][] board, int row, int col, char digit) {
    // Check row
    for (int i = 0; i < 9; i++) {
        if (board[row][i] == digit) {
            return false;
        }
    }

    // Check column
    for (int i = 0; i < 9; i++) {
        if (board[i][col] == digit) {
            return false;
        }
    }

    // Check 3x3 box
    int boxRow = (row / 3) * 3;
    int boxCol = (col / 3) * 3;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (board[boxRow + i][boxCol + j] == digit) {
                return false;
            }
        }
    }

    return true;
}
```

Time Complexity: O(9^(n^2)) where n is the board size (typically 9) **Space Complexity**: O(n^2)

3. Combination Sum

Problem: Find all unique combinations of candidates that sum to a target.

Solution:

```java
public List<List<Integer>> combinationSum(int[] candidates, int target) {
    List<List<Integer>> results = new ArrayList<>();
    backtrack(candidates, target, 0, new ArrayList<>(), results);
    return results;
}

private void backtrack(int[] candidates, int remain, int start,
                       List<Integer> current, List<List<Integer>> results) {
    // Base cases
    if (remain < 0) {
        return;  // Current combination exceeds target
    }
    if (remain == 0) {
        results.add(new ArrayList<>(current));  // Found a valid combination
        return;
    }

    // Try each candidate from the current position
    for (int i = start; i < candidates.length; i++) {
        // Add the current candidate
        current.add(candidates[i]);

        // Continue to use the current candidate (since we can use it multiple
times)
        backtrack(candidates, remain - candidates[i], i, current, results);

        // Backtrack: remove the candidate to try the next one
        current.remove(current.size() - 1);
    }
}
```

Time Complexity: O(n^target) where n is the number of candidates **Space Complexity**: O(target)

Bit Manipulation

Conceptual Explanation

Bit manipulation involves applying various operations on individual bits of numbers. It's often used to optimize algorithms and solve problems more efficiently.

Basic Bit Operations:

- **AND (&)**: Set bit to 1 if both bits are 1
- **OR (|)**: Set bit to 1 if either bit is 1
- **XOR (^)**: Set bit to 1 if bits are different
- **NOT (~)**: Invert all bits
- **Left Shift (<<)**: Shift bits left, multiply by 2
- **Right Shift (>>)**: Shift bits right, divide by 2

Common Bit Manipulation Techniques:

1. **Check if a bit is set**: `(num & (1 << i)) != 0`
2. **Set a bit**: `num |= (1 << i)`
3. **Clear a bit**: `num &= ~(1 << i)`
4. **Toggle a bit**: `num ^= (1 << i)`
5. **Check if power of 2**: `(num & (num - 1)) == 0 && num != 0`
6. **Count set bits**: Use Brian Kernighan's algorithm

Example Problems

1. Count Set Bits

Problem: Count the number of 1's in the binary representation of a number.

Solution:

```
// Brian Kernighan's algorithm
public int countSetBits(int n) {
    int count = 0;
    while (n > 0) {
        n &= (n - 1);  // Clears the least significant set bit
        count++;
    }
    return count;
}

// Using Integer.bitCount
public int countSetBits(int n) {
    return Integer.bitCount(n);
}
```

Time Complexity: O(k) where k is the number of set bits **Space Complexity**: O(1)

2. Find the Missing Number

Problem: Find the missing number in an array containing n distinct numbers in the range [0, n].

Solution:

```
public int missingNumber(int[] nums) {
    int n = nums.length;
    int missing = n;  // Initialize with n

    for (int i = 0; i < n; i++) {
        missing ^= i ^ nums[i];
    }

    return missing;
}
```

Time Complexity: O(n) **Space Complexity**: O(1)

3. Single Number

Problem: Find the number that appears exactly once in an array where all other numbers appear twice.

Solution:

```
public int singleNumber(int[] nums) {
    int result = 0;

    for (int num : nums) {
        result ^= num;
    }

    return result;
}
```

Time Complexity: O(n) **Space Complexity**: O(1)

Advanced Techniques

A* Search Algorithm

Conceptual Explanation

A* is an informed search algorithm that combines elements of Dijkstra's algorithm and greedy best-first search. It uses a heuristic function to estimate the cost from the current node to the goal.

Java Implementation

```
public class AStar {
    private static class Node implements Comparable<Node> {
        int x, y;
        double g;  // Cost from start
        double h;  // Heuristic (estimated cost to goal)
        double f;  // Total cost (g + h)
        Node parent;

        Node(int x, int y) {
            this.x = x;
            this.y = y;
        }

        @Override
        public int compareTo(Node other) {
            return Double.compare(this.f, other.f);
        }
    }
```

```
    public List<Node> findPath(int[][] grid, int startX, int startY, int goalX,
int goalY) {
        int rows = grid.length;
        int cols = grid[0].length;

        boolean[][] visited = new boolean[rows][cols];
        PriorityQueue<Node> openSet = new PriorityQueue<>();

        // Create start node
        Node start = new Node(startX, startY);
        start.g = 0;
        start.h = heuristic(startX, startY, goalX, goalY);
        start.f = start.g + start.h;

        openSet.offer(start);

        // Directions: up, right, down, left
        int[] dx = {-1, 0, 1, 0};
        int[] dy = {0, 1, 0, -1};

        while (!openSet.isEmpty()) {
            Node current = openSet.poll();

            // Mark as visited
            visited[current.x][current.y] = true;

            // Check if goal reached
            if (current.x == goalX && current.y == goalY) {
                return reconstructPath(current);
            }

            // Check neighbors
            for (int i = 0; i < 4; i++) {
                int nx = current.x + dx[i];
                int ny = current.y + dy[i];

                // Check if valid and unvisited
                if (nx >= 0 && nx < rows && ny >= 0 && ny < cols &&
                    grid[nx][ny] == 0 && !visited[nx][ny]) {

                    Node neighbor = new Node(nx, ny);
                    double tentativeG = current.g + 1;  // Assuming uniform cost

                    // Check if this path is better
                    if (!openSet.contains(neighbor) || tentativeG < neighbor.g) {
                        neighbor.parent = current;
                        neighbor.g = tentativeG;
                        neighbor.h = heuristic(nx, ny, goalX, goalY);
                        neighbor.f = neighbor.g + neighbor.h;

                        if (!openSet.contains(neighbor)) {
                            openSet.offer(neighbor);
                        }
```

```
                    }
                }
            }
        }

        return new ArrayList<>();  // No path found
    }

    private double heuristic(int x1, int y1, int x2, int y2) {
        // Manhattan distance
        return Math.abs(x1 - x2) + Math.abs(y1 - y2);
    }

    private List<Node> reconstructPath(Node node) {
        List<Node> path = new ArrayList<>();

        while (node != null) {
            path.add(0, node);  // Add to front of list
            node = node.parent;
        }

        return path;
    }
}
```

Time and Space Complexity

- **Time Complexity**: O(E log V) where E is the number of edges and V is the number of vertices
- **Space Complexity**: O(V)

Rabin-Karp String Matching Algorithm

Conceptual Explanation

Rabin-Karp is a string-searching algorithm that uses hashing to find patterns in text. It calculates a hash value for the pattern and for each possible substring of the text, then compares them.

Java Implementation

```
public class RabinKarp {
    private static final int PRIME = 101;

    public List<Integer> search(String text, String pattern) {
        List<Integer> matches = new ArrayList<>();
        int n = text.length();
        int m = pattern.length();

        if (n < m) {
            return matches;
        }
```

```
        // Calculate hash for pattern and first window of text
        int patternHash = hash(pattern, m);
        int textHash = hash(text.substring(0, m), m);

        // Calculate largest place value used in hash
        int h = 1;
        for (int i = 0; i < m - 1; i++) {
            h = (h * 256) % PRIME;
        }

        // Slide pattern over text
        for (int i = 0; i <= n - m; i++) {
            // Check if hashes match
            if (patternHash == textHash) {
                // Verify character by character
                boolean match = true;
                for (int j = 0; j < m; j++) {
                    if (text.charAt(i + j) != pattern.charAt(j)) {
                        match = false;
                        break;
                    }
                }

                if (match) {
                    matches.add(i);
                }
            }

            // Calculate hash for next window
            if (i < n - m) {
                textHash = (256 * (textHash - text.charAt(i) * h) + text.charAt(i
+ m)) % PRIME;

                // Handle negative hash
                if (textHash < 0) {
                    textHash += PRIME;
                }
            }
        }

        return matches;
    }

    private int hash(String str, int length) {
        int hash = 0;

        for (int i = 0; i < length; i++) {
            hash = (256 * hash + str.charAt(i)) % PRIME;
        }

        return hash;
    }
}
```

Time and Space Complexity

- **Time Complexity**:
  - Average case: O(n + m) where n is the length of the text and m is the length of the pattern
  - Worst case: O(n * m) when hashes collide frequently
- **Space Complexity**: O(1)

Disjoint Set (Union-Find)

Conceptual Explanation

Union-Find is a data structure that keeps track of elements partitioned into disjoint sets. It supports two main operations: union (merge two sets) and find (determine which set an element belongs to).

Java Implementation

```
public class DisjointSet {
    private int[] parent;
    private int[] rank;

    public DisjointSet(int size) {
        parent = new int[size];
        rank = new int[size];

        // Initially, each element is its own parent
        for (int i = 0; i < size; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    // Find with path compression
    public int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);  // Path compression
        }
        return parent[x];
    }

    // Union by rank
    public void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);

        if (rootX == rootY) {
            return;  // Already in the same set
        }

        // Attach smaller rank tree under root of higher rank tree
```

```
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                // If ranks are the same, make one as root and increment its rank
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }

        public boolean isConnected(int x, int y) {
            return find(x) == find(y);
        }
    }
```

Time and Space Complexity

- **Time Complexity**:
  - Find: O(α(n)) where α is the inverse Ackermann function (effectively constant)
  - Union: O(α(n))
- **Space Complexity**: O(n)

Complexity Summary Table

Data Structure / Algorithm	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
Data Structures			
Array (Access)	O(1)	O(1)	O(n)
Array (Search)	O(n)	O(n)	O(n)
Array (Insert/Delete)	O(n)	O(n)	O(n)
Linked List (Access)	O(n)	O(n)	O(n)
Linked List (Search)	O(n)	O(n)	O(n)
Linked List (Insert/Delete)	O(1)	O(1)	O(n)
Stack	O(1)	O(1)	O(n)
Queue	O(1)	O(1)	O(n)
Hash Table	O(1)	O(n)	O(n)
Binary Search Tree	O(log n)	O(n)	O(n)
AVL Tree	O(log n)	O(log n)	O(n)
Red-Black Tree	O(log n)	O(log n)	O(n)

Data Structure / Algorithm	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
Heap	O(log n)	O(log n)	O(n)
Trie	O(m)	O(m)	O(n*m)
Sorting Algorithms			
Selection Sort	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	$O(n^2)$	$O(n^2)$	O(1)
Bubble Sort	$O(n^2)$	$O(n^2)$	O(1)
Merge Sort	O(n log n)	O(n log n)	O(n)
Quick Sort	O(n log n)	$O(n^2)$	O(log n)
Heap Sort	O(n log n)	O(n log n)	O(1)
Counting Sort	O(n + k)	O(n + k)	O(n + k)
Radix Sort	O(d * (n + k))	O(d * (n + k))	O(n + k)
Bucket Sort	O(n + k)	$O(n^2)$	O(n + k)
Searching Algorithms			
Linear Search	O(n)	O(n)	O(1)
Binary Search	O(log n)	O(log n)	O(1)
Interpolation Search	O(log log n)	O(n)	O(1)
Graph Algorithms			
BFS	O(V + E)	O(V + E)	O(V)
DFS	O(V + E)	O(V + E)	O(V)
Dijkstra's Algorithm	O((V + E) log V)	O((V + E) log V)	O(V)
Bellman-Ford	O(V * E)	O(V * E)	O(V)
Floyd-Warshall	$O(V^3)$	$O(V^3)$	$O(V^2)$
Kruskal's Algorithm	O(E log E)	O(E log E)	O(V + E)
Prim's Algorithm	O(E log V)	O(E log V)	O(V)
Topological Sort	O(V + E)	O(V + E)	O(V)

Note: V = number of vertices, E = number of edges, n = number of elements, m = length of strings, d = number of digits, k = range of values