DSA Fundamentals in Java - Complete Guide

Table of Contents

Why DSA Matters

Data Structures and Algorithms are fundamental concepts that help you:

- Write efficient code that runs faster and uses less memory
- Solve complex problems systematically
- Pass technical interviews at top companies
- Become a better programmer overall

Java Basics for DSA

Essential Java Concepts

Before diving into DSA, ensure you understand these Java fundamentals:

Classes and Objects

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

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

Arrays

```java
// Declaration and initialization
int[] arr = new int[5];
int[] nums = {1, 2, 3, 4, 5};

// 2D Arrays
int[][] matrix = new int[3][3];
```

Generics

```java
public class GenericStack<T> {
    private T[] stack;
    private int top;

    public void push(T item) {
        stack[++top] = item;
    }
}
```

Time and Space Complexity

Big O Notation

Understanding algorithm efficiency is crucial:

Time Complexity Examples:

- O(1) - Constant: Accessing array element
- O(log n) - Logarithmic: Binary search
- O(n) - Linear: Linear search
- O(n log n) - Linearithmic: Merge sort
- $O(n^2)$ - Quadratic: Bubble sort
- O(2^n) - Exponential: Recursive fibonacci

Space Complexity:

- Additional memory used by algorithm
- Includes auxiliary space and input space

```java
// O(1) space
public int findMax(int[] arr) {
    int max = arr[0];
    for (int i = 1; i < arr.length; i++) {
        if (arr[i] > max) max = arr[i];
    }
    return max;
}

// O(n) space due to recursion stack
public int factorial(int n) {
    if (n <= 1) return 1;
    return n * factorial(n - 1);
}
```

Core Data Structures

1. Arrays

Advantages: Fast access, cache-friendly **Disadvantages:** Fixed size, expensive insertion/deletion

```java
public class DynamicArray {
    private int[] arr;
    private int size;
    private int capacity;

    public DynamicArray() {
        capacity = 2;
        arr = new int[capacity];
        size = 0;
    }

    public void add(int element) {
        if (size == capacity) {
            resize();
        }
        arr[size++] = element;
    }

    private void resize() {
        capacity *= 2;
        int[] newArr = new int[capacity];
        System.arraycopy(arr, 0, newArr, 0, size);
        arr = newArr;
    }
}
```

2. Linked Lists

Advantages: Dynamic size, efficient insertion/deletion **Disadvantages:** No random access, extra memory for pointers

```java
public class LinkedList {
    private Node head;

    private class Node {
        int data;
        Node next;

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

    public void addFirst(int data) {
        Node newNode = new Node(data);
        newNode.next = head;
        head = newNode;
    }

    public void addLast(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
            return;
        }

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

    public boolean search(int key) {
        Node current = head;
        while (current != null) {
            if (current.data == key) return true;
            current = current.next;
        }
        return false;
    }
}
```

3. Stacks

LIFO (Last In, First Out) - Think of a stack of plates

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

    public Stack(int size) {
        maxSize = size;
        arr = new int[maxSize];
        top = -1;
    }

    public void push(int value) {
        if (isFull()) {
            throw new RuntimeException("Stack overflow");
        }
        arr[++top] = value;
    }

    public int pop() {
        if (isEmpty()) {
            throw new RuntimeException("Stack underflow");
        }
        return arr[top--];
    }

    public int peek() {
        if (isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        return arr[top];
    }

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

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

4. Queues

FIFO (First In, First Out) - Think of a line at a store

```java
public class Queue {
    private int[] arr;
    private int front, rear, size, capacity;

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

    public void enqueue(int value) {
        if (isFull()) {
            throw new RuntimeException("Queue overflow");
        }
        rear = (rear + 1) % capacity;
        arr[rear] = value;
        size++;
    }

    public int dequeue() {
        if (isEmpty()) {
            throw new RuntimeException("Queue underflow");
        }
        int value = arr[front];
        front = (front + 1) % capacity;
        size--;
        return value;
    }

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

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

5. Trees

Binary Tree Implementation:

```java
public class BinaryTree {
    private TreeNode root;

    private class TreeNode {
        int val;
        TreeNode left, right;

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

    // In-order traversal (Left, Root, Right)
    public void inorderTraversal(TreeNode node) {
        if (node != null) {
            inorderTraversal(node.left);
            System.out.print(node.val + " ");
            inorderTraversal(node.right);
        }
    }

    // Pre-order traversal (Root, Left, Right)
    public void preorderTraversal(TreeNode node) {
        if (node != null) {
            System.out.print(node.val + " ");
            preorderTraversal(node.left);
            preorderTraversal(node.right);
        }
    }

    // Post-order traversal (Left, Right, Root)
    public void postorderTraversal(TreeNode node) {
        if (node != null) {
            postorderTraversal(node.left);
            postorderTraversal(node.right);
            System.out.print(node.val + " ");
        }
    }
}
```

Binary Search Tree:

java

```java
public class BST {
    private TreeNode root;

    public void insert(int val) {
        root = insertRec(root, val);
    }

    private TreeNode insertRec(TreeNode root, int val) {
        if (root == null) {
            return new TreeNode(val);
        }

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

        return root;
    }

    public boolean search(int val) {
        return searchRec(root, val);
    }

    private boolean searchRec(TreeNode root, int val) {
        if (root == null) return false;
        if (root.val == val) return true;

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

Essential Algorithms

1. Searching Algorithms

Linear Search:

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

Binary Search:

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

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

        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            left = mid + 1;
        } else {
            right = mid - 1;
        }
    }
    return -1; // Not found
}
```

2. Sorting Algorithms

Bubble Sort ($O(n^2)$):

```java
public static void bubbleSort(int[] arr) {
    int n = arr.length;
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

Merge Sort (O(n log n)):

```java
public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}

private static void merge(int[] arr, int left, int mid, int right) {
    int[] temp = new int[right - left + 1];
    int i = left, j = mid + 1, k = 0;

    while (i <= mid && j <= right) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        } else {
            temp[k++] = arr[j++];
        }
    }

    while (i <= mid) temp[k++] = arr[i++];
    while (j <= right) temp[k++] = arr[j++];

    System.arraycopy(temp, 0, arr, left, temp.length);
}
```

Quick Sort (Average O(n log n)):

```java
public static void quickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

private static int partition(int[] arr, int low, int high) {
    int pivot = arr[high];
    int i = low - 1;

    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr, i, j);
        }
    }
    swap(arr, i + 1, high);
    return i + 1;
}

private static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}
```

Java Collections Framework

Understanding Java's built-in data structures:

java

```java
import java.util.*;

// ArrayList - Dynamic array
List<Integer> list = new ArrayList<>();
list.add(1);
list.get(0);

// LinkedList - Doubly linked list
LinkedList<Integer> linkedList = new LinkedList<>();
linkedList.addFirst(1);
linkedList.addLast(2);

// Stack
Stack<Integer> stack = new Stack<>();
stack.push(1);
int top = stack.pop();

// Queue
Queue<Integer> queue = new LinkedList<>();
queue.offer(1);
int front = queue.poll();

// PriorityQueue - Min heap by default
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(3);
pq.offer(1);
int min = pq.poll(); // Returns 1

// HashMap
Map<String, Integer> map = new HashMap<>();
map.put("key", 1);
int value = map.get("key");

// HashSet
Set<Integer> set = new HashSet<>();
set.add(1);
boolean contains = set.contains(1);

// TreeMap - Sorted map
TreeMap<Integer, String> treeMap = new TreeMap<>();

// TreeSet - Sorted set
TreeSet<Integer> treeSet = new TreeSet<>();
```

Problem-Solving Approach

Step-by-Step Method:

1. **Understand the Problem**
    - Read carefully
    - Identify inputs and outputs
    - Look for edge cases
2. **Plan Your Approach**
    - Think of brute force solution first
    - Optimize using appropriate data structures
    - Consider time/space tradeoffs
3. **Code Implementation**
    - Write clean, readable code
    - Handle edge cases
    - Use meaningful variable names
4. **Test Your Solution**
    - Test with example cases
    - Test edge cases
    - Verify complexity

Common Problem Patterns:

Two Pointers:

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

Sliding Window:

java

```java
public int maxSumSubarray(int[] nums, int k) {
    int maxSum = 0, windowSum = 0;

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

    // Slide the window
    for (int i = k; i < nums.length; i++) {
        windowSum += nums[i] - nums[i - k];
        maxSum = Math.max(maxSum, windowSum);
    }
    return maxSum;
}
```

Practice Strategy

Learning Path:

1. **Week 1-2: Fundamentals**
    - Arrays and Strings
    - Basic sorting and searching
    - Time/Space complexity
2. **Week 3-4: Linear Data Structures**
    - Linked Lists
    - Stacks and Queues
    - Hash Tables
3. **Week 5-6: Trees and Graphs**
    - Binary Trees
    - Binary Search Trees
    - Graph traversal (BFS, DFS)
4. **Week 7-8: Advanced Topics**
    - Dynamic Programming
    - Greedy algorithms
    - Advanced sorting

Practice Platforms:

- LeetCode
- HackerRank
- CodeSignal
- GeeksforGeeks

Tips for Success:

- Practice consistently (1-2 problems daily)
- Focus on understanding, not just memorizing
- Implement data structures from scratch
- Analyze time and space complexity
- Review and optimize your solutions
- Participate in coding contests

Remember: DSA mastery comes with consistent practice and understanding the underlying concepts, not just memorizing solutions!