PARUL UNIVERSITY

FACULTY OF ENGINEERING & TECHNOLOGY

B. Tech 5th Semester

Assignment Questions-I

Subject: (Design and Analysis of Algorithm) Code: (303105218)

Date: (10/06/2024) To be submitted by: 28th June 2024

1. Discuss the different types of algorithms and their classifications. Explain at least two design techniques with examples.

Ans: **Types of Algorithms:**

1. **Search Algorithms:** These are used to find an element within a data structure.
   - Example: Binary Search, Linear Search
2. **Sorting Algorithms:** These arrange elements in a particular order (ascending or descending).
   - Example: Quick Sort, Merge Sort, Bubble Sort
3. **Graph Algorithms:** These are used to solve problems related to graph theory.
   - Example: Dijkstra’s Algorithm, Depth-First Search (DFS), Breadth-First Search (BFS)
4. **Dynamic Programming Algorithms:** These solve complex problems by breaking them down into simpler subproblems and storing the results of these subproblems.
   - Example: Fibonacci Sequence, Knapsack Problem
5. **Greedy Algorithms:** These build up a solution piece by piece, always choosing the next piece that offers the most immediate benefit.
   - Example: Prim's Algorithm, Kruskal’s Algorithm

Classifications of Algorithms:

1. **By Implementation:**
   - Recursive Algorithm
   - Iterative Algorithm
2. **By Design Paradigm:**
   - Divide and Conquer
   - Dynamic Programming
   - Greedy Algorithms
   - Backtracking
   - Branch and Bound
3. **By Problem Type:**
   - Sorting
   - Searching
   - String Matching
   - Graph Problems

Design Techniques:

1. **Divide and Conquer:**
   - Example: Merge Sort
     - **Divide:** Split the array into two halves.
     - **Conquer:** Recursively sort the two halves.
     - **Combine:** Merge the sorted halves to produce the sorted array.
   - Example: Binary Search
     - **Divide:** Check the middle element of the array.
     - **Conquer:** Recursively search the left or right half depending on the comparison.
     - **Combine:** Return the position of the element if found.
2. **Dynamic Programming:**
   - Example: Fibonacci Sequence
     - **Subproblems:** Break down the Fibonacci sequence calculation into subproblems of smaller Fibonacci numbers.
     - **Memoization:** Store the results of subproblems to avoid redundant calculations.
   - Example: Knapsack Problem
     - **Subproblems:** Calculate the maximum value that can be obtained for smaller weights.
     - **Memoization:** Use a table to store the results of subproblems to build the solution to the overall problem.

2. Explain the Big Oh, Big Omega, and Big Theta notations. Provide examples of each and discuss their significance in analyzing algorithms.

Ans: **Big Oh (O):** Describes an upper bound on the time complexity of an algorithm, meaning it provides a worst-case scenario.

- **Example:** $O(n^2)$ for Bubble Sort.
- **Significance:** Ensures that the algorithm will not take more time than the given upper bound.

Big Omega (Ω): Describes a lower bound on the time complexity of an algorithm, meaning it provides a best-case scenario.

- **Example:** $\Omega(n)$ for a Linear Search.
- **Significance:** Guarantees that the algorithm will take at least this much time.

Big Theta (Θ): Describes a tight bound on the time complexity, meaning it provides both the upper and lower bounds.

- **Example:** $\Theta(n \log n)$ for Merge Sort.
- **Significance:** Gives an exact asymptotic behavior of the algorithm.

3. Describe the best-case, worst-case, and average-case scenarios for an algorithm. Illustrate these with the example of the Insertion Sort algorithm.

Ans: **Best-Case:** The minimum time required for the algorithm to complete.

- **Example:** Insertion Sort with already sorted array. Complexity: $O(n)$.

Worst-Case: The maximum time required for the algorithm to complete.

- **Example:** Insertion Sort with a reverse sorted array. Complexity: $O(n^2)$.

Average-Case: The expected time required for the algorithm to complete over all inputs.

- **Example:** Insertion Sort for random order. Complexity: $O(n^2)$.

Insertion Sort Example:

Best-Case: The array is already sorted.

- Time Complexity: $O(n)$ **Worst-Case:** The array is sorted in reverse order.
- Time Complexity: $O(n^2)$ **Average-Case:** The array elements are in random order.
- Time Complexity: $O(n^2)$

4. Explain the importance of loop invariants in algorithm correctness. Prove the correctness of the Insertion Sort algorithm using a loop invariant.

Ans: **Loop Invariant:** A property that holds true before and after each iteration of a loop.

Correctness Proof using Insertion Sort:

1. **Initialization:** Before the first iteration, the subarray consists of just one element, which is trivially sorted.
2. **Maintenance:** If the subarray is sorted before an iteration, it remains sorted after the iteration.
3. **Termination:** When the loop terminates, the subarray is the entire array, which is sorted.

5. Compare and contrast Bubble Sort, Selection Sort, and Insertion Sort in terms of their time complexity, space complexity, and practical usage scenarios. Provide a detailed analysis of each.

Ans: **Bubble Sort:**

- **Time Complexity:** $O(n^2)$
- **Space Complexity:** $O(1)$
- **Usage:** Simple to implement but inefficient for large datasets.

Selection Sort:

- **Time Complexity:** $O(n^2)$
- **Space Complexity:** $O(1)$
- **Usage:** Useful when memory write operations are costly.

Insertion Sort:

- **Time Complexity:** $O(n)$ in best case, $O(n^2)$ in worst case
- **Space Complexity:** $O(1)$
- **Usage:** Efficient for small datasets or nearly sorted data.

6. Analyze the time complexity of the binary search algorithm using the divide-and-conquer approach. Provide a recurrence relation and solve it to find the complexity.

Ans: **Binary Search Algorithm:**

- **Divide and Conquer Approach:**
  - **Divide:** Find the middle element.
  - **Conquer:** Recursively search the left or right half.
  - **Combine:** Return the position if found.

Recurrence Relation:

- $T(n) = T(n/2) + O(1)$
- Solving this recurrence gives: $T(n) = O(\log n)$

7. Describe the partitioning process in Quick Sort. Write down the recurrence relation for Quick Sort and use the Master Method to derive its average and worst-case time complexities.

Ans: **Partitioning Process:**

- Choose a pivot element and partition the array into two subarrays.
- Elements less than the pivot go to the left, and elements greater go to the right.

Recurrence Relation:

- $T(n) = T(k) + T(n-k-1) + O(n)$

Average and Worst-Case Time Complexities:

- **Average Case:** $O(n \log n)$ using Master Method.
- **Worst Case:** $O(n^2)$ when the pivot is the smallest or largest element.

8. Explain the merge process in Merge Sort. Derive the time complexity of Merge Sort using a recurrence relation and solve it.

Ans: **Merge Process:**

- Divide the array into two halves.
- Recursively sort the two halves.
- Merge the sorted halves.

Recurrence Relation:

- $T(n) = 2T(n/2) + O(n)$
- Solving this recurrence gives: $T(n) = O(n \log n)$

9. Describe Strassen's algorithm for matrix multiplication. Compare its time complexity with the conventional matrix multiplication algorithm.

Ans: **Strassen's Algorithm:**

- Uses divide-and-conquer to reduce the number of multiplications.
- Reduces complexity from $O(n^3)$ to $O(n^2.81)$.

Comparison:

- Conventional algorithm: $O(n^3)$
- Strassen's algorithm: $O(n^2.81)$

10. Using the divide-and-conquer approach, design an algorithm to find the maximum and minimum elements in an array. Provide the recurrence relation and derive its time complexity.

Ans: **Divide-and-Conquer Algorithm:**

- **Divide:** Split the array into two halves.
- **Conquer:** Recursively find the maximum and minimum of the two halves.
- **Combine:** Compare the results of the two halves to get the overall maximum and minimum.

Recurrence Relation:

- $T(n) = 2T(n/2) + O(1)$
- Solving this recurrence gives: $T(n) = O(n)$