Data Structures and Algorithms in Java

1. Introduction to Data Structures and Algorithms

Data Structures and Algorithms (DSA) are foundational concepts in computer science and software engineering, serving as essential tools for solving complex problems efficiently and effectively. Here are some key reasons why DSA is crucial:

Why we need DSA?

1. Optimized Problem Solving:
DSA provide efficient methods for organizing, storing, and manipulating data, which is essential for creating optimized solutions to a wide range of problems. Well-designed algorithms can drastically reduce execution time and resource usage.

2. Resource Management:
Efficient data structures and algorithms help manage computational resources such as time and memory. They enable programs to run faster and utilize less memory, making them essential for resource-constrained environments like embedded systems or high-performance computing.

3. Scalability:
As data sizes and problem complexities increase, the importance of DSA becomes more apparent. Without proper structures and algorithms, programs may become impractical or impossible to scale to handle larger datasets or more tasks that are complex.

4. Real-World Applications:
DSA are at the core of numerous real-world applications. From databases, search engines, and social networks to financial systems, robotics, and artificial intelligence, DSA enable the efficient processing of vast amounts of data and complex computations.

5. Algorithm Design and Analysis:
Studying DSA enhances your ability to design, analyze, and evaluate algorithms. This skill is vital for creating innovative solutions, optimizing existing systems, and making informed decisions when choosing between different approaches.

6. Problem Classification:
DSA provide a framework for categorizing and classifying problems. By understanding problem classes (e.g., sorting, searching, graph traversal), you can apply appropriate algorithms to solve them efficiently.

7. Interviews and Coding Challenges:
Many technical job interviews assess candidates' DSA knowledge. Proficiency in DSA can greatly enhance your performance in coding challenges and technical interviews, which are common parts of the hiring process in the tech industry.

8. Critical Thinking and Logical Reasoning:
DSA encourage you to think critically, break down problems into smaller components, and develop systematic solutions. These problem-solving skills are valuable not only in programming but also in various aspects of life.

9. Innovation:
Deep knowledge of DSA allows you to innovate and invent new algorithms or data structures to solve unique or emerging problems. Many breakthroughs in computer science and technology are driven by novel DSA concepts.

10. Continuous Learning:
The field of computer science is constantly evolving. New data structures and algorithms are developed, and existing ones are refined. Learning DSA equips you with the ability to adapt to changing technological landscapes.

In summary, Data Structures and Algorithms provide the fundamental building blocks for efficient and effective problem solving in computer science and software engineering. They enable the creation of optimized solutions, scalability, resource management, and innovation across a wide range of applications. Whether you are writing code for a simple application or developing complex software systems, a strong foundation in DSA is essential for success.

2. Arrays

- **Definition and Characteristics:**

Arrays are contiguous memory locations used to store a collection of elements of the same type. They offer constant-time access to elements but have fixed sizes.

- **Basic Operations and Example: Array Manipulation::**

Arrays are one of the simplest and most widely used data structures. They provide a way to store a collection of elements of the same data type in contiguous memory locations. Here are the basic operations associated with arrays:

1. Declaration and Initialization:
In Java, you can declare an array using square brackets `[]`. You must specify the data type and the size of the array. You can initialize an array when declaring it or later using assignment statements.

```java
// Declaration and initialization
int[] numbers = new int[5];  // Creates an integer array of size 5

// Initialization after declaration
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;
```

2. Accessing Array Elements:

Array elements are accessed using their index, which starts at 0 for the first element and increases by 1 for each subsequent element.

```java
int secondNumber = numbers[1];  // Accesses the second element (20)
```

3. Modifying Array Elements:

You can modify array elements by assigning new values to them using their index.

```java
numbers[2] = 35;  // Changes the third element's value to 35
```

4. Array Length:

The `length` property of an array gives you the number of elements it can hold. Note that the length is fixed once the array is created.

```java
int length = numbers.length;  // Gets the length of the array (5 in this case)
```

5. Iterating Through an Array:

You can use loops to iterate through array elements.

```java
for (int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);  // Prints each element
}
```

6. Multidimensional Arrays:
Arrays can have more than one dimension, forming matrices or higher-dimensional structures.

```java
int[][] matrix = new int[3][4];  // Creates a 3x4 matrix

matrix[1][2] = 42;  // Accesses and modifies an element in the matrix
```

7. Array Copying:
You can copy arrays using `System.arraycopy()` or using loops.

```java
int[] copy = new int[numbers.length];
System.arraycopy(numbers, 0, copy, 0, numbers.length);
```

8. Searching in an Array:
You can search for a specific element in an array using linear search or binary search (for sorted arrays).

9. Sorting an Array:
Sorting algorithms rearrange the elements of an array in a specific order (ascending or descending). Common sorting algorithms include bubble sort, selection sort, and insertion sort, merge sort, and quick sort.

```java
Arrays.sort(numbers);  // Sorts the array in ascending order
sort(numbers);  // Sorts the array in ascending order
```

10. Time and Space Complexity:
It is important to consider the time complexity (how long an operation takes) and space complexity (how much memory an operation uses) when working with arrays and their operations.

Arrays are fundamental building blocks in programming, and understanding their basic operations is crucial for any programmer. They form the basis for more complex data structures and algorithms, and mastery of array manipulation is essential for efficient problem solving.

3. Linked Lists

Linked lists consist of nodes, where each node holds data and a reference to the next node. They provide dynamic memory allocation and efficient insertion/deletion.

- **Singly Linked Lists:**

In a singly linked list, each node contains an element and a reference (or link) to the next node in the sequence. The last node points to null, indicating the end of the list. Singly linked lists are relatively simple and require less memory than doubly linked lists since they only have one reference per node.

- **Doubly Linked Lists:**

In a doubly linked list, each node contains an element, a reference to the next node, and a reference to the previous node. This bidirectional linking allows for traversal in both directions. However, doubly linked lists require more memory than singly linked lists due to the additional reference in each node.

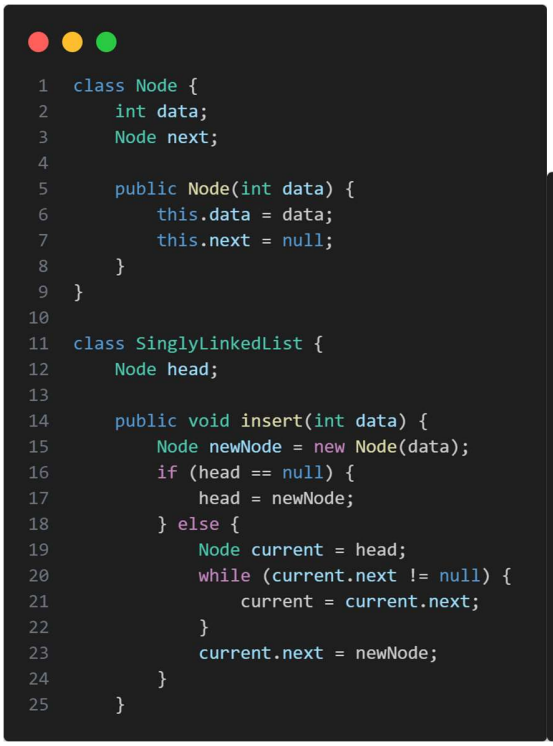
- **Circular Linked Lists:**

In a circular linked list, the last node of the list points back to the first node, forming a loop. Circular linked lists can be either singly or doubly linked. They can be useful for certain applications, such as implementing circular buffers.

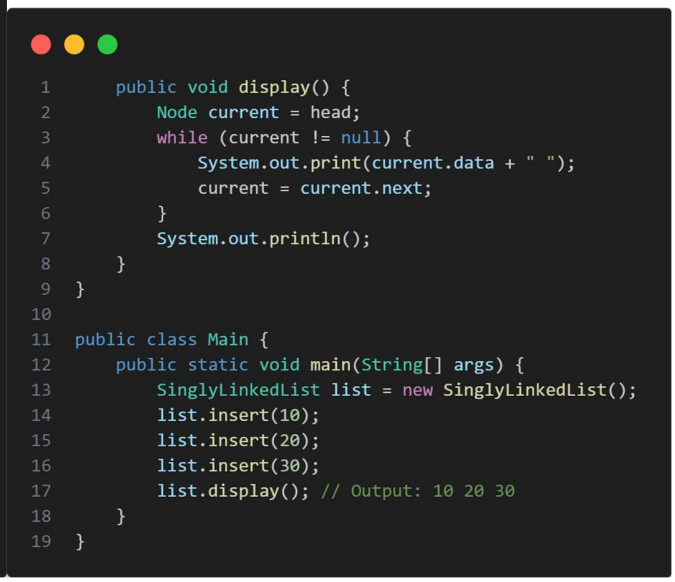
- **Single vs doubly vs Circular:**

Singly linked lists are memory-efficient and suitable for simple traversal.
Doubly linked lists provide bidirectional traversal but use more memory due to the extra reference.
Circular linked lists have no clear "end," making them useful for applications with cyclic patterns.
Here is a brief implementation example of a singly linked list in Java:

- **Example: Linked List Implementation:**



```java
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}

class SinglyLinkedList {
    Node head;

    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
    }
```



```java
    public void display() {
        Node current = head;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }
}

public class Main {
    public static void main(String[] args) {
        SinglyLinkedList list = new SinglyLinkedList();
        list.insert(10);
        list.insert(20);
        list.insert(30);
        list.display(); // Output: 10 20 30
    }
}
```

4. Stacks:

Stacks follow the Last-In-First-Out (LIFO) principle, while queues follow the First-In-First-Out (FIFO) principle. They are useful for managing data in a specific order.

- LIFO Principle:

The LIFO principle stands for "Last-In-First-Out," and it is a fundamental concept in computer science and data structures. It is often associated with the behavior of a data structure called a stack. In a LIFO arrangement, the last item added to the structure is the first one to be removed.

LIFO Principle and Stacks:

A stack is a linear data structure that follows the LIFO principle. It models a collection of elements with two main operations:

- Basic Operations:

1. **Push:** This operation adds an element to the top of the stack.
2. **Pop:** This operation removes and returns the top element from the stack.

Think of a physical stack of items, such as a stack of plates. You can only add or remove plates from the top of the stack. Similarly, in a stack data structure, the last element pushed onto the stack is the first one to be popped off.

Example:

Let us illustrate the LIFO principle with a simple example using a stack:

```java
import java.util.Stack;

public class LIFOPrincipleExample {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        // Push elements onto the stack
        stack.push(10);
        stack.push(20);
        stack.push(30);

        // Pop elements from the stack
        int popped1 = stack.pop(); // Removes 30
        int popped2 = stack.pop(); // Removes 20

        System.out.println("Popped elements: " + popped1 + ", " + popped2);
    }
}
```

In this example, the elements are added to the stack in the order 10, 20, 30. However, when we pop elements off the stack, they are removed in the reverse order: 30 is removed before 20.

Common Applications:
The LIFO principle has various real-world applications, including:

1. **Function Call Stack:** In programming, the function call stack uses the LIFO principle to manage the order of function calls and returns. When a function is called, its context is pushed onto the stack, and when the function returns, its context is popped off the stack.

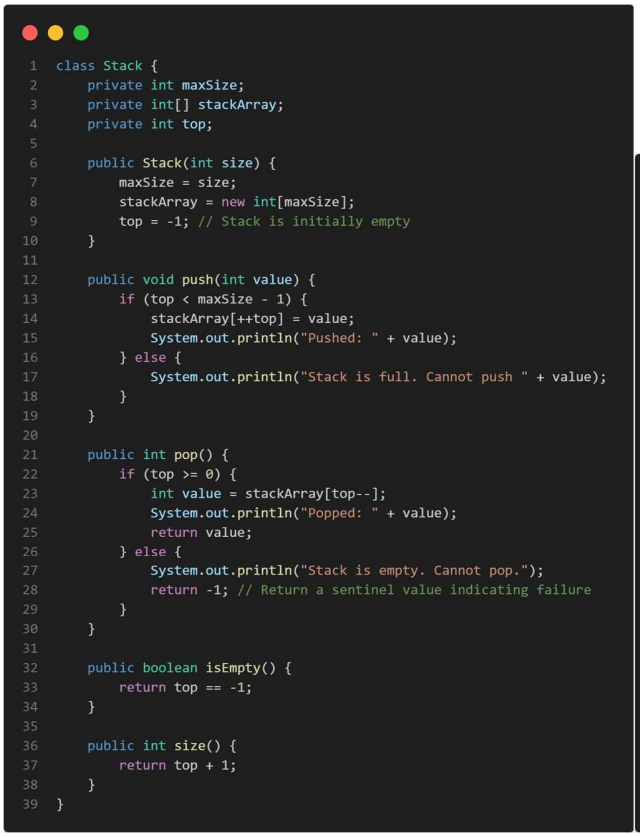
2. **Expression Evaluation:** In compiler and interpreter design, stacks are used to evaluate expressions and manage the order of operations.

3. **Undo/Redo Operations:** The LIFO principle is used to implement undo and redo functionalities in applications. Each action is pushed onto the undo stack, and redo stack, allowing for easy reversal of actions.

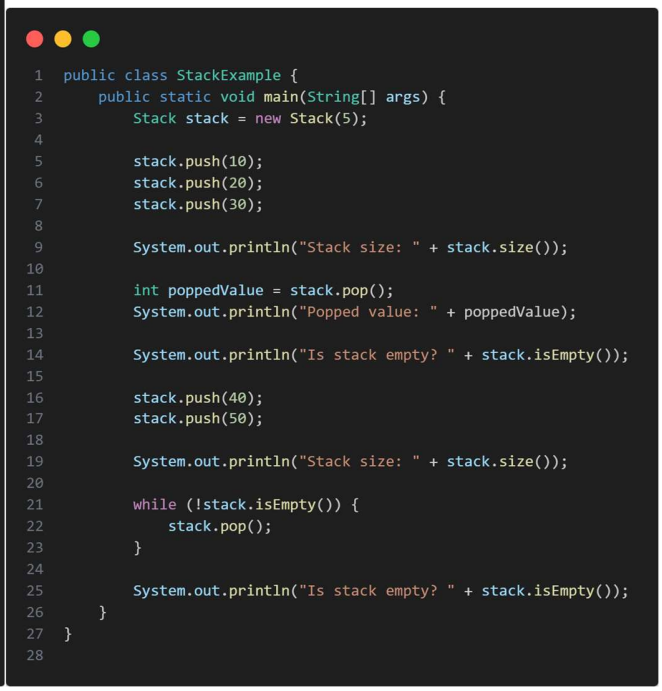
4. **Backtracking Algorithms:** Certain algorithms, like depth-first search, utilize the LIFO principle to backtrack and explore different paths.

The LIFO principle provides an elegant way to manage data and operations, and it is a fundamental concept that you will encounter in various aspects of computer science and programming.

- **Example: Implementing a Stack:**



```java
class Stack {
    private int maxSize;
    private int[] stackArray;
    private int top;

    public Stack(int size) {
        maxSize = size;
        stackArray = new int[maxSize];
        top = -1; // Stack is initially empty
    }

    public void push(int value) {
        if (top < maxSize - 1) {
            stackArray[++top] = value;
            System.out.println("Pushed: " + value);
        } else {
            System.out.println("Stack is full. Cannot push " + value);
        }
    }

    public int pop() {
        if (top >= 0) {
            int value = stackArray[top--];
            System.out.println("Popped: " + value);
            return value;
        } else {
            System.out.println("Stack is empty. Cannot pop.");
            return -1; // Return a sentinel value indicating failure
        }
    }

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

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



```java
public class StackExample {
    public static void main(String[] args) {
        Stack stack = new Stack(5);

        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println("Stack size: " + stack.size());

        int poppedValue = stack.pop();
        System.out.println("Popped value: " + poppedValue);

        System.out.println("Is stack empty? " + stack.isEmpty());

        stack.push(40);
        stack.push(50);

        System.out.println("Stack size: " + stack.size());

        while (!stack.isEmpty()) {
            stack.pop();
        }

        System.out.println("Is stack empty? " + stack.isEmpty());
    }
}
```

5. Queues:

Queues are another fundamental data structure that follows the First-In-First-Out (FIFO) principle. They are used to store and manage elements in a way that the first element added to the queue is the first one to be removed. Queues are commonly used in scenarios where tasks or data need to be processed in the order they are received.

- **FIFO Principle**

The FIFO principle, also known as "First-In-First-Out," is a fundamental concept in computer science and data structures. It refers to the order in which elements are processed or accessed. The element that is added or inserted first will be the first one to be removed or accessed. FIFO is commonly associated with data structures called queues.
A queue is a linear data structure that follows the FIFO principle. It models a collection of elements with two main operations:

1. **Enqueue:** This operation adds an element to the back (rear) of the queue.
2. **Dequeue:** This operation removes and returns the front element from the queue.

Think of a physical queue of people waiting in line. The person who arrives first is the one who gets served first. Similarly, in a queue data structure, the element that is enqueued first is the one that gets dequeued first.

- **Types of Queues (Linear, Circular, Priority)**

Certainly! Queues come in different variations to suit various scenarios and requirements. Here are the main types of queues: linear queues, circular queues, and priority queues.

1. Linear Queue:
A linear queue is the most basic type of queue. It follows the FIFO (First-In-First-Out) principle, where the first element added to the queue is the first one to be removed. Linear queues are used in scenarios where data needs to be processed in the order it arrives.
- Elements are added at the rear and removed from the front.
- Basic operations: Enqueue (adding), Dequeue (removing), IsEmpty, Size.
- Used in scenarios like task scheduling, printer job management, etc.
- Implemented using arrays or linked lists.

2. Circular Queue:
A circular queue, also known as a circular buffer or ring buffer, is an extension of the linear queue. In a circular queue, the last element is connected back to the first element, forming a circular structure. This allows efficient use of memory and helps in scenarios where the queue has a fixed size and needs to wrap around when it reaches its end.
- Elements are added at the rear and removed from the front.
- When the queue reaches its maximum capacity, it wraps around to the beginning.
- Provides efficient memory usage and is suitable for scenarios with cyclic patterns.
- Avoids wasting memory due to shifting elements in a fixed-size linear queue.
- Used in scenarios like buffer management, data streaming, and circular motion systems.

3. Priority Queue:
A priority queue is a type of queue where each element has an associated priority. Elements are removed from the queue based on their priority rather than their order of insertion. The element with the highest priority is dequeued first. Priority queues are often implemented using data structures like binary heaps.
- Elements are associated with priorities; higher-priority elements are dequeued first.
- Used in scenarios where processing order is based on priority levels.
- Common implementations include binary heaps (min-heap or max-heap), Fibonacci heaps, etc.
- Suitable for applications like scheduling tasks with varying levels of urgency.

- **Example: Queue Implementation:** Here is an example of implementing a queue in Java:

```java
import java.util.LinkedList;

class Queue {
    private LinkedList<Integer> list;

    public Queue() {
        list = new LinkedList<>();
    }

    public void enqueue(int value) {
        list.addLast(value);
        System.out.println("Enqueued: " + value);
    }

    public int dequeue() {
        if (!isEmpty()) {
            int value = list.removeFirst();
            System.out.println("Dequeued: " + value);
            return value;
        } else {
            System.out.println("Queue is empty. Cannot dequeue.");
            return -1; // Return a sentinel value indicating failure
        }
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

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

```java
public class QueueExample {
    public static void main(String[] args) {
        Queue queue = new Queue();

        queue.enqueue(10);
        queue.enqueue(20);
        queue.enqueue(30);

        System.out.println("Queue size: " + queue.size());

        int dequeuedValue = queue.dequeue();
        System.out.println("Dequeued value: " + dequeuedValue);

        System.out.println("Is queue empty? " + queue.isEmpty());

        queue.enqueue(40);
        queue.enqueue(50);

        System.out.println("Queue size: " + queue.size());

        while (!queue.isEmpty()) {
            queue.dequeue();
        }

        System.out.println("Is queue empty? " + queue.isEmpty());
    }
}
```

6. Trees:

Trees are hierarchical data structures with a root node and child nodes. Binary trees have at most two children per node. Binary Search Trees (BST) maintain a sorted order for efficient searching.

- **Binary Trees**

A binary tree is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. It is a fundamental concept in computer science and is used in various applications such as searching, sorting, hierarchical data representation, and more. Here is an overview of binary trees:

Basic Terminology:

- **Node:** Each element in a binary tree is called a node. A node contains data and references to its left and right children (or null if they don't exist).
- **Root:** The topmost node in a binary tree is called the root.
- **Leaf:** A node with no children is called a leaf node.
- **Parent and Children:** A node is considered the parent of its children, and its children are nodes that are directly connected to it.
- **Subtree:** A subtree is a tree that consists of a node and all its descendants.
- **Depth:** The depth of a node is the number of edges from the root to that node.
- **Height:** The height of a binary tree is the length of the longest path from the root to any leaf. It is the maximum depth of any node in the tree.

Types of Binary Trees:

1. **Full Binary Tree:** A binary tree in which every node has either 0 or 2 children.

2. Complete Binary Tree: A binary tree in which all levels are completely filled except possibly the last level, which is filled from left to right.
3. Perfect Binary Tree: A binary tree in which all internal nodes have exactly two children, and all leaf nodes are at the same level.
4. Balanced Binary Tree: A binary tree in which the height of the left and right subtrees of any node differ by at most one.
5. Degenerate (or Skewed) Tree: A tree in which each parent has only one child, creating a linear structure.

Common Operations on Binary Trees:
- **Insertion**: Adding a new node to the tree while maintaining the binary tree properties.
- **Deletion**: Removing a node from the tree while maintaining the binary tree properties.
- **Traversal**: Visiting all the nodes of the tree in a specific order.
  - Inorder (Left-Root-Right)
  - Preorder (Root-Left-Right)
  - Postorder (Left-Right-Root)
- **Searching**: Finding a specific node in the tree.

Here is a simple example of implementing a binary tree in Java:

```java
class Node {
    int data;
    Node left;
    Node right;

    public Node(int data) {
        this.data = data;
        this.left = null;
        this.right = null;
    }
}

public class BinaryTreeExample {
    public static void main(String[] args) {
        Node root = new Node(10);
        root.left = new Node(5);
        root.right = new Node(15);
        root.left.left = new Node(3);
        root.left.right = new Node(7);
        root.right.right = new Node(20);

        // Perform tree traversal
        System.out.println("Inorder Traversal:");
        inorderTraversal(root);

        System.out.println("\nPreorder Traversal:");
        preorderTraversal(root);

        System.out.println("\nPostorder Traversal:");
        postorderTraversal(root);
    }

```

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

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

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

- **Binary Search Trees**

A Binary Search Tree (BST) is a type of binary tree that follows a specific ordering property: for each node, all nodes in its left subtree have values less than the node's value, and all nodes in its right subtree have values greater than the node's value. This property makes searching, insertion, and

deletion operations efficient compared to regular binary trees. Here's an overview of Binary Search Trees:

BST Properties:
- Each node has a value (key) and can have at most two children: a left child and a right child.
- For any node:
  - All nodes in the left subtree have values less than the node's value.
  - All nodes in the right subtree have values greater than the node's value.
- This ordering property holds true for every node in the tree.

Operations on Binary Search Trees:
1. **Insertion:** Adding a new node while maintaining the BST property.
2. **Deletion:** Removing a node while maintaining the BST property.
3. **Search:** Finding a node with a specific value efficiently.
4. **Traversal:** Visiting all nodes in a specific order:
   - Inorder (Left-Root-Right)
   - Preorder (Root-Left-Right)
   - Postorder (Left-Right-Root)
5. **Finding Minimum and Maximum:** Finding the smallest and largest values in the tree.
6. **Successor and Predecessor:** Finding the next and previous values in the tree in terms of the order.
7. **Balancing:** Ensuring the tree remains balanced to maintain efficient operations.

Advantages of Binary Search Trees:
- Efficient searching: The binary search property allows for quick search operations with a time complexity of O(h), where h is the height of the tree.
- Efficient insertion and deletion: With proper balancing, insertion and deletion can also be achieved in O(h) time.
- Ordered data storage: BSTs maintain the order of the data elements, which can be useful for various applications.

Types of Balanced Binary Search Trees:
- **AVL Tree:** A self-balancing BST where the height difference between the left and right subtrees is limited to at most one.
- **Red-Black Tree:** Another self-balancing BST that ensures no path in the tree is more than twice as long as any other path.
- **Splay Tree:** A self-adjusting BST where recently accessed elements move closer to the root, improving future access times.
- **B-Tree:** A balanced tree designed for disk-based storage and database systems.

Here is a simple example of implementing a Binary Search Tree in Java:

```java
class Node {
    int key;
    Node left;
    Node right;

    public Node(int key) {
        this.key = key;
        this.left = null;
        this.right = null;
    }
}

public class BinarySearchTreeExample {
    static Node insert(Node root, int key) {
        if (root == null) {
            return new Node(key);
        }
        if (key < root.key) {
            root.left = insert(root.left, key);
        } else if (key > root.key) {
            root.right = insert(root.right, key);
        }
        return root;
    }
```

```java
static void inorderTraversal(Node root) {
    if (root != null) {
        inorderTraversal(root.left);
        System.out.print(root.key + " ");
        inorderTraversal(root.right);
    }
}

public static void main(String[] args) {
    Node root = null;
    int[] keys = {15, 10, 20, 8, 12, 17, 25};

    for (int key : keys) {
        root = insert(root, key);
    }

    System.out.println("Inorder Traversal:");
    inorderTraversal(root);

```

- AVL Trees

An AVL tree is a self-balancing binary search tree that maintains its balance by ensuring that the height difference between the left and right subtrees of any node (called the balance factor) is no greater than 1. This balance factor ensures that the tree remains relatively balanced, preventing skewed structures that can lead to inefficient search, insertion, and deletion operations. AVL trees are named after their inventors, Adelson-Velsky and Landis.

AVL Tree Properties:

- For any node in the tree, the heights of the left and right subtrees differ by at most 1.
- The balance factor of a node is calculated as the height of its left subtree minus the height of its right subtree.

Operations in AVL Trees:

1. **Rotation:** When an insertion or deletion causes an imbalance, rotations are performed to restore the balance. There are four types of rotations: left rotation, right rotation, left-right rotation (double rotation), and right-left rotation (double rotation).
2. **Insertion:** After inserting a node, the balance factor of its ancestors is checked, and rotations are performed if needed to maintain the AVL balance property.
3. **Deletion:** Similar to insertion, after deleting a node, the balance factor of ancestors is checked, and rotations are applied as necessary.
4. **Search, Traversal, and Other Operations:** AVL trees support the same operations as regular binary search trees, but with the added benefit of maintaining balance, ensuring logarithmic search and efficient insertion and deletion times.

Advantages of AVL Trees:

- Efficient: The height of an AVL tree is guaranteed to be logarithmic, ensuring efficient search, insertion, and deletion operations with a time complexity of O(log n).
- Predictable Performance: Unlike unbalanced binary search trees, AVL trees provide consistent performance for various operations.

Disadvantages:
- Overhead: Maintaining balance requires extra operations during insertion and deletion, making AVL trees slightly more complex and computationally intensive than regular binary search trees.
- Space: AVL trees consume more memory to store the balance factors or heights of nodes.

Here is an example of implementing an AVL tree in Java:

```java
class Node {
    int key;
    int height;
    Node left;
    Node right;

    public Node(int key) {
        this.key = key;
        this.height = 1;
        this.left = null;
        this.right = null;
    }
}

public class AVLTreeExample {
    static int height(Node node) {
        if (node == null) {
            return 0;
        }
        return node.height;
    }

    static int balanceFactor(Node node) {
        if (node == null) {
            return 0;
        }
        return height(node.left) - height(node.right);
    }

    static Node rightRotate(Node y) {
        Node x = y.left;
        Node T2 = x.right;

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

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

        return x;
    }
```

```java
    // Perform left rotation
    // ... (implementation similar to rightRotate)

    // Perform double rotation (left-right rotation)
    // ... (implementation using rotations)

    // Perform double rotation (right-left rotation)
    // ... (implementation using rotations)

    static Node insert(Node root, int key) {
        if (root == null) {
            return new Node(key);
        }
        if (key < root.key) {
            root.left = insert(root.left, key);
        } else if (key > root.key) {
            root.right = insert(root.right, key);
        } else {
            return root; // Duplicate keys not allowed
        }

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

        int balance = balanceFactor(root);

        // Perform rotations if needed to restore balance
        // ... (implementation of rotations)

        return root;
    }

    // ... (other operations like search, traversal, etc.)

    public static void main(String[] args) {
        Node root = null;
        int[] keys = {10, 5, 15, 3, 7, 12, 20};

        for (int key : keys) {
            root = insert(root, key);
        }

        // Perform AVL tree operations as needed
    }
}
```

7. Graphs:

Graphs consist of nodes and edges connecting these nodes. They can be used to represent complex relationships and are crucial in various applications.

- **Definitions and Concepts:**

A graph is a fundamental data structure used to represent relationships between different entities. It consists of a set of vertices (also called nodes) and a set of edges that connect pairs of vertices. Graphs are widely used in various fields, including computer science, mathematics, social networks, transportation systems, and more. Here are the key definitions and concepts related to graphs:

Basic Definitions:

- **Vertex (Node):** Represents an entity or element. In a graph, vertices can have labels or values.
- **Edge:** Represents a connection between two vertices. An edge can be directed (pointing from one vertex to another) or undirected (bi-directional).
- **Degree:** The degree of a vertex is the number of edges incident on it. For directed graphs, the in-degree is the number of incoming edges, and the out-degree is the number of outgoing edges.
- **Adjacent Vertices:** Two vertices are adjacent if there is an edge connecting them.
- **Path:** A sequence of vertices where each vertex is adjacent to the next one.
- **Cycle:** A path that starts and ends at the same vertex.
- **Connected Graph:** A graph in which there is a path between every pair of vertices.
- **Disconnected Graph:** A graph that is not connected. It consists of multiple connected components.
- **Weighted Graph:** A graph in which each edge has a weight or cost associated with it.
- **Directed Graph (Digraph):** A graph in which edges have a direction, indicating a one-way connection.
- **Undirected Graph:** A graph in which edges do not have a direction, indicating a two-way connection.

- **Graph Representation (Adjacency Matrix, Adjacency List):**

1. **Adjacency Matrix:** A 2D matrix where each cell represents whether there is an edge between two vertices. It can be used for both directed and undirected graphs.
2. **Adjacency List:** A collection of linked lists (or arrays) where each list represents the vertices adjacent to a particular vertex.

Types of Graphs:

1. **Directed Acyclic Graph (DAG):** A directed graph with no cycles.
2. **Tree:** A connected, undirected graph with no cycles.
3. **Forest:** A collection of disjoint trees.
4. **Complete Graph:** A graph where there is an edge between every pair of distinct vertices.
5. **Bipartite Graph:** A graph whose vertices can be divided into two disjoint sets, with edges only connecting vertices from different sets.

Graph Algorithms:

1. **Graph Traversal:**
   - Breadth-First Search (BFS): Visits all vertices at the same level before moving to the next level.
   - Depth-First Search (DFS): Explores as far as possible along one branch before backtracking.

2. **Shortest Path Algorithms:**
   - Dijkstra's Algorithm: Finds the shortest paths from a single source vertex to all other vertices.
   - Bellman-Ford Algorithm: Computes shortest paths even in graphs with negative-weight edges.
3. **Minimum Spanning Tree Algorithms:**

- Prim's Algorithm: Builds a minimum spanning tree by adding edges from a starting vertex.
- Kruskal's Algorithm: Constructs a minimum spanning tree by adding edges with the smallest weights.

4. Topological Sorting: Arranges the vertices of a directed acyclic graph in a linear order based on their dependencies.

5. Cycle Detection: Determines whether a graph contains cycles. These are just some of the key definitions, concepts, and algorithms related to graphs. Graphs are a rich area of study with a wide range of applications, and understanding their properties and algorithms is essential for solving complex problems in various domains.

- **Depth-First Search (DFS) Algorithm:**

Depth-First Search (DFS) is a graph traversal algorithm that explores vertices and their edges as deeply as possible before backtracking. It's a fundamental algorithm used to traverse and search graphs, trees, and other data structures. DFS is often implemented using recursion or an explicit stack data structure. Here's how the DFS algorithm works:

Algorithm Steps:

1. Start at a selected source vertex (or node).
2. Mark the source vertex as visited.
3. Explore an unvisited adjacent vertex of the current vertex.
4. If an unvisited adjacent vertex is found, repeat steps 2 and 3 for that vertex.
5. If no unvisited adjacent vertex is found, backtrack to the previous vertex.
6. Repeat steps 3 to 5 until all vertices are visited.

DFS Implementation (Recursive):

Here is a simple implementation of the DFS algorithm using recursion:

```java
import java.util.*;

class Graph {
    private int V; // Number of vertices
    private LinkedList<Integer>[] adjList;

    public Graph(int vertices) {
        V = vertices;
        adjList = new LinkedList[V];
        for (int i = 0; i < V; i++) {
            adjList[i] = new LinkedList<>();
        }
    }

    public void addEdge(int v, int w) {
        adjList[v].add(w);
    }

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

        for (Integer neighbor : adjList[vertex]) {
            if (!visited[neighbor]) {
                DFS(neighbor, visited);
            }
        }
    }
```

```java
    public void performDFS(int startVertex) {
        boolean[] visited = new boolean[V];
        DFS(startVertex, visited);
    }
}

public class DFSExample {
    public static void main(String[] args) {
        Graph graph = new Graph(7);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.addEdge(2, 5);
        graph.addEdge(2, 6);

        System.out.println("Depth-First Traversal (starting from vertex 0):");
        graph.performDFS(0);
    }
}
```

In this example, the `Graph` class represents the graph using an adjacency list. The `DFS` method is the recursive DFS traversal implementation. The `performDFS` method initiates the DFS traversal from a specified starting vertex.

DFS can also be implemented iteratively using a stack data structure. The key idea is to push adjacent unvisited vertices onto the stack and pop vertices off the stack when there are no more unvisited neighbors.

DFS is useful for applications like pathfinding, connectivity analysis, topological sorting, and cycle detection. It's important to note that DFS doesn't guarantee the shortest path in unweighted graphs and can get stuck in infinite loops if not properly implemented or if cycles are present.

- **Breadth-First Search (BFS) Algorithm:**

Breadth-First Search (BFS) is another graph traversal algorithm that explores vertices and their edges level by level. It visits all vertices at the same level before moving on to the next level. BFS is often used to find the shortest path between two vertices in an unweighted graph. It is implemented using a queue data structure. Here's how the BFS algorithm works:

Algorithm Steps:

1. Start at a selected source vertex (or node).
2. Enqueue the source vertex into a queue and mark it as visited.
3. While the queue is not empty:
   - Dequeue a vertex from the queue.
   - Process the vertex (print it, save it, etc.).
   - Enqueue all unvisited adjacent vertices of the dequeued vertex and mark them as visited.

BFS Implementation:

Here is a simple implementation of the BFS algorithm using a queue:

```java
import java.util.*;

class Graph {
    private int V; // Number of vertices
    private LinkedList<Integer>[] adjList;

    public Graph(int vertices) {
        V = vertices;
        adjList = new LinkedList[V];
        for (int i = 0; i < V; i++) {
            adjList[i] = new LinkedList<>();
        }
    }

    public void addEdge(int v, int w) {
        adjList[v].add(w);
    }

    public void BFS(int startVertex) {
        boolean[] visited = new boolean[V];
        Queue<Integer> queue = new LinkedList<>();

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

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

            for (Integer neighbor : adjList[vertex]) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                }
            }
        }
    }
}
```

```java
public class BFSExample {
    public static void main(String[] args) {
        Graph graph = new Graph(7);

        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.addEdge(2, 5);
        graph.addEdge(2, 6);

        System.out.println("Breadth-First Traversal (starting from vertex 0):");
        graph.BFS(0);
    }
}
```

In this example, the `Graph` class and the adjacency list are similar to the previous example. The `BFS` method implements the BFS traversal using a queue.

BFS is useful for applications like finding the shortest path in unweighted graphs, network traversal, web crawling, and more. It guarantees that the shortest path is found when the graph is unweighted because it explores vertices level by level.

8. Sorting Algorithms

Sorting algorithms arrange data in a specific order, while searching algorithms find a particular element in a dataset.

- **Bubble Sort:**

Bubble Sort is a simple comparison-based sorting algorithm that repeatedly steps through the list of elements to be sorted, compares adjacent elements, and swaps them if they are in the wrong order. The algorithm gets its name because smaller elements "bubble" to the top of the list while larger elements "sink" to the bottom. Bubble Sort is not very efficient for large datasets, but it is easy to understand and implement. Here's how the Bubble Sort algorithm works:

Algorithm Steps:

1. Start from the beginning of the list.
2. Compare the current element with the next element.
3. If the current element is greater than the next element, swap them.
4. Move to the next pair of elements and repeat steps 2 and 3.
5. Continue these steps until the end of the list is reached.
6. After completing one pass, the largest element is guaranteed to be at the end of the list.
7. Repeat steps 1 to 6 for the remaining elements, excluding the already sorted ones.
8. Repeat the process until no swaps are needed, indicating that the list is sorted.

Bubble Sort Implementation:

Here's a simple implementation of the Bubble Sort algorithm in Java:

```java
public class BubbleSortExample {
    static void bubbleSort(int[] array) {
        int n = array.length;
        boolean swapped;

        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
                if (array[j] > array[j + 1]) {
                    // Swap array[j] and array[j+1]
                    int temp = array[j];
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                    swapped = true;
                }
            }

            // If no two elements were swapped in the inner loop, the array is sorted
            if (!swapped) {
                break;
            }
        }
    }
```

```java
    public static void main(String[] args) {
        int[] arr = {64, 34, 25, 12, 22, 11, 90};
        System.out.println("Original array: " + Arrays.toString(arr));

        bubbleSort(arr);

        System.out.println("Sorted array: " + Arrays.toString(arr));
    }
}
```

In this example, the `bubbleSort` function implements the Bubble Sort algorithm. It iterates through the array and swaps adjacent elements if they are out of order. The process continues until the entire array is sorted. The outer loop controls the number of passes, and the inner loop performs comparisons and swaps within each pass.

Bubble Sort has a time complexity of O(n^2) in the worst and average cases, making it inefficient for large datasets. Other sorting algorithms like Merge Sort, Quick Sort, and Heap Sort generally provide better performance for larger inputs.

- **Selection Sort:**

Selection Sort is a simple comparison-based sorting algorithm that works by dividing the input array into two parts: the sorted part and the unsorted part. The algorithm repeatedly finds the minimum (or maximum, depending on the sorting order) element from the unsorted part and swaps it with the first element of the unsorted part. As a result, the sorted part grows, and the unsorted part shrinks. Selection Sort is not very efficient for large datasets, but it is straightforward to understand and implement. Here's how the Selection Sort algorithm works:

Algorithm Steps:

1. Find the minimum (or maximum) element in the unsorted part of the array.
2. Swap the minimum element with the first element of the unsorted part.
3. Move the boundary between the sorted and unsorted parts one element to the right.
4. Repeat steps 1 to 3 until the entire array is sorted.

Selection Sort Implementation:

Here is a simple implementation of the Selection Sort algorithm in Java:

```java
import java.util.Arrays;

public class SelectionSortExample {
    static void selectionSort(int[] array) {
        int n = array.length;

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

            // Find the index of the minimum element in the unsorted part
            for (int j = i + 1; j < n; j++) {
                if (array[j] < array[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap the found minimum element with the first element of the unsorted part
            int temp = array[minIndex];
            array[minIndex] = array[i];
            array[i] = temp;
        }
    }

    public static void main(String[] args) {
        int[] arr = {64, 25, 12, 22, 11};
        System.out.println("Original array: " + Arrays.toString(arr));

        selectionSort(arr);

        System.out.println("Sorted array: " + Arrays.toString(arr));
    }
}
```

In this example, the `selectionSort` function implements the Selection Sort algorithm. It iterates through the array and finds the minimum element in the unsorted part, then swaps it with the first element of the unsorted part. The boundary between the sorted and unsorted parts is moved one element to the right in each iteration.

Selection Sort also has a time complexity of O(n^2) in the worst and average cases, making it inefficient for large datasets. Similar to Bubble Sort, other sorting algorithms like Merge Sort, Quick Sort, and Heap Sort offer better performance for larger inputs.

- **Insertion Sort:**

Insertion Sort is a simple comparison-based sorting algorithm that builds the final sorted array one element at a time. It is well-suited for small datasets or partially sorted arrays. The algorithm maintains a "sorted" portion of the array and repeatedly inserts the next element from the unsorted portion into its correct position within the sorted portion. Insertion Sort is efficient for small inputs and has some advantages for nearly sorted arrays. Here's how the Insertion Sort algorithm works.

Algorithm Steps:

1. Start with the first element as the "sorted" portion of the array.
2. Take the next element from the unsorted portion and compare it with elements in the sorted portion, moving larger elements to the right.
3. Insert the current element into its correct position within the sorted portion.
4. Repeat steps 2 and 3 until all elements are in the sorted portion.

Insertion Sort Implementation:

Here is a simple implementation of the Insertion Sort algorithm in Java:

```java
import java.util.Arrays;

public class InsertionSortExample {
    static void insertionSort(int[] array) {
        int n = array.length;

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

            // Move elements of the sorted portion that are greater than key to the right
            while (j >= 0 && array[j] > key) {
                array[j + 1] = array[j];
                j--;
            }

            // Insert the key into its correct position
            array[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        int[] arr = {64, 25, 12, 22, 11};
        System.out.println("Original array: " + Arrays.toString(arr));

        insertionSort(arr);

        System.out.println("Sorted array: " + Arrays.toString(arr));
    }
}
```

In this example, the `insertionSort` function implements the Insertion Sort algorithm. It starts with the second element and repeatedly shifts larger elements to the right to make space for the current element to be inserted at its correct position.

Insertion Sort has a time complexity of O(n^2) in the worst and average cases. It performs well for small inputs or partially sorted arrays but becomes inefficient for larger datasets. For larger datasets, other sorting algorithms like Merge Sort, Quick Sort, and Heap Sort are more suitable due to their better average-case performance.

- **Merge Sort:**

Merge Sort is a widely used comparison-based sorting algorithm that employs a divide-and-conquer strategy to efficiently sort an array or a list of elements. It works by dividing the input into smaller halves, recursively sorting them, and then merging the sorted halves to produce the final sorted result. Merge Sort is known for its stable performance and guaranteed time complexity. Here's how the Merge Sort algorithm works:

Algorithm Steps:

1. Divide the unsorted array into two halves.
2. Recursively sort the left and right halves.
3. Merge the two sorted halves to produce a single sorted array.

Merge Sort Implementation:

Here is a simple implementation of the Merge Sort algorithm in Java:

```java
import java.util.Arrays;

public class MergeSortExample {
    static void merge(int[] array, int left, int middle, int right) {
        int n1 = middle - left + 1;
        int n2 = right - middle;

        int[] leftArray = new int[n1];
        int[] rightArray = new int[n2];

        for (int i = 0; i < n1; i++) {
            leftArray[i] = array[left + i];
        }
        for (int j = 0; j < n2; j++) {
            rightArray[j] = array[middle + 1 + j];
        }

        int i = 0, j = 0, k = left;
        while (i < n1 && j < n2) {
            if (leftArray[i] <= rightArray[j]) {
                array[k] = leftArray[i];
                i++;
            } else {
                array[k] = rightArray[j];
                j++;
            }
            k++;
        }
```

```java
        while (i < n1) {
            array[k] = leftArray[i];
            i++;
            k++;
        }

        while (j < n2) {
            array[k] = rightArray[j];
            j++;
            k++;
        }
    }

    static void mergeSort(int[] array, int left, int right) {
        if (left < right) {
            int middle = left + (right - left) / 2;

            mergeSort(array, left, middle);
            mergeSort(array, middle + 1, right);

            merge(array, left, middle, right);
        }
    }

    public static void main(String[] args) {
        int[] arr = {64, 25, 12, 22, 11};
        System.out.println("Original array: " + Arrays.toString(arr));

        mergeSort(arr, 0, arr.length - 1);

        System.out.println("Sorted array: " + Arrays.toString(arr));
    }
}
```

In this example, the `mergeSort` function implements the Merge Sort algorithm. It divides the array into smaller halves and recursively sorts them using the `merge` function, which merges two sorted subarrays into a single sorted array.

Merge Sort has a time complexity of O(n log n) in the worst, average, and best cases, making it a reliable choice for sorting large datasets. It is stable (maintains the relative order of equal elements) and performs consistently well regardless of the input distribution.

- **Quick Sort:**

Quick Sort is a widely used comparison-based sorting algorithm that employs a divide-and-conquer strategy to efficiently sort an array or a list of elements. It works by selecting a "pivot" element from the array and partitioning the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted. Quick Sort is known for its efficiency and is often used in practice. Here's how the Quick Sort algorithm works:

Algorithm Steps:

1. Choose a pivot element from the array (typically the first or last element).
2. Partition the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.
3. Recursively apply Quick Sort to the sub-arrays.
4. Combine the sorted sub-arrays and the pivot to produce the final sorted array.

Quick Sort Implementation:

Here is a simple implementation of the Quick Sort algorithm in Java:

```java
import java.util.Arrays;

public class QuickSortExample {
    static int partition(int[] array, int low, int high) {
        int pivot = array[high];
        int i = low - 1;

        for (int j = low; j < high; j++) {
            if (array[j] < pivot) {
                i++;

                int temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
        int temp = array[i + 1];
        array[i + 1] = array[high];
        array[high] = temp;

        return i + 1;
    }

    static void quickSort(int[] array, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(array, low, high);

            quickSort(array, low, pivotIndex - 1);
            quickSort(array, pivotIndex + 1, high);
        }
    }

    public static void main(String[] args) {
        int[] arr = {64, 25, 12, 22, 11};
        System.out.println("Original array: " + Arrays.toString(arr));

        quickSort(arr, 0, arr.length - 1);

        System.out.println("Sorted array: " + Arrays.toString(arr));
    }
}
```

In this example, the `partition` function implements the partitioning step of the Quick Sort algorithm. It selects a pivot and rearranges elements so that those less than the pivot are on the left and those greater are on the right. The `quickSort` function applies the Quick Sort algorithm recursively to the sub-arrays.

Quick Sort has an average and best-case time complexity of O(n log n), making it efficient for sorting large datasets. However, its worst-case time complexity is O(n^2), which can be mitigated by choosing a good pivot strategy (e.g., selecting the median). Quick Sort is also not stable, meaning that it may change the relative order of equal elements.

9. Searching Algorithms:

Searching algorithms are methods used to locate specific elements within a data structure, such as an array, list, or tree. These algorithms are fundamental in computer science and are essential for finding specific values efficiently. Here are some common searching algorithms:

```
int[] arr = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};
```

1. Linear Search:

- Linear Search is a simple and straightforward method where each element in the array is checked sequentially until the target element is found or the end of the array is reached.
- **Time Complexity:** O(n)
- Suitable for small arrays or unsorted data.

```
int target = 23;
int index = -1;

for (int i = 0; i < arr.length; i++) {
    if (arr[i] == target) {
        index = i;
        break;
    }
}

if (index != -1) {
    System.out.println("Linear Search: Element found at index " + index);
} else {
    System.out.println("Linear Search: Element not found");
}
```

2. Binary Search:

- Binary Search is used on sorted arrays. It repeatedly divides the search interval in half, comparing the middle element to the target value. Depending on the comparison, the search continues in the left or right half.
- **Time Complexity:** O(log n)
- Efficient for large datasets but requires the data to be sorted.

```java
int target = 23;
int low = 0;
int high = arr.length - 1;
int index = -1;

while (low <= high) {
    int mid = low + (high - low) / 2;

    if (arr[mid] == target) {
        index = mid;
        break;
    } else if (arr[mid] < target) {
        low = mid + 1;
    } else {
        high = mid - 1;
    }
}

if (index != -1) {
    System.out.println("Binary Search: Element found at index " + index);
} else {
    System.out.println("Binary Search: Element not found");
}
```

3. Hashing:

- Hashing uses a hash function to map elements to specific positions (buckets) in a hash table. When searching, the hash function is used to quickly locate the bucket containing the desired element.
- **Time Complexity:** Average case O(1), worst case O(n) due to collisions
- Provides fast searching and insertion but may have collisions that need to be resolved.

4. Interpolation Search:

- Interpolation Search is an optimization of Binary Search, particularly for uniformly distributed data. It estimates the position of the target element based on the values of the endpoints and interpolates to find the correct position.
- **Time Complexity:** O(log log n) for uniformly distributed data, O(n) in the worst case

5. Exponential Search:

- Exponential Search is used when the size of the array is not known. It starts with a small range and then increases the range exponentially until it encompasses the target value. After that, a binary search is performed.
- **Time Complexity:** O(log n)

6. Jump Search:

- Jump Search is used on sorted arrays. It divides the array into blocks of fixed size and jumps through these blocks to find the one that contains the target value. After locating the block, a linear search is performed within it.
- **Time Complexity: O(√n)**

7. Fibonacci Search:

- Fibonacci Search is a hybrid of Binary and Fibonacci sequence. It uses Fibonacci numbers to define the search intervals and performs comparisons similarly to Binary Search.
- Time Complexity: O(log n)

8. Ternary Search:

- Ternary Search is an extension of Binary Search. Instead of dividing the array into two parts, it divides it into three parts and compares the target element with elements at one-third and two-thirds positions.
- **Time Complexity: O(log3 n):**

These searching algorithms have different characteristics and are suitable for various scenarios. The choice of which algorithm to use depends on factors such as data size, data distribution, and whether the data is sorted.

10. Hashing

Hashing involves mapping data to an array index using a hash function. It allows for quick data retrieval. Hashing is a technique used to map data of arbitrary size to fixed-size values, typically integers, known as hash codes or hash values. It's commonly used for data storage, retrieval, and quick data lookup. Hashing is widely used in data structures like hash tables, which provide fast access to elements based on their keys.

Here is how hashing works:

1. Hash Function:

A hash function takes an input (or key) and produces a hash code that is typically an integer. The hash code should be deterministic, meaning that the same input will always produce the same hash code. An ideal hash function distributes hash codes uniformly to minimize collisions.

2. Bucket (Slot) Array:

A bucket array, often called a hash table, is an array where data is stored. The hash code serves as an index to determine the location (bucket or slot) in the array where the data will be stored or retrieved.

3. Collision Handling:

Collisions occur when two different inputs produce the same hash code. Collision handling strategies are used to resolve this issue:

- Separate Chaining: Each bucket contains a linked list or another data structure to hold multiple elements that share the same hash code.

- Open Addressing: In case of collision, the algorithm searches for the next available slot in the array to store the data.

- Double Hashing: A secondary hash function is used to calculate the next slot to check in case of collision.

Hashing Example:

Let's say we have a hash table with 10 slots and a simple hash function that takes the remainder when dividing the key by 10. We will store some names in the hash table.

```java
class HashTable {
    private String[] table;
    private int size;

    public HashTable(int size) {
        this.size = size;
        this.table = new String[size];
    }

    public int hash(String key) {
        int sum = 0;
        for (char c : key.toCharArray()) {
            sum += c;
        }
        return sum % size;
    }

    public void insert(String key, String value) {
        int index = hash(key);
        table[index] = value;
    }

```

```java
public String get(String key) {
        int index = hash(key);
        return table[index];
    }
}

public class HashingExample {
    public static void main(String[] args) {
        HashTable hashTable = new HashTable(10);

        hashTable.insert("John", "555-1234");
        hashTable.insert("Jane", "555-5678");
        hashTable.insert("Alice", "555-9876");

        System.out.println("John's phone number: " + hashTable.get("John"));
        System.out.println("Alice's phone number: " + hashTable.get("Alice"));
        System.out.println("Jane's phone number: " + hashTable.get("Jane"));
    }
}
```

In this example, the `HashTable` class demonstrates a basic implementation of a hash table. The `hash` method calculates the hash code, and the `insert` and `get` methods store and retrieve data using the calculated hash codes.

Hashing is widely used in various applications, including databases, caches, cryptography, and more, due to its efficiency in data retrieval and storage.

- **Hash Tables:**

A hash table, also known as a hash map, is a data structure that implements an associative array abstract data type, a structure that can map keys to values. Hash tables use a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. Hash tables offer efficient insertion, deletion, and search operations, making them widely used for implementing data structures like dictionaries, caches, and sets. Here's how hash tables work:

Components of a Hash Table:

1. **Array:** The core of the hash table, which is an array of buckets or slots, where data is stored.

2. **Hash Function:** This function takes a key as input and produces a hash code (or hash value) that maps to an index in the array. It should distribute keys uniformly to minimize collisions.

3. Collision Resolution: Collisions occur when two different keys produce the same hash code. Collision resolution techniques are used to handle this situation and ensure correct storage and retrieval of data.

Basic Operations:

1. Insertion: Given a key and a value, the hash function produces the hash code. The hash code determines the index where the value is stored in the array.

2. Search (Get): Given a key, the hash function produces the hash code, and the value can be retrieved from the corresponding index in the array.

3. Deletion: Given a key, the hash function produces the hash code, and the value can be removed from the corresponding index in the array.

Collision Resolution Techniques:

1. Separate Chaining: Each bucket contains a linked list or another data structure to handle multiple values that map to the same index.

2. Open Addressing: In case of a collision, the algorithm searches for the next available slot in the array to store the data. Common methods include linear probing and quadratic probing.

3. Double Hashing: A secondary hash function is used to calculate the next slot to check in case of a collision.

Hash Table Example:
Here is a simple example of a hash table implementation in Java:

```java
class HashTable {
    private int size;
    private String[] data;

    public HashTable(int size) {
        this.size = size;
        this.data = new String[size];
    }

    private int hash(String key) {
        int hash = 0;
        for (char c : key.toCharArray()) {
            hash += c;
        }
        return hash % size;
    }

    public void put(String key, String value) {
        int index = hash(key);
        data[index] = value;
    }
```

```java
    public String get(String key) {
        int index = hash(key);
        return data[index];
    }
}

public class HashTableExample {
    public static void main(String[] args) {
        HashTable hashTable = new HashTable(10);

        hashTable.put("John", "555-1234");
        hashTable.put("Jane", "555-5678");

        System.out.println("John's phone number: " + hashTable.get("John"));
        System.out.println("Jane's phone number: " + hashTable.get("Jane"));
    }
}
```

In this example, the `HashTable` class implements a basic hash table. The `hash` function calculates the hash code, and the `put` and `get` methods handle insertion and retrieval of data using the hash code as the index.

Hash tables provide efficient average-case time complexity for insertion, search, and deletion operations. However, they require careful consideration of the hash function to ensure good distribution and collision resolution strategy to maintain efficiency.

- **Collision Handling (Chaining, Open Addressing):**

Collision handling is an essential aspect of hash tables since collisions occur when two different keys produce the same hash code. Collision resolution techniques are used to manage and organize the storage of multiple values that map to the same hash code. Two common collision resolution techniques are chaining and open addressing.

1. Chaining:

Chaining involves using a data structure (often a linked list or another hash table) to hold multiple values that map to the same index in the main hash table array. Each bucket in the main array contains a reference to the head of the linked list, and when collisions occur, new elements are simply appended to the list.

Chaining is relatively simple to implement and works well when the hash table size is sufficiently large. It handles an arbitrary number of collisions per bucket efficiently.

Chaining Example:

```java
class HashTable {
    private int size;
    private LinkedList<Pair>[] buckets;

    public HashTable(int size) {
        this.size = size;
        this.buckets = new LinkedList[size];
        for (int i = 0; i < size; i++) {
            buckets[i] = new LinkedList<>();
        }
    }

    private int hash(String key) {
        // hash function implementation
    }

    public void put(String key, String value) {
        int index = hash(key);
        buckets[index].add(new Pair(key, value));
    }
```

```java
public String get(String key) {
        int index = hash(key);
        for (Pair pair : buckets[index]) {
            if (pair.key.equals(key)) {
                return pair.value;
            }
        }
        return null;
    }

    private class Pair {
        String key;
        String value;

        Pair(String key, String value) {
            this.key = key;
            this.value = value;
        }
    }
}
```

2. Open Addressing:

Open addressing involves placing collided elements directly into the array itself rather than using an auxiliary data structure. When a collision occurs, the algorithm searches for the next available slot in the array using a predetermined sequence (linear probing, quadratic probing, etc.). The key idea is to ensure that every slot is probed, so there's no need for additional data structures like linked lists.

Open addressing has better cache performance compared to chaining and is suitable for situations where memory usage is a concern.

Open Addressing Example:

```java
class HashTable {
    private int size;
    private String[] data;

    public HashTable(int size) {
        this.size = size;
        this.data = new String[size];
    }

    private int hash(String key) {
        // hash function implementation
    }

    public void put(String key, String value) {
        int index = hash(key);

        // Linear probing
        while (data[index] != null) {
            index = (index + 1) % size;
        }
```

```java
data[index] = value;
    }

    public String get(String key) {
        int index = hash(key);

        // Linear probing
        while (data[index] != null && !data[index].equals(key)) {
            index = (index + 1) % size;
        }

        return data[index];
    }
}
```

Both chaining and open addressing have their pros and cons. Chaining is simpler and can handle an arbitrary number of collisions, but it requires additional memory for the linked lists. Open addressing saves memory and has better cache performance but might lead to clustering and increased search time under certain conditions. The choice between the two depends on factors such as memory usage, performance requirements, and expected collision frequency.

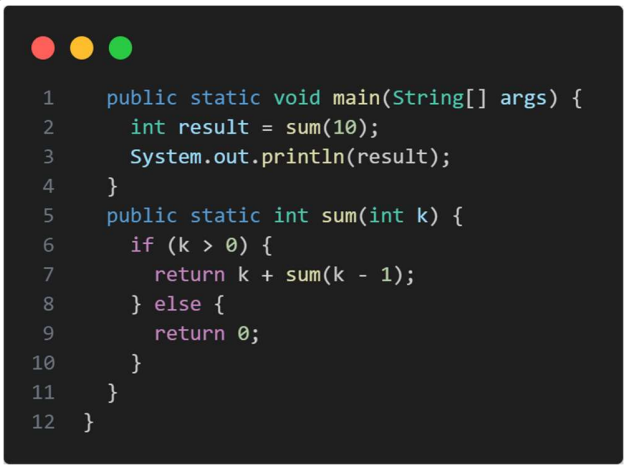
11. Recursion

Recursion involves solving a problem by breaking it down into smaller instances of the same problem.

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Here is a simple example of a Recursion implementation in Java:



```java
public static void main(String[] args) {
  int result = sum(10);
  System.out.println(result);
}
public static int sum(int k) {
  if (k > 0) {
    return k + sum(k - 1);
  } else {
    return 0;
  }
}
}
```

- **Recursive vs. Iterative Approaches:**

Recursive and iterative approaches are two different strategies for solving problems in programming. Each has its advantages and drawbacks, and the choice between them often depends on the specific problem, coding style, and language preference. Let's compare recursive and iterative approaches:

Recursive Approach:

1. Concept: In a recursive approach, a function calls itself to solve a problem by breaking it down into smaller, similar subproblems. It uses the divide-and-conquer strategy.

2. Clarity: Recursive code can often be more elegant and closer to the problem's description, making it easier to understand for some complex problems.

3. Complexity: Recursive code can be simpler to write for problems with a recursive nature, like traversing tree structures (e.g., binary trees, graphs) or handling recursive mathematical formulas.

4. Memory: Recursive calls are maintained on the call stack, which can lead to stack overflow errors for deeply nested recursion. Tail recursion (a specific form of recursion where the recursive call is the last operation in the function) can be optimized by some compilers or interpreters to use a constant amount of stack space.

5. Performance: Recursive functions can be less efficient in terms of performance and memory usage compared to their iterative counterparts, particularly for deeply nested recursion.

Iterative Approach:

1. Concept: In an iterative approach, a problem is solved using loops, and there's no self-referencing of functions. It involves using variables and loop constructs like `for` and `while`.

2. Clarity: Iterative code can be more straightforward to understand for problems that naturally involve iteration, such as looping through arrays or performing iterative calculations.

3. Complexity: Iterative code is often used when you need precise control over the flow of execution and when you don't want to rely on function calls for problem solving.

4. Memory: Iterative code typically uses a constant amount of memory for its variables, making it more memory-efficient than deep recursion.

5. Performance: Iterative approaches often have better performance than recursive ones due to reduced function call overhead and better memory management.

When to Choose Recursion:

Use recursion when the problem can be naturally divided into smaller, similar subproblems.

- Use recursion when code readability and elegance are essential for understanding and maintaining the solution.

- Be cautious when dealing with deeply nested recursion to avoid stack overflow errors.

When to Choose Iteration:

- Use iteration for problems that involve loops, repetitive calculations, or where precise control over program flow is necessary.

- Use iteration when performance and memory efficiency are critical, especially for large datasets or deeply nested scenarios.

- Consider iterative solutions when you want to avoid the overhead of function calls and the call stack.

Ultimately, the choice between recursion and iteration depends on the specific problem and the trade-offs between code clarity, performance, and memory usage. In some cases, a combination of both techniques can be the most effective solution.

- Example: Recursive Factorial Calculation:

```java
public class RecursiveFactorial {
    public static int factorial(int n) {
        // Base case: when n is 0 or 1, the factorial is 1
        if (n == 0 || n == 1) {
            return 1;
        } else {
            // Recursive case: n! = n * (n-1)!
            return n * factorial(n - 1);
        }
    }

    public static void main(String[] args) {
        int number = 5;
        int result = factorial(number);
        System.out.println("Factorial of " + number + " is " + result);
    }
}
```

In this Java program:

- The `factorial` function takes an integer `n` as input.
- It has a base case: when `n` is 0 or 1, it returns 1 because the factorial of 0 and 1 is 1.
- In the recursive case, if `n` is greater than 1, the function calls itself with `n - 1` and multiplies the result by `n`. This recursive call continues until `n` reaches the base case.

When you run this Java program, it will calculate and print the factorial of 5, which is 120.

Remember that when using recursion, it is essential to have a base case that terminates the recursion to avoid infinite recursion.

12. Dynamic Programming

Dynamic Programming involves breaking down a problem into smaller sub problems and solving each sub problem only once, storing the solutions to avoid redundant calculations.

Dynamic programming (DP) is a powerful optimization technique used in computer science and mathematics for solving problems by breaking them down into smaller subproblems and reusing solutions to those subproblems. DP is particularly useful for solving problems with overlapping subproblems, which means that the same subproblem is solved multiple times in a naive (inefficient) recursive approach. By storing and reusing the results of subproblems, DP significantly improves the efficiency of solving these problems.

Here are some key concepts and principles of dynamic programming:

1. Optimal Substructure: Many problems can be solved by breaking them down into smaller subproblems, and the optimal solution to the overall problem can be constructed from optimal solutions to these subproblems.

2. Overlapping Subproblems: In some problems, subproblems are solved multiple times with the same inputs. DP stores the results of subproblems in a data structure (usually an array or a table) to avoid redundant calculations.

3. Memoization (Top-Down): Memoization is a technique where the results of subproblems are stored in a data structure (often a dictionary or an array) and are checked before solving a subproblem. If the result is already computed, it's retrieved instead of recalculating it.

4. Tabulation (Bottom-Up): Tabulation is another DP technique where solutions to subproblems are computed iteratively, starting from the simplest subproblems and gradually building up to the desired solution. Tabulation typically uses a table (array) to store results.

5. State Transition: DP problems often involve defining a recurrence relation or state transition equation that relates the solution to a larger problem with solutions to smaller subproblems. This equation forms the basis for solving the problem.

6. Choice and Optimization: Some DP problems involve making choices at each step to maximize or minimize an objective function (e.g., maximize profit, minimize cost). DP helps find the optimal choices at each step.

Common examples of problems that can be solved using dynamic programming include:

- Fibonacci sequence calculation
- Longest common subsequence
- Coin change problem
- Knapsack problem
- Shortest path problems (e.g., Dijkstra's algorithm)
- Matrix chain multiplication
- Edit distance (Levenshtein distance) calculation
- Rod cutting problem

Here's a simplified example of dynamic programming in Java, computing the nth Fibonacci number using Memoization:

```java
import java.util.HashMap;
import java.util.Map;

public class Fibonacci {
    private static Map<Integer, Integer> memo = new HashMap<>();

    public static int fibonacci(int n) {
        if (n <= 1) {
            return n;
        }

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

        int result = fibonacci(n - 1) + fibonacci(n - 2);
        memo.put(n, result);
        return result;
    }

    public static void main(String[] args) {
        int n = 10;
        int result = fibonacci(n);
        System.out.println("Fibonacci(" + n + ") = " + result);
    }
}
```

In this example, memoization is used to store intermediate results, making the computation of Fibonacci numbers much more efficient, especially for larger values of `n`. Dynamic programming is a versatile technique used to solve a wide range of problems in computer science and beyond.

13. Greedy Algorithms

Greedy algorithms make locally optimal choices at each step to find a globally optimal solution.

Greedy algorithms are a class of algorithms used to make a series of decisions that optimize for the current best choice at each step, with the hope that this will lead to a globally optimal solution. The greedy strategy is to make the locally optimal choice at each step, without worrying about its impact on future steps. While greedy algorithms can be simple and efficient, they may not always provide the best overall solution, as they don't consider the entire problem space.

Key characteristics of greedy algorithms:

1. Greedy Choice Property: A greedy algorithm makes a series of choices, and at each step, it selects the option that appears to be the best at that moment. This choice is based on the current state of the problem and doesn't consider future consequences.

2. Optimal Substructure: The overall problem can be divided into subproblems, and the optimal solution to the original problem can be constructed from optimal solutions to these subproblems. In other words, a greedy choice at one step should not invalidate future steps.

3. Greedy Algorithms Do not Always Guarantee Optimality: While greedy algorithms often provide good approximate solutions quickly, they do not guarantee that the result will be globally optimal. In some cases, a locally optimal choice at each step might lead to a suboptimal overall solution.

Common examples of problems that can be solved using greedy algorithms include:

- **Knapsack Problem:** Given a set of items, each with a weight and a value, determine the maximum value that can be obtained by selecting a subset of the items while not exceeding a given weight limit.

The Knapsack Problem is a classic optimization problem in computer science and mathematics. It is commonly encountered in various fields, including operations research, economics, and computer science. The problem can be described as follows:

Problem Statement:

Given a set of items, each with a weight and a value, determine the maximum value that can be obtained by selecting a subset of the items while not exceeding a given weight limit (the capacity of the knapsack).

The Knapsack Problem can be divided into two main variations:

1. 0/1 Knapsack Problem: In this variation, each item can be either selected (included in the knapsack) or rejected (not included). You cannot take a fractional part of an item.

2. Fractional Knapsack Problem: In this variation, you can take a fractional part of an item, which means that you can choose a portion of an item based on its weight and value.

Dynamic Programming Solution (0/1 Knapsack):

One of the most widely used techniques to solve the 0/1 Knapsack Problem is dynamic programming. The dynamic programming approach involves constructing a table (often a 2D array) where each cell represents the maximum value that can be obtained with a specific combination of items and a specific knapsack capacity.

Here is a step-by-step explanation of the dynamic programming solution for the 0/1 Knapsack Problem:

1. Create a 2D array `dp` where `dp[i][j]` represents the maximum value that can be obtained with the first `i` items when the knapsack capacity is `j`.

2. Initialize the first row and first column of the `dp` array to be all zeros because, with no items or no capacity, the maximum value is zero.

3. Iterate through the items and capacity:

   - For each item `i` and capacity `j`, calculate two values:

     - `dp[i][j]` when item `i` is not included in the knapsack (equal to `dp[i-1][j]`).

     - `dp[i][j]` when item `i` is included in the knapsack (equal to `value[i] + dp[i-1][j - weight[i]]` if `weight[i] <= j`).

   - Update `dp[i][j]` to be the maximum of the two calculated values.

4. The value in the bottom-right cell of the `dp` array (`dp[n][W]`, where `n` is the number of items and `W` is the knapsack capacity) represents the maximum value that can be obtained with the given items and capacity.

5. To find out which items were selected, backtrack through the `dp` array, starting from the bottom-right cell, and determine which items were included in the optimal solution.

Here is a Java implementation of the 0/1 Knapsack Problem using dynamic programming:

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

        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= capacity; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0;
                } else if (weights[i - 1] <= j) {
                    dp[i][j] = Math.max(dp[i - 1][j], values[i - 1] + dp[i - 1][j - weights[i - 1]]);
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        return dp[n][capacity];
    }

    public static void main(String[] args) {
        int[] values = {60, 100, 120};
        int[] weights = {10, 20, 30};
        int capacity = 50;

        int maxValue = knapsack(values, weights, capacity);
        System.out.println("Maximum value: " + maxValue);
    }
}
```

In this example, the `knapsack` function calculates the maximum value that can be obtained by selecting items from the given arrays of values and weights while respecting the knapsack's capacity. The `dp` array is used to store intermediate results.

- **Minimum Spanning Tree (MST):** Find the subset of edges in a weighted graph that connects all vertices while minimizing the total edge weight. Algorithms like Kruskal's and Prim's are greedy approaches to this problem.

- **Dijkstra's Shortest Path:** Find the shortest path from a starting node to all other nodes in a weighted graph. Dijkstra's algorithm uses a greedy strategy by repeatedly selecting the vertex with the smallest tentative distance from the start.

- **Huffman Coding:** Given a set of characters and their frequencies, construct a binary tree in a way that minimizes the total length of encoded characters.

- **Activity Selection:** Given a set of activities with start and finish times, find the maximum number of activities that do not overlap.

The Activity Selection Problem is a classic problem in algorithm design that deals with a set of activities, each with a start time and an end time. The objective is to select the maximum number of non-overlapping activities that can be scheduled without conflicts, given that each activity can only be executed in its entirety.

Here is a step-by-step explanation of the greedy algorithm used to solve the Activity Selection Problem:

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

class Activity {
    int startTime;
    int endTime;

    Activity(int startTime, int endTime) {
        this.startTime = startTime;
        this.endTime = endTime;
    }
}

public class ActivitySelection {
    public static List<Activity> selectActivities(List<Activity> activities) {
        List<Activity> selectedActivities = new ArrayList<>();

        // Sort activities based on finish times
        Collections.sort(activities, Comparator.comparingInt(a -> a.endTime));

        // Select the first activity
        selectedActivities.add(activities.get(0));

        // Iterate through the remaining activities
        for (int i = 1; i < activities.size(); i++) {
            Activity currentActivity = activities.get(i);
            Activity lastSelectedActivity = selectedActivities.get(selectedActivities.size() - 1);

            // If the current activity can be scheduled without overlapping, select it
            if (currentActivity.startTime >= lastSelectedActivity.endTime) {
                selectedActivities.add(currentActivity);
            }
        }

        return selectedActivities;
    }

    public static void main(String[] args) {
        List<Activity> activities = new ArrayList<>();
        activities.add(new Activity(1, 4));
        activities.add(new Activity(3, 5));
        activities.add(new Activity(0, 6));
        activities.add(new Activity(5, 7));
        activities.add(new Activity(3, 8));
        activities.add(new Activity(5, 9));
        activities.add(new Activity(6, 10));
        activities.add(new Activity(8, 11));

        List<Activity> selectedActivities = selectActivities(activities);

        System.out.println("Selected Activities:");
        for (Activity activity : selectedActivities) {
            System.out.println("[" + activity.startTime + ", " + activity.endTime + "]");
        }
    }
}
```

1. Sort the activities: First, sort the activities based on their finish times in ascending order. Sorting makes it easier to identify the optimal set of activities.

2. Select the first activity: The first activity in the sorted list will always be selected because it has the earliest finish time.

3. Iterate through the remaining activities: Starting from the second activity, iterate through the sorted list of activities. At each step, compare the start time of the current activity with the finish time of the previously selected activity. If the start time of the current activity is greater than or equal to the finish time of the previously selected activity, add the current activity to the selected set.

4. Repeat step 3: Continue this process until all activities have been considered or until no more activities can be added to the selected set.

5. The selected set of activities is the solution: The activities in the selected set represent the maximum number of non-overlapping activities that can be scheduled.

Here's a Java implementation of the Activity Selection Problem using the greedy algorithm:

In this example, we first sort the activities by finish time and then use the greedy approach to select the maximum number of non-overlapping activities. The selected activities are printed as the output.

Greedy algorithms are often used when the problem exhibits the greedy choice property, and it is computationally expensive or impractical to explore all possible solutions. However, they should be used with caution, as they may not always produce the best possible solution. In some cases, dynamic programming or other optimization techniques might be necessary to find the globally optimal solution.

- **Example: Fractional Knapsack**

The Fractional Knapsack Problem is a variation of the classic Knapsack Problem where you can take fractions of items to maximize the total value within the knapsack's weight capacity. Here's a step-by-step explanation and a Java implementation of the Fractional Knapsack Problem:

Problem Statement:
Given a set of items, each with a weight and a value, and a knapsack with a maximum weight capacity, determine the maximum value that can be obtained by selecting a fraction of the items to fit within the knapsack.

Greedy Algorithm for Fractional Knapsack:
1. Calculate the value-to-weight ratio for each item: `value[i] / weight[i]` for each item `i`.
2. Sort the items based on their value-to-weight ratios in descending order. This step ensures that you consider items with the highest value-to-weight ratios first.
3. Initialize the total value of the knapsack, `totalValue`, to 0.
4. Iterate through the sorted items:
   - If the current item can fit entirely into the knapsack (i.e., `weight[i] <= remainingCapacity`), add its value to `totalValue`, subtract its weight from `remainingCapacity`, and mark the entire item as selected.
   - If the current item cannot fit entirely, calculate the fraction of the item that can fit into the knapsack, add its proportional value to `totalValue`, and break the loop (as no more items can be added).
5. Return `totalValue` as the maximum achievable value.

Here is the Java implementation of the Fractional Knapsack Problem:

```java
import java.util.Arrays;
import java.util.Comparator;

class Item {
    int value;
    int weight;
    double valueToWeightRatio;

    public Item(int value, int weight) {
        this.value = value;
        this.weight = weight;
        this.valueToWeightRatio = (double) value / weight;
    }
}

public class FractionalKnapsack {
    public static double fractionalKnapsack(int capacity, Item[] items) {
        // Sort items by value-to-weight ratio in descending order
        Arrays.sort(items, Comparator.comparingDouble((Item item) -> item.valueToWeightRatio).reversed());

        double totalValue = 0.0;
        int remainingCapacity = capacity;

        for (Item item : items) {
            if (remainingCapacity >= item.weight) {
                // Entire item can be added to the knapsack
                totalValue += item.value;
                remainingCapacity -= item.weight;
            } else {
                // Fraction of the item can be added
                totalValue += item.valueToWeightRatio * remainingCapacity;
                break; // No more items can be added
            }
        }

        return totalValue;
    }

    public static void main(String[] args) {
        Item[] items = {
            new Item(60, 10),
            new Item(100, 20),
            new Item(120, 30)
        };

        int capacity = 50;

        double maxValue = fractionalKnapsack(capacity, items);
        System.out.println("Maximum achievable value: " + maxValue);
    }
}
```

In this example, we have a knapsack with a capacity of 50 and three items with their respective values and weights. The `fractionalKnapsack` function calculates the maximum achievable value using the greedy algorithm and fractional item selection.

14. Conclusion

Data Structures and Algorithms are crucial for efficient problem solving and form the foundation of computer science. Mastery of these concepts is essential for any software developer.

This report provides a comprehensive overview of various DSA concepts using Java. Each section includes explanations and code examples to aid understanding. It is intended to serve as a starting point for further exploration and practical implementation of DSA concepts in Java.

Project: Sudoku solver(backtracking)

Layout – index.html

```html
<div id="container">
  <h1 class="padd">Sudoku Solver</h1>
  <table id="sudoku-board">
    <colgroup>
      <col>
      <col>
      <col>
      <colgroup>
        <col>
        <col>
        <col>
        <colgroup>
          <col>
          <col>
          <col>
        <tbody>
      <tr>
        <td contenteditable="true"></td>
        <td contenteditable="true"></td>
        <td contenteditable="true"></td>
        <td contenteditable="true"></td>
        <td contenteditable="true"></td>
        <td contenteditable="true"></td>
        <td contenteditable="true"></td>
        <td contenteditable="true"></td>
        <td contenteditable="true"></td>
    <tr>
      .......
```

```html
    <div>
      <button id="solve-button">Solve!</button>
    </div>
    <div>
      <button id="clear-button">Clear board</button>
    </div>
</div>
```

Scripts –index.html

```
<script type="text/javascript">
  document.getElementById("sudoku-board").addEventListener("keyup", function (event) {
    if (event.target && event.target.nodeName == "TD") {
      var validNum = /[1-9]/;
      var tdEl = event.target;
      if (tdEl.innerText.length > 0 && validNum.test(tdEl.innerText[0])) {
        tdEl.innerText = tdEl.innerText[0];
      } else {
        tdEl.innerText = "";
      }
    }
  });

  document.getElementById("solve-button").addEventListener("click", function (event) {
    var boardString = boardToString();
    var solution = SudokuSolver.solve(boardString);
    if (solution) {
      stringToBoard(solution);
    } else {
      alert("Invalid board!");
    }
  })

  document.getElementById("clear-button").addEventListener("click", clearBoard);

  function clearBoard() {
    var tds = document.getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
      tds[i].innerText = "";
    }
  }

  function boardToString() {
    var string = "";
    var validNum = /[1-9]/;
    var tds = document.getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
      if (validNum.test(tds[i].innerText[0])) {
        string += tds[i].innerText;
      } else {
        string += "-";
      }
    }
    return string;
  }

  function stringToBoard(string) {
    var currentCell;
    var validNum = /[1-9]/;
    var cells = string.split("");
    var tds = document.getElementsByTagName("td");
    for (var i = 0; i < tds.length; i++) {
      currentCell = cells.shift();
      if (validNum.test(currentCell)) {
        tds[i].innerText = currentCell;
      }
    }
  }
</script>
```

CSS- style.css

```css
body {
    font-family: monospace;
}

#container {
    text-align: center;
}

table {
    border-collapse: collapse;
    font-size: 2em;
    margin: 0 auto;
}

colgroup,
tbody {
    border: solid medium;
}

td {
    border: solid thin;
    height: 1.4em;
    width: 1.4em;
    text-align: center;
    padding: 0;
}

button {
    margin-top: 15px;
    font-size: 1.5em;
}

padd {
    padding-bottom: 100px;
}
```

JavaScript: sudoku.js

```
"use strict";

var EASY_PUZZLE = "1-58-2----9--764-52--4--819-19--73-6762-83-9-----61-5---76---3-43--2-5-16--3-89--";
var MEDIUM_PUZZLE = "-3-5--8-45-42---1---8--9---79-8-61-3-----54---5------78-----7-2---7-46--61-3--5--";
var HARD_PUZZLE = "8----------36------7--9-2---5---7-------457-----1---3---1----68--85---1--9----4--";

// Set this variable to true to publicly expose otherwise private functions inside of SudokuSolver
var TESTABLE = true;

var SudokuSolver = function (testable) {
  var solver;

  // PUBLIC FUNCTIONS
  function solve(boardString) {
    var boardArray = boardString.split("");
    if (boardIsInvalid(boardArray)) {
      return false;
    }
    return recursiveSolve(boardString);
  }

  function solveAndPrint(boardString) {
    var solvedBoard = solve(boardString);
    console.log(toString(solvedBoard.split("")));
    return solvedBoard;
  }

  // PRIVATE FUNCTIONS
  function recursiveSolve(boardString) {
    var boardArray = boardString.split("");
    if (boardIsSolved(boardArray)) {
      return boardArray.join("");
    }
    var cellPossibilities = getNextCellAndPossibilities(boardArray);
    var nextUnsolvedCellIndex = cellPossibilities.index;
    var possibilities = cellPossibilities.choices;
    for (var i = 0; i < possibilities.length; i++) {
      boardArray[nextUnsolvedCellIndex] = possibilities[i];
      var solvedBoard = recursiveSolve(boardArray.join(""));
      if (solvedBoard) {
        return solvedBoard;
      }
    }
    return false;
```

```
}

function boardIsInvalid(boardArray) {
  return !boardIsValid(boardArray);
}

function boardIsValid(boardArray) {
  return allRowsValid(boardArray) && allColumnsValid(boardArray) && allBoxesValid(boardArray);
}

function boardIsSolved(boardArray) {
  for (var i = 0; i < boardArray.length; i++) {
    if (boardArray[i] === "-") {
      return false;
    }
  }
  return true;
}

function getNextCellAndPossibilities(boardArray) {
  for (var i = 0; i < boardArray.length; i++) {
    if (boardArray[i] === "-") {
      var existingValues = getAllIntersections(boardArray, i);
      var choices = ["1", "2", "3", "4", "5", "6", "7", "8", "9"].filter(function (num) {
        return existingValues.indexOf(num) < 0;
      });
      return { index: i, choices: choices };
    }
  }
}

function getAllIntersections(boardArray, i) {
  return getRow(boardArray, i).concat(getColumn(boardArray, i)).concat(getBox(boardArray, i));
}

function allRowsValid(boardArray) {
  return [0, 9, 18, 27, 36, 45, 54, 63, 72].map(function (i) {
    return getRow(boardArray, i);
  }).reduce(function (validity, row) {
    return collectionIsValid(row) && validity;
  }, true);
}

function getRow(boardArray, i) {
  var startingEl = Math.floor(i / 9) * 9;
```

```
  return boardArray.slice(startingEl, startingEl + 9);
}

function allColumnsValid(boardArray) {
  return [0, 1, 2, 3, 4, 5, 6, 7, 8].map(function (i) {
    return getColumn(boardArray, i);
  }).reduce(function (validity, row) {
    return collectionIsValid(row) && validity;
  }, true);
}

function getColumn(boardArray, i) {
  var startingEl = Math.floor(i % 9);
  return [0, 1, 2, 3, 4, 5, 6, 7, 8].map(function (num) {
    return boardArray[startingEl + num * 9];
  });
}

function allBoxesValid(boardArray) {
  return [0, 3, 6, 27, 30, 33, 54, 57, 60].map(function (i) {
    return getBox(boardArray, i);
  }).reduce(function (validity, row) {
    return collectionIsValid(row) && validity;
  }, true);
}

function getBox(boardArray, i) {
  var boxCol = Math.floor(i / 3) % 3;
  var boxRow = Math.floor(i / 27);
  var startingIndex = boxCol * 3 + boxRow * 27;
  return [0, 1, 2, 9, 10, 11, 18, 19, 20].map(function (num) {
    return boardArray[startingIndex + num];
  });
}

function collectionIsValid(collection) {
  var numCounts = {};
  for (var i = 0; i < collection.length; i++) {
    if (collection[i] != "-") {
      if (numCounts[collection[i]] === undefined) {
        numCounts[collection[i]] = 1;
      } else {
        return false;
      }
    }
```

```
    }
    return true;
  }

  function toString(boardArray) {
    return [0, 9, 18, 27, 36, 45, 54, 63, 72].map(function (i) {
      return getRow(boardArray, i).join(" ");
    }).join("\n");
  }

  if (testable) {
    // These methods will be exposed publicly when testing is on.
    solver = {
      solve: solve,
      solveAndPrint: solveAndPrint,
      recursiveSolve: recursiveSolve,
      boardIsInvalid: boardIsInvalid,
      boardIsValid: boardIsValid,
      boardIsSolved: boardIsSolved,
      getNextCellAndPossibilities: getNextCellAndPossibilities,
      getAllIntersections: getAllIntersections,
      allRowsValid: allRowsValid,
      getRow: getRow,
      allColumnsValid: allColumnsValid,
      getColumn: getColumn,
      allBoxesValid: allBoxesValid,
      getBox: getBox,
      collectionIsValid: collectionIsValid,
      toString: toString
    };
  } else {
    // These will be the only public methods when testing is off.
    solver = {
      solve: solve,
      solveAndPrint: solveAndPrint
    };
  }

  return solver;
}(TESTABLE);
```

Output:

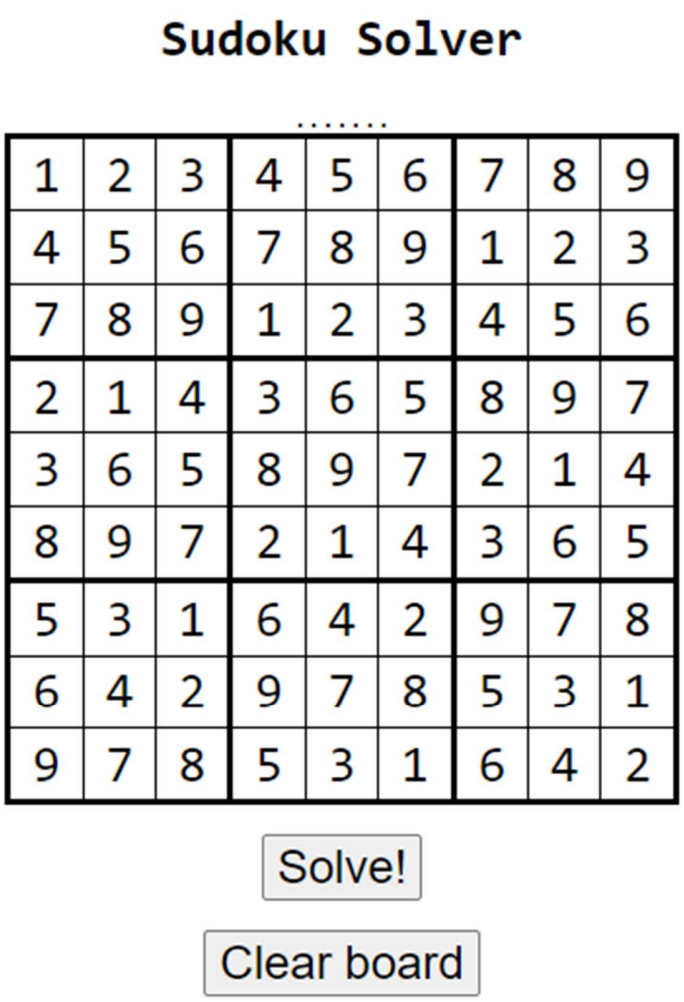
Sudoku Solver

.....

1	2	3	4	5	6	7	8	9
4	5	6	7	8	9	1	2	3
7	8	9	1	2	3	4	5	6
2	1	4	3	6	5	8	9	7
3	6	5	8	9	7	2	1	4
8	9	7	2	1	4	3	6	5
5	3	1	6	4	2	9	7	8
6	4	2	9	7	8	5	3	1
9	7	8	5	3	1	6	4	2

Solve!

Clear board



15. References:

- Geeksforgeeks
- w3schoools
- YouTube