ADSA Assignment

Algorithms and Data Structures Analysis

Name: Pragya Srivastava
Roll No: A125013
Course: MTech CSE 1st semester

Question 1

Question: Explain the representation of a Binary Heap using an array and prove that all leaf nodes are stored in the second half of the array.

Answer

A Binary Heap is a complete binary tree that satisfies the heap property:

- In a **Max Heap**, the value of each node is greater than or equal to its children.
- In a **Min Heap**, the value of each node is less than or equal to its children.

A binary heap is commonly represented using an array instead of pointers.

Array Representation

For a heap stored in an array indexed from 1:

- Parent of node at index i is at $\lfloor i/2 \rfloor$
- Left child is at $2i$
- Right child is at $2i + 1$

Proof: Leaf Nodes in Second Half

In a complete binary tree with n nodes:

- Nodes from index $\lfloor n/2 \rfloor + 1$ to n have no children
- These nodes cannot have left or right children because their child indices exceed n

Hence, all leaf nodes lie in the second half of the array.

Time Complexity

This observation is based purely on index calculation.

$$\boxed{O(1)}$$

—

Question 2

Question: Given the following functions, arrange them in increasing order of growth rate:

$$1,\ \log n,\ \sqrt{n},\ n^{0.51},\ n,\ n \log n,\ n^2,\ n^3$$

Answer

To compare growth rates, we analyze how each function behaves as $n \to \infty$.

Increasing Order of Growth

$$1 < \log n < \sqrt{n} < n^{0.51} < n < n \log n < n^2 < n^3$$

Explanation

- Constant functions grow slowest.
- Logarithmic functions grow slower than polynomial functions.
- Any power of n grows faster than $\log n$.
- Higher powers dominate lower powers.

Conclusion

As input size increases, functions on the right dominate those on the left.

—

Question 3

Question: Compare Binary Search and Ternary Search. Which is more efficient and why?

Answer

Binary Search

Binary Search divides the array into two halves at each step.

- Requires sorted input
- Reduces problem size by half

Ternary Search

Ternary Search divides the array into three parts using two mid points.

Comparison

Binary Search performs one comparison per level, while Ternary Search performs two comparisons per level.

Time Complexity

$$\text{Binary Search: } O(\log_2 n)$$

$$\text{Ternary Search: } O(\log_3 n)$$

Even though $\log_3 n < \log_2 n$, the constant factor in ternary search is higher due to extra comparisons.

Conclusion

$$\boxed{\text{Binary Search is more efficient in practice}}$$

—

Question 4

Question: Explain the Heap Sort algorithm and analyze its time complexity.

Answer

Heap Sort is a comparison-based sorting algorithm that uses a Binary Heap.

Algorithm Steps

1. Build a Max Heap from the input array
2. Swap the root with the last element
3. Reduce heap size and heapify
4. Repeat until sorted

Time Complexity Analysis

- Building heap: $O(n)$
- Heapify operation: $O(\log n)$
- Performed n times

$$\text{Total Time Complexity} = O(n \log n)$$

Space Complexity

Heap Sort is an in-place algorithm:

$$\boxed{O(1)}$$

Conclusion

Heap Sort guarantees $O(n \log n)$ time in all cases and does not require extra memory.

Question 5

Question: Explain Heap Sort with an example and analyze its time complexity.

Answer

Heap Sort is a comparison-based sorting algorithm that uses a **Binary Heap** data structure. It works by first organizing the data into a heap and then repeatedly extracting the maximum (or minimum) element.

Algorithm

1. Build a Max Heap from the given array.
2. Swap the root (maximum element) with the last element.
3. Reduce the heap size by one.
4. Heapify the root to maintain heap property.
5. Repeat until the heap size becomes 1.

Time Complexity

- Building the heap: $O(n)$
- Heapify operation: $O(\log n)$
- Performed for n elements

$$\boxed{O(n \log n)}$$

Conclusion

Heap Sort is efficient, in-place, and guarantees $O(n \log n)$ time in best, average, and worst cases.

—

Question 6

Question: Explain the concept of Longest Common Subsequence (LCS) and its dynamic programming solution.

Answer

The **Longest Common Subsequence (LCS)** problem is to find the longest sequence that appears in the same relative order in two strings (not necessarily contiguous).

Dynamic Programming Approach

Let $X[1..m]$ and $Y[1..n]$ be two sequences.

Define:

$$L[i][j] = \text{length of LCS of } X[1..i] \text{ and } Y[1..j]$$

Recurrence Relation

$$L[i][j] = \begin{cases} 0, & i = 0 \text{ or } j = 0 \\ L[i-1][j-1] + 1, & X[i] = Y[j] \\ \max(L[i-1][j], L[i][j-1]), & X[i] \neq Y[j] \end{cases}$$

Time and Space Complexity

$$\text{Time Complexity} = O(mn)$$

$$\text{Space Complexity} = O(mn)$$

Conclusion

Dynamic Programming efficiently solves LCS by storing intermediate results and avoiding redundant computations.

—

Question 7

Question: Explain the Knuth–Morris–Pratt (KMP) string matching algorithm.

Answer

The KMP algorithm is a pattern matching algorithm that avoids unnecessary re-comparisons by using a preprocessing table.

Key Idea

Instead of restarting the comparison from the beginning after a mismatch, KMP uses information from previous matches.

LPS Array

LPS (Longest Prefix Suffix) array stores the length of the longest proper prefix that is also a suffix for each prefix of the pattern.

Algorithm Steps

1. Precompute the LPS array for the pattern.
2. Compare pattern with text.
3. On mismatch, shift the pattern using LPS values.

Time Complexity

$$\boxed{O(n+m)}$$

where n is the length of text and m is the length of pattern.

Conclusion

KMP is efficient and avoids redundant comparisons, making it suitable for large texts.

—

Question 8

Question: Explain Strassen's Matrix Multiplication algorithm and analyze its complexity.

Answer

Strassen's algorithm is an optimized divide-and-conquer method for matrix multiplication.

Key Idea

Instead of using 8 recursive multiplications for 2×2 matrices, Strassen uses only 7 multiplications.

Matrix Division

Each matrix is divided into four submatrices:

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}, \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

Time Complexity

$$T(n) = 7T\left(\frac{n}{2}\right) + O(n^2)$$

Using Master Theorem:

$$\boxed{O(n^{\log_2 7}) \approx O(n^{2.81})}$$

Conclusion

Strassen's algorithm is faster than traditional matrix multiplication for large matrices.

Question 9

Question: Explain Dijkstra's Algorithm for finding the shortest path from a single source.

Answer

Dijkstra's Algorithm is a greedy algorithm used to find the shortest path from a given source vertex to all other vertices in a weighted graph with non-negative edge weights.

Algorithm

1. Initialize distance of source as 0 and all other vertices as ∞.
2. Mark all vertices as unvisited.
3. Select the unvisited vertex with the minimum distance.
4. Update distances of its adjacent vertices.
5. Mark the selected vertex as visited.
6. Repeat until all vertices are visited.

Correctness Proof

At each step, the algorithm selects the closest unvisited vertex. Since all edge weights are non-negative, once a vertex is selected, its shortest distance is finalized.

Time Complexity

- Using adjacency matrix: $O(V^2)$
- Using priority queue: $O((V + E)\log V)$

Conclusion

Dijkstra's Algorithm efficiently computes shortest paths and is widely used in routing and network applications.

—

Question 10

Question: Explain the Floyd–Warshall Algorithm for All-Pairs Shortest Paths.

Answer

The Floyd–Warshall Algorithm computes shortest paths between all pairs of vertices in a weighted graph.

Dynamic Programming Approach

Let $dist[i][j]$ represent the shortest distance from vertex i to j.

Recurrence Relation

$$dist[i][j] = \min(dist[i][j],\ dist[i][k] + dist[k][j])$$

for all intermediate vertices k.

Algorithm Steps

1. Initialize distance matrix using adjacency matrix.
2. Consider each vertex as an intermediate vertex.
3. Update all-pairs distances using the recurrence relation.

Time Complexity

$$\boxed{O(V^3)}$$

Conclusion

Floyd–Warshall is simple and effective for dense graphs and supports negative edge weights (without negative cycles).

—

Question 11

Question: Explain Prim's Algorithm for finding the Minimum Spanning Tree.

Answer

Prim's Algorithm is a greedy algorithm that finds a Minimum Spanning Tree (MST) by expanding a tree one vertex at a time.

Algorithm

1. Start with an arbitrary vertex.
2. Select the minimum weight edge connecting the tree to a new vertex.
3. Add the selected edge and vertex to the MST.
4. Repeat until all vertices are included.

Correctness Argument

Prim's Algorithm always chooses the minimum-weight edge that expands the tree, ensuring no cycles and minimal total cost.

Time Complexity

- Adjacency Matrix: $O(V^2)$
- Priority Queue: $O(E \log V)$

Conclusion

Prim's Algorithm is efficient for dense graphs and ensures optimal MST construction.

—

Question 12

Question: Explain Kruskal's Algorithm for finding the Minimum Spanning Tree.

Answer

Kruskal's Algorithm is a greedy algorithm that builds an MST by selecting edges in increasing order of weight.

Algorithm

1. Sort all edges in ascending order of weight.
2. Initialize disjoint sets for each vertex.
3. Pick the smallest edge that does not form a cycle.
4. Add the edge to MST.
5. Repeat until MST contains $V - 1$ edges.

Cycle Detection

Union–Find (Disjoint Set Union) is used to efficiently detect cycles.

Time Complexity

$$\boxed{O(E \log E)}$$

Conclusion

Kruskal's Algorithm is efficient for sparse graphs and produces an optimal MST.

Question 13

Question: Explain the Knuth–Morris–Pratt (KMP) Pattern Matching Algorithm.

Answer

The Knuth–Morris–Pratt (KMP) algorithm is an efficient string matching algorithm that avoids unnecessary comparisons by utilizing information from previously matched characters.

Key Idea

When a mismatch occurs, the pattern itself contains enough information to determine where the next match could begin. This is achieved using the Longest Prefix Suffix (LPS) array.

LPS Array

The LPS array stores the length of the longest proper prefix which is also a suffix for each prefix of the pattern.

Algorithm Steps

1. Preprocess the pattern to build the LPS array.
2. Match characters of the text and pattern sequentially.
3. On mismatch, use the LPS array to avoid rechecking characters.

Correctness

Since the algorithm never rechecks characters in the text, all possible matches are examined efficiently without redundancy.

Time Complexity

$$\boxed{O(n+m)}$$

where n is the length of the text and m is the length of the pattern.

Conclusion

KMP significantly improves string matching performance compared to naive methods.

—

Question 14

Question: Explain the Longest Common Subsequence (LCS) problem using Dynamic Programming.

Answer

The Longest Common Subsequence (LCS) problem finds the longest sequence that appears in the same relative order in two strings.

Dynamic Programming Formulation

Let $L[i][j]$ denote the length of the LCS of prefixes $X[1..i]$ and $Y[1..j]$.

Recurrence Relation

$$L[i][j] = \begin{cases} 0, & i = 0 \text{ or } j = 0 \\ L[i-1][j-1] + 1, & X[i] = Y[j] \\ \max(L[i-1][j], L[i][j-1]), & X[i] \neq Y[j] \end{cases}$$

Algorithm Steps

1. Initialize a DP table.
2. Fill table using recurrence relation.
3. Backtrack to construct the LCS.

Time Complexity

$$\boxed{O(m \times n)}$$

Conclusion

Dynamic programming ensures an optimal solution for the LCS problem.

—

Question 15

Question: Explain Heap Sort Algorithm.

Answer

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure.

Algorithm

1. Build a max heap from the input array.
2. Swap the root with the last element.
3. Reduce heap size and heapify.
4. Repeat until array is sorted.

Correctness

The heap property ensures the largest element is always at the root, leading to a sorted array after successive removals.

Time Complexity

$$\boxed{O(n \log n)}$$

Space Complexity

$$\boxed{O(1)}$$

Conclusion

Heap Sort is efficient and does not require extra memory.

—

Question 16

Question: Explain Binary Search (Recursive and Non-Recursive).

Answer

Binary Search is an efficient algorithm for finding an element in a sorted array by repeatedly dividing the search space into half.

Iterative Approach

- Compare middle element with key.
- If matched, return index.
- Else search left or right half.

Recursive Approach

The same logic is applied using recursive function calls.

Correctness

At each step, the search space is reduced by half, ensuring logarithmic efficiency.

Time Complexity

$$\boxed{O(\log n)}$$

Conclusion

Binary Search is optimal for searching in sorted datasets.