**HW**

**Section A- Algorithms**

**1. What is an algorithm?**

An algorithm is a step-by-step procedure or a set of rules for solving a specific problem or performing a computation. It takes an input, processes it through a defined sequence of operations, and produces an output.

**2. Why is algorithm analysis important?**

Algorithm analysis is crucial because it helps understand an algorithm's efficiency and performance. By analyzing an algorithm, we can determine its time complexity (speed) and space complexity (memory usage), which aids in selecting the best algorithm for a given problem.

**3. What are the key criteria for analyzing an algorithm?** The key criteria for analyzing an algorithm include:

* **Time Complexity:** Measures the number of operations an algorithm performs relative to input size.
* **Space Complexity:** Measures the amount of memory required by the algorithm.
* **Correctness:** Ensures the algorithm produces the right output for all valid inputs.
* **Optimality:** Determines whether the algorithm is the most efficient one available.
* **Scalability:** Evaluates how well the algorithm performs as input size grows.

**4. What are the different approaches to developing algorithms?** The different approaches to developing algorithms include:

* **Brute Force:** Tries all possible solutions and selects the best one.
* **Divide and Conquer:** Breaks the problem into smaller subproblems, solves them recursively, and combines the results.
* **Dynamic Programming:** Stores results of overlapping subproblems to avoid redundant calculations.
* **Greedy Approach:** Makes the locally optimal choice at each step with the hope of finding a global optimum.
* **Backtracking:** Explores all possibilities recursively and abandons paths that fail to meet the criteria.
* **Randomized Algorithms:** Uses random choices to solve the problem efficiently.

**5. What are the characteristics of a good algorithm?** A good algorithm should have the following characteristics:

* **Correctness:** It must solve the problem correctly for all valid inputs.
* **Efficiency:** It should run in the least possible time and use minimal resources.
* **Clarity and Simplicity:** It should be easy to understand, implement, and debug.
* **Scalability:** It should perform well even with large input sizes.
* **Deterministic:** Given the same input, it should produce the same output every time.
* **Finiteness:** It must terminate after a finite number of steps.

**Section B - Data Structures**

**1. Different Types of Data Structures**

Data structures are fundamental components used to organize and manage data efficiently. They can be broadly categorized as follows:

* **Linear Data Structures: Elements are arranged in a sequential manner.**
  + Array
  + Linked List
  + Stack
  + Queue
* **Non-Linear Data Structures: Elements are connected in a hierarchical or networked fashion.**
  + Graph
  + Tree
  + Hash Table
  + Heap
* **Other Data Structures:**
  + Trie
  + Disjoint Set
  + Priority Queue

**2. Difference Between an Array and a Linked List**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Array** | **Linked List** |
| **Memory Allocation** | Contiguous memory | Non-contiguous memory |
| **Access Time** | O(1) (Direct Access) | O(n) (Sequential Access) |
| **Insertion/Deletion** | Costly, requires shifting | Efficient, just update pointers |
| **Memory Usage** | Fixed size, may waste memory | Uses extra memory for pointers |

**3. How Does a Stack Work?**

A stack is a Last In, First Out (LIFO) data structure, where elements are added and removed from the top.

**Operations:**

* **Push:** Add an element to the top.
* **Pop:** Remove the top element.
* **Peek (Top):** Retrieve the top element without removing it.
* **isEmpty:** Check if the stack is empty.

**Real-time Example:**

* Browser Back Button: When navigating webpages, the history is stored in a stack. Clicking back pops the last visited page.

**4. Operations on a Queue and Different Types of Queues**

A queue follows the First In, First Out (FIFO) principle.

**Operations:**

* **Enqueue:** Add an element to the rear.
* **Dequeue:** Remove an element from the front.
* **Front:** Retrieve the front element without removing it**.**
* **isEmpty:** Check if the queue is empty.

**Types of Queues:**

1. **Simple Queue:** Basic FIFO queue.
2. **Circular Queue:** The rear and front are connected to make it circular.
3. **Priority Queue:** Elements are dequeued based on priority.
4. **Double-Ended Queue (Deque):** Insertions and deletions are possible at both ends.

**5. What is a Graph? Different Types of Graphs**

A graph is a collection of nodes (vertices) connected by edges.

**Types of Graphs:**

1. **Directed Graph (Digraph**)**:** Edges have direction.
2. **Undirected Graph:** Edges do not have direction.
3. **Weighted Graph:** Edges have weights (costs).
4. **Unweighted Graph:** All edges are equal.
5. **Cyclic Graph:** A path forms a cycle.
6. **Acyclic Graph:** No cycles exist.
7. **Connected Graph:** All vertices are connected.
8. **Disconnected Graph:** Some vertices are not connected.

**Section C - Recursion**

**1. What is Recursion, and How Does It Work?**

Recursion is a programming technique where a function calls itself to solve a problem. It breaks a problem into smaller subproblems of the same type until a base condition is met.

**Example:**

public int factorial(int n) {

if (n == 0) return 1; // Base Case

return n \* factorial(n - 1); // Recursive Case

}

**2. Why is Recursion Used in Programming?**

**Recursion is used for:**

* Solving problems that can be broken into smaller subproblems (e.g., divide and conquer algorithms).
* Making code more readable and elegant.
* Reducing complex looping logic.
* Implementing data structures like trees and graphs.

**3. Advantages and Disadvantages of Recursion**

**Advantages:**

* Reduces code complexity for problems like tree traversal.
* Makes the implementation of some algorithms simpler.
* Useful for problems with a naturally recursive structure.

**Disadvantages:**

* Can lead to high memory usage due to function call stack.
* Increases execution time compared to iteration in some cases.
* Risk of stack overflow if recursion depth is too high.

**4. Difference Between Recursion and Iteration**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Recursion** | **Iteration** |
| **Approach** | Function calls itself | Uses loops (for, while) |
| **Memory Usage** | Uses call stack | Uses less memory |
| **Performance** | Can be slower due to overhead | Generally faster |
| **Complexity** | Simpler for problems like tree traversal | More efficient for simple loops |

**5. What Are Base Cases in Recursion, and Why Are They Important?**

A base case is the condition that stops the recursion. Without a base case, recursion would continue indefinitely, leading to a stack overflow.

**Example:**

public void printNumbers(int n) {

if (n == 0) return; // Base Case

System.out.println(n);

printNumbers(n - 1); // Recursive Call

}

**6. Which Problems Can Be Solved Using Recursion?**

**Recursion is commonly used in:**

* **Mathematical Computations:** Factorial, Fibonacci, GCD.
* **Sorting Algorithms:** QuickSort, MergeSort.
* **Tree and Graph Traversal:** Depth-First Search (DFS), Binary Tree Operations.
* **Dynamic Programming:** Solving subproblems recursively and caching results.
* **Backtracking**: Solving puzzles like Sudoku, N-Queens Problem.