Name: Tanvi Malewar

PRN: 246109054

Batch: IT B3

Assignment for ISE2 DS Theory

Q. 1 Write an algorithm a pointer-based graph representation (structs + pointers) for an arbitrary

graph, then traverse using both DFS (recursive) and BFS (queue-based). Compare memory usage.

=>

```
Algorithm createGraph(nodes)
Create a pointer-based graph representation using structs and pointers
Pre: - nodes is the number of vertices in the graph
Post: - Returns a graph structure with n nodes
1. Create graph ← new Graph
2. For i ← 0 to nodes - 1
1. Create newNode ← new Node
2. newNode.data ← i
3. newNode.visited ← false
4. newNode.neighbors ← empty list
5. graph.nodes.add(newNode)
3. End For
4. Return graph
end createGraph
```

```
Algorithm DFS(node)
Traverse graph using Depth First Search (recursive)
Pre: - node is the starting node for traversal
Post: - Prints DFS traversal order
1. If node = NULL then Return
2. Print node.data
3. node.visited ← true
4. For each neighbor in node.neighbors
```

```
            1. If neighbor.visited = false then
                1. DFS(neighbor)
    5. End For
end DFS
```

```
Algorithm BFS(startNode)
Traverse graph using Breadth First Search (queue-based)
Pre: - startNode is the starting node for traversal
Post:
- Prints BFS traversal order
1. Create queue
2. queue.enqueue(startNode)
3. startNode.visited ← true
4. While queue is not empty
    1. current ← queue.dequeue()
    2. Print current.data
    3. For each neighbor in current.neighbors
        1. If neighbor.visited = false then
            1. neighbor.visited ← true
            2. queue.enqueue(neighbor)
    4. End For
5. End While
end BFS
```

Memory Usage Comparison: - DFS: O(h) memory for recursion stack (h = height/depth) - BFS: O(w) memory for queue (w = maximum width) - DFS better for deep graphs, BFS better for wide graphs

Q2. Write an algorithm (with pseudo-code) to find all pairs of elements in an array whose sum

equals a given target. Analyze its time and space complexity.

=>

```
Algorithm findPairs(arr, target)
Find all pairs of elements in an array whose sum equals a given target
```

```
Pre: - arr is an array of integers - target is the desired sum
Post: - Returns list of pairs that sum to target
1. Create empty hashMap
2. Create empty resultList
3. For i ← 0 to length(arr) - 1
    1. complement ← target - arr[i]
    2. If complement exists in hashMap then
        1. Add pair(arr[i], complement) to resultList
    3. End If
    4. Add arr[i] to hashMap with value i
4. End For
5. Return resultList
end findPairs
```

Time Complexity: O(n)

 Outer loop runs n-1 times, inner loop runs n-i-1 times.

 Total iterations ≈ $n*(n-1)/2 \rightarrow O(n^2)$

 Space Complexity: O(1) (no extra space apart from loop variables).

Space Complexity: O(n)

Hash table stores at most n elements → O(n)

Q3. Write an algorithm to Create a structure Student with fields for name, roll number, and marks.

=>

```
Algorithm createStudent(name, rollNo, marks)
Create a structure Student with fields for name, roll number, and marks
Pre: - name is student name - rollNo is roll number - marks is student marks
Post: - Returns Student structure
1. STRUCT Student:
    1. name: string
    2. rollNo: integer
    3. marks: float
2. Create newStudent ← new Student
```

```
3. newStudent.name ← name
4. newStudent.rollNo ← rollNo
5. newStudent.marks ← marks
6. Return newStudent
end createStudent
```

Q4. Write an algorithm create a functions to insert a node at any position in a singly linked list.

Then

modify your function to handle circular linked lists.

=>

```
Algorithm insertAtPosition(head, data, position)
Insert a node at any position in a singly linked list
Pre: - head is the head of linked list - data is the value to insert - position is the insertion position (0-based)
Post: - Returns updated head of list
1. Create newNode ← new Node(data)
2. If position = 0 then
1. newNode.next ← head
2. Return newNode
3. End If
4. current ← head
5. For i ← 1 to position - 1
1. If current = NULL then break
2. current ← current.next
6. End For
7. If current ≠ NULL then
1. newNode.next ← current.next
2. current.next ← newNode
8. End If
9. Return head
end insertAtPosition
```

```
Algorithm insertCircular(head, data, position)

Insert a node at any position in a circular linked list

Pre: - head is the head of circular linked list - data is the value to insert - position is the
insertion position

Post: - Returns updated head of circular list

1. Create newNode ← new Node(data)
2. If head = NULL then
    1. newNode.next ← newNode
    2. Return newNode
3. End If
4. If position = 0 then
    1. last ← head
    2. While last.next ≠ head do
        1. last ← last.next
    3. End While
    4. newNode.next ← head
    5. last.next ← newNode
    6. Return newNode
5. End If
6. current ← head
7. For i ← 1 to position - 1
    1. current ← current.next
    2. If current = head then break
8. End For
9. newNode.next ← current.next
10. current.next ← newNode
11. Return head
end insertCircular
```

Q5 . Write an algorithm for Given two sorted singly linked lists, write an algorithm to merge them

into a single sorted list without creating new nodes (relink existing ones).

=>

```
Algorithm mergeSortedLists(l1, l2)
Merge two sorted singly linked lists into single sorted list
Pre: - l1, l2 are heads of two sorted linked lists


Post: - Returns head of merged sorted list
  1. Create dummy ← new Node(0)
  2. current ← dummy
  3. While l1 ≠ NULL AND l2 ≠ NULL
    1. If l1.data ≤ l2.data then
      1. current.next ← l1
      2. l1 ← l1.next
    2. Else
      1. current.next ← l2
      2. l2 ← l2.next
    3. End If
    4. current ← current.next
  4. End While
  5. If l1 ≠ NULL then
    1. current.next ← l1
  6. Else
    1. current.next ← l2
  7. End If
  8. Return dummy.next
end mergeSortedLists
```

Q6. Write an algorithm function to reverse a doubly linked list in-place. Prove that your algorithm

runs in O(n) time.

=>

```
Algorithm reverseDoublyLinkedList(head)
Reverse a doubly linked list in-place
Pre: - head is the head of doubly linked list
Post: - Returns new head of reversed list
  1. current ← head
  2. temp ← NULL
  3. While current ≠ NULL
    1. temp ← current.prev
    2. current.prev ← current.next
    3. current.next ← temp
    4. current ← current.prev
4. End While
5. If temp ≠ NULL then
1. head ← temp.prev
6. End If
7. Return head
end reverseDoublyLinkedList
```

Time Complexity Proof: O(n) - Each node visited exactly once - Constant time operations per node - Total operations: 4n → O(n)

Q7. Write an algorithm priority queue using a linked list where lower numbers indicate higher

priority. Show step-by-step insertion and deletion operations.

=>

```
Algorithm insertPriorityQueue(head, data, priority)
Insert into priority queue (lower number = higher priority)
Pre: - head is head of priority queue - data is value to insert - priority is priority value
Post: - Returns updated head of priority queue
1. Create newNode ← new Node(data, priority)
2. If head = NULL OR priority < head.priority then
1. newNode.next ← head
2. Return newNode
```

```
3. End If
4. current ← head
5. While current.next ≠ NULL AND current.next.priority ≤ priority
1. current ← current.next
6. End While
7. newNode.next ← current.next
8. current.next ← newNode
9. Return head
end insertPriorityQueue
```

```
Algorithm deletePriorityQueue(head)
Delete highest priority element from queue
Pre: - head is head of priority queue
Post: - Returns updated head after deletion
1. If head = NULL then Return NULL
2. temp ← head
3. head ← head.next
4. Delete temp
5. Return head
end deletePriorityQueue
```

Proof of O(n) Time Complexity for Reversing a Doubly Linked List:

1. Let the doubly linked list have n nodes.

2. The algorithm uses a while loop:

```
while (current != NULL) {
swap current->next and current->prev
move current to current->prev
}
```

3. Iteration Analysis: - Each iteration processes exactly one node. - Operations per node (swapping two pointers and moving current) take O(1) time.

4. Total Iterations: - The loop runs once per node → n iterations.

5. Total Time: - Time per node × number of nodes = O(1) × n = O(n)

6. Space Analysis: - Only two extra pointers (current and temp) are used → O(1) space.

Conclusion: - Each node is visited exactly once. - Constant-time operations per node. - Therefore, the algorithm runs in O(n) time and uses O(1) extra space.

Q8. Write an algorithm to detect cycles in a directed graph using DFS. Discuss time complexity.

=>

```
Algorithm hasCycleDFS(graph)
Detect cycles in directed graph using DFS
Pre: - graph is the graph structure
Post: - Returns true if cycle exists, false otherwise
1. For each node in graph.nodes
1. visited ← false
2. recStack ← false
  2. End For
  3. For each node in graph.nodes
    1. If visited[node] = false then
      1. If hasCycleUtil(node, visited, recStack) then
        1. Return true
      2. End If
    2. End If
  4. End For
  5. Return false
end hasCycleDFS
```

```
Algorithm hasCycleUtil(node, visited, recStack)
Helper function for cycle detection
  1. visited*node+ ← true
  2. recStack*node+ ← true
  3. For each neighbor in node.neighbors
    1. If visited[neighbor] = false then
      1. If hasCycleUtil(neighbor, visited, recStack) then
        1. Return true
      2. End If
    2. Else If recStack[neighbor] = true then
      1. Return true
    3. End If
  4. End For
  5. recStack*node+ ← false
  6. Return false
end hasCycleUtil
```

Time Complexity: O(V + E)

1.Each vertex is visited exactly once → O(V)

2. Each edge is explored once in DFS → O(E)

3. Total time = O(V + E) for adjacency list representation

Q 9. Write an algorithm for BFS and DFS traversals for a graph represented using adjacency lists.

Compare their order of visiting vertices for a sample graph.

=>

```
Algorithm BFS_Adjacency(graph, start)
BFS traversal for adjacency list representation
Pre: - graph is adjacency list - start is starting vertex
Post: - Prints BFS traversal order
    1. Create visited array of size V, initialize to false
    2. Create queue
    3. visited*start+ ← true
    4. queue.enqueue(start)
    5. While queue not empty
      1. current ← queue.dequeue()
      2. Print current
      3. For each neighbor in graph[current]
        1. If visited[neighbor] = false then
          1. visited*neighbor+ ← true
          2. queue.enqueue(neighbor)
        2. End If
      4. End For
    6. End While
end BFS_Adjacency
```

```
Algorithm DFS_Adjacency(graph, start)
DFS traversal for adjacency list representation
Pre: - graph is adjacency list - start is starting vertex
Post: - Prints DFS traversal order
  1. Create visited array of size V, initialize to false
  2. DFS_Util(graph, start, visited)
end DFS_Adjacency
```

```
Algorithm DFS_Util(graph, v, visited)
  1. visited*v+ ← true
  2. Print v
  3. For each neighbor in graph[v]
    1. If visited[neighbor] = false then
      1. DFS_Util(graph, neighbor, visited)
    2. End If
  4. End For
end DFS_Util
```

Order of Visiting Vertices for Sample Graph:

BFS starting from 1: 1 → 2 → 3 → 4 → 5

DFS starting from 1: 1 → 2 → 4 → 5 → 3

Comparison:

1. BFS visits vertices level by level (closer vertices first)

2. DFS visits vertices depth-first (goes deep before backtracking)

3. BFS may be used to find shortest paths in unweighted graphs

4. DFS is useful for detecting cycles, topological sorting, etc.

Q10. Write an algorithm to sort a large dataset using external merge sort (simulate using file I/O).

Explain each pass with an example.

=>

```
Algorithm externalMergeSort(inputFile, outputFile, memorySize)
Sort large dataset using external merge sort
Pre: - inputFile is unsorted data file - outputFile is sorted output file - memorySize is available memory
Post: - Creates sorted output file
1. // Pass 0: Create sorted runs
2. runs ← empty list
3. While more data in inputFile
    1. Read chunk of memorySize from inputFile
    2. Sort chunk in memory using quickSort
    3. Write sorted chunk to run file
    4. runs.add(runFile)
4. End While
5. // Subsequent passes: Merge runs
6. While runs.size > 1
    1. newRuns ← empty list
    2. For each pair of runs in runs
        1. mergedRun ← mergeTwoRuns(run1, run2)
        2. newRuns.add(mergedRun)
    3. End For
    4. runs ← newRuns
7. End While
```

```
8. Copy final run to outputFile
end externalMergeSort
```

Example:

Dataset: 8, 3, 7, 1, 9, 2, 5, 4, 6

Pass 1 (Create Sorted Runs in Memory of size 4): - Chunk 1: 8,3,7,1 → sort → 1,3,7,8 → write to run1 - Chunk 2: 9,2,5,4 → sort → 2,4,5,9 → write to run2 - Chunk 3: 6 → sort → 6 → write to run3

Pass 2 (Merge Runs): - Merge run1 (1,3,7,8) and run2 (2,4,5,9) → 1,2,3,4,5,7,8,9 → write to temp run - Merge temp run and run3 (6) → 1,2,3,4,5,6,7,8,9 → final sorted file

Q11. Write an algorithm to find the kth smallest element in an unsorted array using Quick Sort

partitioning logic. Analyze its expected complexity.

=>

```
Algorithm findKthSmallest(arr, k)
Find kth smallest element using QuickSelect
Pre: - arr is unsorted array - k is the position (1-based)
Post: - Returns kth smallest element
1. left ← 0
2. right ← length(arr) - 1
3. While left ≤ right
    1. pivotIndex ← partition(arr, left, right)
    2. If pivotIndex = k - 1 then
        1. Return arr[pivotIndex]
    3. Else If pivotIndex < k - 1 then
        1. left ← pivotIndex + 1
    4. Else
        1. right ← pivotIndex - 1
    5. End If
4. End While
5. Return -1 // not found
end findKthSmallest
```

```
Algorithm partition(arr, left, right)
```

```
1. pivot ← arr*right+
2. i ← left - 1
3. For j ← left to right - 1
   1. If arr*j+ ≤ pivot then
      1. i ← i + 1
      2. Swap arr[i] and arr[j]
   2. End If
4. End For
5. Swap arr[i + 1] and arr[right]
6. Return i + 1
end partition
```

Expected Time Complexity Analysis:

1. Each partition operation splits the array into two parts
2. On average, pivot divides array roughly in half
3. Only one side is recursively processed
4. Recurrence: $T(n) = T(n/2) + O(n) \rightarrow T(n) = O(n)$
5. Therefore, expected time complexity = $O(n)$

Worst-Case Time Complexity: - If pivot always picks max/min element → $T(n) = O(n^2)$ - Can be improved to $O(n)$ worst-case using Median-of-Medians pivot selection

Space Complexity: - $O(1)$ extra space (in-place partitioning)

Expected Complexity: $O(n)$ average case, $O(n^2)$ worst case

Q12 Write an algorithm Heap Sort using a binary heap data structure. Show how heapify works

step by step.

=>

```
Algorithm heapSort(arr)
Sort array using heap sort
Pre: - arr is array to be sorted
Post: - arr is sorted in ascending order
1. n ← length(arr)
2. // Build max heap
```

```
3. For i ← n/2 - 1 down to 0
    1. heapify(arr, n, i)
4. End For
5. // Extract elements from heap
6. For i ← n - 1 down to 1
    1. Swap arr[0] and arr[i]
    2. heapify(arr, i, 0)
7. End For
end heapSort
```

```
Algorithm heapify(arr, n, i)
Maintain heap property
1. largest ← i
2. left ← 2*i + 1
3. right ← 2*i + 2
4. If left < n AND arr[left] > arr[largest] then
    1. largest ← left
5. End If
6. If right < n AND arr[right] > arr[largest] then
    1. largest ← right
7. End If
8. If largest ≠ i then
    1. Swap arr[i] and arr[largest]
    2. heapify(arr, n, largest)
9. End If
end heapify
```

Step-by-Step Heapify Example:

Original Array: [4, 10, 3, 5, 1]

Build Max Heap: - i = 1: Heapify subtree *10,5,1+ → already max heap → *10,5,1+ - i = 0: Heapify subtree [4,10,3,5,1]:

compare 4 with children 10 and 3 → largest = 10 → swap 4 and 10

Array now: [10,4,3,5,1]

Heapify node 1 (4) with children 5,1 → largest = 5 → swap 4 and 5

Array now: *10,5,3,4,1+ → Max Heap complete

Heap Sort Passes:

1. Swap root 10 with last element 1 → *1,5,3,4,10+

Heapify root 1 → largest = 5 → swap → *5,1,3,4,10+

Heapify node 1 → largest = 4 → swap → *5,4,3,1,10+

2. Swap root 5 with last element 1 → *1,4,3,5,10+

Heapify root 1 → largest = 4 → swap → *4,1,3,5,10+

3. Swap root 4 with last element 3 → *3,1,4,5,10+

Heapify root 3 → largest = 3 → already max heap

4. Swap root 3 with last element 1 → *1,3,4,5,10+

Heapify root 1 → largest = 3 → swap → *3,1,4,5,10+

Final Sorted Array: [1,3,4,5,10]

Time Complexity: - Build Max Heap: O(n) - Heapify n elements over log n levels → O(n log n) - Total = O(n log n)

Space Complexity: O(1) extra space (in-place)

Q13. Write an algorithm that converts an infix expression to postfix and evaluates the postfix

expression using stacks implemented with linked lists. The expression may include parentheses

and multi-digit operands.

=>

```
Algorithm infixToPostfix(expression)

Convert infix expression to postfix

Pre: - expression is valid infix expression

Post: - Returns postfix expression

  1. Create empty stack

  2. Create empty output

  3. For each token in expression

    1. If token is operand then

      1. output.append(token)

    2. Else If token = '(' then
      1. stack.push(token)
    3. Else If token = ')' then
      1. While stack.top ≠ '('
        1. output.append(stack.pop())
      2. End While
      3. stack.pop() // remove '('
    4. Else // operator
      1. While stack not empty AND precedence(token) ≤ precedence(stack.top)
        1. output.append(stack.pop())
      2. End While
      3. stack.push(token)
    5. End If
  4. End For
  5. While stack not empty
    1. output.append(stack.pop())
  6. End While
  7. Return output
end infixToPostfix

Algorithm evaluatePostfix(postfix)
Evaluate postfix expression
  1. Create empty stack
  2. For each token in postfix
    1. If token is operand then
      1. stack.push(token)
    2. Else // operator
      1. operand2 ← stack.pop()
      2. operand1 ← stack.pop()
      3. result ← apply operator token to operand1 and operand2
      4. stack.push(result)
    3. End If
  3. End For
  4. Return stack.pop()
end evaluatePostfix
```

Q14. Write an algorithm store and retrieve student records using file handling and hashed indexing.

Each record contains: Roll No., Name, Branch, and CGPA.

=>

```
Algorithm storeStudentRecord(rollNo, name, branch, cgpa)
Store student record using hashed indexing
Pre: - Student record details
Post: - Record stored in file with index
1. index ← hash(rollNo) % TABLE_SIZE
2. record ← formatRecord(rollNo, name, branch, cgpa)
3. Write record to data file at position determined by index
4. Update index file with rollNo and file position
end storeStudentRecord
Algorithm retrieveStudentRecord(rollNo)
Retrieve student record using hashed indexing
Pre: - rollNo is student roll number
Post: - Returns student record if found
1. index ← hash(rollNo) % TABLE_SIZE
2. Lookup file position from index file using index
3. Read record from data file at that position
4. If record.rollNo = rollNo then
1. Return record
5. Else
1. Handle collision (linear probing)
```

```
    2. Check subsequent positions
6. End If
end retrieveStudentRecord
```

Q15.Write an algorithm to Sort a singly linked list using Merge Sort (not array conversion).

=>

```
Algorithm mergeSortLinkedList(head)
Sort singly linked list using merge sort
Pre: - head is head of linked list
Post: - Returns head of sorted list
1. If head = NULL OR head.next = NULL then
    1. Return head
2. End If
3. middle ← findMiddle(head)
4. nextToMiddle ← middle.next
5. middle.next ← NULL
6. left ← mergeSortLinkedList(head)
7. right ← mergeSortLinkedList(nextToMiddle)
8. sorted ← merge(left, right)
9. Return sorted
end mergeSortLinkedList
```

```
Algorithm findMiddle(head)
Find middle of linked list
1. slow ← head
2. fast ← head.next
3. While fast ≠ NULL AND fast.next ≠ NULL
    1. slow ← slow.next
    2. fast ← fast.next.next
4. End While
5. Return slow
end findMiddle
```

Q16.Differentiate between singly, doubly, and circular linked lists. Write algorithms for reversing a

doubly linked list.

=>

Difference Between Singly, Doubly, and Circular Linked Lists:

1. Singly Linked List (SLL): - Each node contains data and a pointer to the next node. - Traversal is possible only in one direction (head → tail). - Last node points to NULL.

2. Doubly Linked List (DLL): - Each node contains data, a pointer to the next node, and a pointer to the previous node. - Traversal is possible in both directions (head ↔ tail). - Useful for bidirectional traversal and easier deletion of nodes.

3. Circular Linked List (CLL): - Can be singly or doubly linked. - Last node points back to the first node (head), forming a circle. - Traversal can start at any node and continues indefinitely until stopped.

```
Algorithm reverseDoublyLL(head)

Reverse doubly linked list

Pre: - head is head of doubly linked list

Post: - Returns new head of reversed list

1. current ← head

2. temp ← NULL

3. While current ≠ NULL

1. temp ← current.prev

2. current.prev ← current.next

3. current.next ← temp

4. current ← current.prev

4. End While

5. If temp ≠ NULL then

1. head ← temp.prev

6. End If

7. Return head

end reverseDoublyLL
```

Differences: - Singly: One direction, less memory - Doubly: Two directions, more functionality - Circular: Last points to first, continuous

Q17. Write an algorithm to concatenate two singly linked lists. Analyze its time and space complexity.

=>

```
Algorithm concatenateLists(list1, list2)
Concatenate two singly linked lists
Pre: - list1, list2 are heads of two lists
Post: - Returns head of concatenated list
1. If list1 = NULL then
1. Return list2
2. End If
3. current ← list1
4. While current.next ≠ NULL
1. current ← current.next
5. End While
6. current.next ← list2
7. Return list1
end concatenateLists
```

Time Complexity Analysis:

1. Traversal of the first list to reach the last node → O(n1), where n1 = number of nodes in list1
2. Connecting last node of list1 to head2 → O(1)
3. Total time complexity = O(n1)

Space Complexity Analysis:

1. No extra nodes are created; only a temporary pointer is used → O(1)
2. Therefore, space complexity = O(1)

Time Complexity: O(n) where n is length of first list

Space Complexity: O(1)

Q18 . Define a priority queue and explain different methods of implementation. Compare its performance with a regular queue.

=>

Definition:

A Priority Queue is an abstract data structure where each element has a priority, and elements are

served based on their priority rather than just their insertion order. - Higher priority elements are dequeued before lower priority elements. - If two elements have the same priority, they may follow FIFO order.

Methods of Implementation:

1. Using Arrays (Unsorted): - Insert: O(1) → append at end - Delete (extract max/min): O(n) → scan entire array to find highest priority - Simple but slow for extraction

2. Using Arrays (Sorted): - Insert: O(n) → insert at correct position to maintain order - Delete (extract max/min): O(1) → remove first or last element - Faster extraction but slower insertion

3. Using Linked List: - Sorted linked list: - Insert: O(n) → traverse to find correct position - Delete: O(1) → remove head or tail - Unsorted linked list: - Insert: O(1) → append at end - Delete: O(n) → search for max/min

4. Using Binary Heap (Efficient Method): - Max-heap for max-priority, min-heap for min-priority - Insert: O(log n) - Delete (extract max/min): O(log n) - Maintains partial order efficiently

Conclusion: - Regular queue serves elements in FIFO order. - Priority queue serves elements based on priority.

- Using a heap, a priority queue is efficient for dynamic datasets with frequent insertions and

deletions.

```
Algorithm enqueuePriority(queue, item, priority)
Insert into priority queue
Pre: - queue is priority queue - item is data item - priority is priority value
Post: - Item inserted according to priority
1. Create new node with item and priority
2. If queue is empty OR priority < queue.front.priority then
1. Insert at front
3. Else
1. Traverse to find correct position
2. Insert node
4. End If
end enqueuePriority
Algorithm dequeuePriority(queue)
```

```
Remove highest priority item
1. If queue is empty then Return NULL
2. item ← queue.front.data
3. queue.front ← queue.front.next
4. Return item
end dequeuePriority
```

Comparison: - Regular Queue: FIFO, O(1) operations - Priority Queue: Priority-based, O(n) insertion

Q19. Explain the working principles of Quick Sort and Merge Sort. Write an algorithm for both and

analyze their time complexities.

=>

Algorithm 1: Quick Sort

Principle: - Divide and Conquer - Pick a pivot element from the array - Partition the array such that: - Elements less than pivot go to left - Elements greater than pivot go to right - Recursively apply Quick Sort on left and right subarrays

Pseudo-code:

```
QuickSort(A, low, high):
1. if low < high:
    pivotIndex = Partition(A, low, high)
    QuickSort(A, low, pivotIndex - 1)
    QuickSort(A, pivotIndex + 1, high)


Partition(A, low, high):
1. pivot = A[high]
2. i = low - 1
3. for j = low to high-1:
    if A[j] <= pivot:
      i = i + 1
      swap A[i], A[j]
4. swap A[i+1], A[high]
5. return i + 1
```

Time Complexity: - Best/Average case: O(n log n) → pivot splits array roughly in half - Worst case: O(n^2) → pivot is always smallest/largest element - Space Complexity: O(log n) due to recursion stack

---

Algorithm 2: Merge Sort

Principle: - Divide and Conquer - Divide array into two halves - Recursively sort each half - Merge two sorted halves into a single sorted array

Pseudo-code:

```
MergeSort(A, low, high):
1. if low < high:
      mid = (low + high)/2
      MergeSort(A, low, mid)
      MergeSort(A, mid+1, high)
      Merge(A, low, mid, high)

Merge(A, low, mid, high):
1. Create temporary arrays L[low..mid], R[mid+1..high]
2. i = 0, j = 0, k = low
3. while i < size(L) and j < size(R):
      if L[i] <= R[j]:
A[k] = L[i]; i++
else:
A[k] = R[j]; j++
k++
4. Copy remaining elements of L and R if any
```

Time Complexity: - All cases: O(n log n) - Space Complexity: O(n) due to temporary arrays - Stable sorting algorithm

Algorithm quickSort(arr, low, high)

```
Sort array using quick sort
Pre: - arr is array to sort - low, high are indices
Post: - arr sorted in range [low, high]
1. If low < high then
    1. pi ← partition(arr, low, high)
    2. quickSort(arr, low, pi - 1)
    3. quickSort(arr, pi + 1, high)
2. End If
end quickSort
```

```
Algorithm mergeSort(arr, left, right)
Sort array using merge sort
Pre: - arr is array to sort - left, right are indices
Post: - arr sorted in range [left, right]
1. If left < right then
    1. mid ← (left + right) / 2
    2. mergeSort(arr, left, mid)
    3. mergeSort(arr, mid + 1, right)
    4. merge(arr, left, mid, right)
2. End If
end mergeSort
```

Time Complexity: - Quick Sort: $O(n \log n)$ average, $O(n^2)$ worst - Merge Sort: $O(n \log n)$ all cases

Q20. Write algorithms for binary search (iterative and recursive). Trace the algorithm for a given

array and target value.

=>

```
Algorithm binarySearchIterative(arr, target)
Binary search iterative version
Pre: - arr is sorted array - target is value to find
Post: - Returns index if found, -1 otherwise
  1. left ← 0
  2. right ← length(arr) - 1
  3. While left ≤ right
    1. mid ← (left + right) / 2
    2. If arr[mid] = target then
      1. Return mid
    3. Else If arr[mid] < target then
      1. left ← mid + 1
    4. Else
      1. right ← mid - 1
    5. End If
  4. End While
  5. Return -1
end binarySearchIterative
```

```
Algorithm binarySearchRecursive(arr, left, right, target)
Binary search recursive version
  1. If left > right then Return -1
  2. mid ← (left + right) / 2
  3. If arr[mid] = target then
    1. Return mid
  4. Else If arr[mid] < target then
    1. Return binarySearchRecursive(arr, mid + 1, right, target)
  5. Else
    1. Return binarySearchRecursive(arr, left, mid - 1, target)
  6. End If
end binarySearchRecursive
```

Trace Example:

Array: [2, 5, 8, 12, 16, 23, 38, 45, 56, 72]

Target: 23

Iterative Trace:

1. low=0, high=9, mid=(0+9)/2=4, A*4+=16 < 23 → low=mid+1=5
2. low=5, high=9, mid=(5+9)/2=7, A*7+=45 > 23 → high=mid-1=6
3. low=5, high=6, mid=(5+6)/2=5, A*5+=23 == 23 → found at index 5

Recursive Trace:

1. BinarySearchRecursive(A, 0, 9, 23)

mid=4, A*4+=16 < 23 → search in right half (5..9)

2. BinarySearchRecursive(A, 5, 9, 23)

mid=7, A*7+=45 > 23 → search in left half (5..6)

3. BinarySearchRecursive(A, 5, 6, 23)

mid=5, A*5+=23 → found at index 5

Time Complexity: - O(log n) in all cases for sorted array

Space Complexity: - Iterative: O(1) - Recursive: O(log n) due to recursion stack

Q21.Write an algorithm a binary search tree (BST) insertion recursively. Analyze the time complexity for best, average, and worst cases.

=>

```
Algorithm insertBST(root, key)
Insert into Binary Search Tree recursively
Pre: - root is root of BST - key is value to insert
Post: - Returns root of updated BST
1. If root = NULL then
1. Return new Node(key)
2. End If
3. If key < root.data then
1. root.left ← insertBST(root.left, key)
4. Else If key > root.data then
1. root.right ← insertBST(root.right, key)
5. End If
```

6. Return root

end insertBST

Time Complexity Analysis:

Let n = number of nodes in BST

1. Best Case: - BST is perfectly balanced - Height h = log2(n) - Insert traverses height → O(log n)

2. Average Case: - Random insertion order - Average height ≈ log2(n) - Time complexity = O(log n)

3. Worst Case: - BST becomes skewed (all nodes in one side) - Height h = n - Time complexity = O(n)

Space Complexity: - Recursive call stack depth = height of BST - Best/Average case: O(log n) - Worst case: O(n)

Time Complexity: - Best Case: O(log n) - balanced tree - Average Case: O(log n) - Worst Case: O(n) - skewed tree

Q22.Write an algorithm for doubly linked list to maintain a student grade book. Include: insert,

delete, reverse traversal, and compute average marks.

=>

```
Algorithm insertStudent(head, name, rollNo, marks)
Insert student record
Pre: - head is head of list - Student details
Post: - Returns updated head
1. Create newStudent ← new Node(name, rollNo, marks)
2. If head = NULL then
1. Return newStudent
3. End If
4. newStudent.next ← head
5. head.prev ← newStudent
6. Return newStudent
end insertStudent
Algorithm deleteStudent(head, rollNo)
Delete student record
```

```
1. current ← head
2. While current ≠ NULL AND current.rollNo ≠ rollNo
    1. current ← current.next
3. End While
4. If current = NULL then Return head
5. If current.prev ≠ NULL then
    1. current.prev.next ← current.next
6. Else
    1. head ← current.next
7. End If
8. If current.next ≠ NULL then
    1. current.next.prev ← current.prev
9. End If
10. Delete current
11. Return head
end deleteStudent
Algorithm computeAverage(head)
Compute average marks
1. current ← head
2. sum ← 0
3. count ← 0
4. While current ≠ NULL
    1. sum ← sum + current.marks
    2. count ← count + 1
    3. current ← current.next
5. End While
6. If count > 0 then
    1. Return sum / count
7. Else
    1. Return 0
```

```
8. End If
end computeAverage
```

Q23.Write an algorithm to reverse a singly linked list in groups of size k. Show an example with

k=3.

=>

```
Algorithm reverseKGroup(head, k)
Reverse singly linked list in groups of size k
Pre: - head is head of linked list - k is group size
Post: - Returns head of reversed list
1. current ← head
2. next ← NULL
3. prev ← NULL
4. count ← 0
5. // Reverse first k nodes
6. While current ≠ NULL AND count < k
1. next ← current.next
2. current.next ← prev
3. prev ← current
4. current ← next
5. count ← count + 1
7. End While
8. // Recursively reverse remaining
9. If next ≠ NULL then
1. head.next ← reverseKGroup(next, k)
10. End If
11. Return prev
end reverseKGroup
```

Example:

Original List: 1 → 2 → 3 → 4 → 5 → 6 → 7 → 8 → 9

k = 3

Step-by-step:

1. First group (1,2,3): - Reverse → 3 → 2 → 1

2. Second group (4,5,6): - Reverse → 6 → 5 → 4

3. Third group (7,8,9): - Reverse → 9 → 8 → 7

Resulting List: 3 → 2 → 1 → 6 → 5 → 4 → 9 → 8 → 7

Time Complexity: - Each node visited exactly once → O(n)

Space Complexity: - Recursive stack depth ≈ n/k → O(n/k) extra space

Q24. Write an algorithm Fibonacci series and compare runtime with binary search on sorted arrays

of sizes 100.

=>

```
Algorithm fibonacci(n)

Generate Fibonacci series

Pre: - n is number of terms

Post: - Returns nth Fibonacci number

  1. If n ≤ 1 then
    1. Return n
  2. End If
  3. a ← 0
  4. b ← 1
  5. For i ← 2 to n
    1. c ← a + b
    2. a ← b
    3. b ← c
  6. End For
  7. Return b
end fibonacci
```

Runtime Comparison: - Fibonacci: O(n) time complexity - Binary Search: O(log n) time complexity - For n=100: Fibonacci ~100 operations, Binary Search ~7 operations

Q25.Write an algorithm for Heap Sort and count the number of comparisons and swaps for a sample array of 10 elements.

=>

```
Algorithm heapSortWithCount(arr)
Sort array using heap sort and count operations
Pre: - arr is array to sort
Post: - arr is sorted, returns comparison and swap counts
  1. comparisons ← 0
  2. swaps ← 0
  3. n ← length(arr)
  4. // Build max heap
  5. For i ← n/2 - 1 down to 0
    1. heapifyWithCount(arr, n, i, comparisons, swaps)
  6. End For
  7. // Extract elements
  8. For i ← n - 1 down to 1
    1. swaps ← swaps + 1
    2. Swap arr[0] and arr[i]
    3. heapifyWithCount(arr, i, 0, comparisons, swaps)
  9. End For
  10. Return comparisons, swaps
end heapSortWithCount
```

```
Algorithm heapifyWithCount(arr, n, i, comparisons, swaps)
1. largest ← i
2. left ← 2*i + 1
3. right ← 2*i + 2
4. If left < n then
1. comparisons ← comparisons + 1
2. If arr[left] > arr[largest] then
```

```
            1. largest ← left
        3. End If
    5. End If
    6. If right < n then
        1. comparisons ← comparisons + 1
        2. If arr[right] > arr[largest] then
            1. largest ← right
        3. End If
    7. End If
    8. If largest ≠ i then
        1. swaps ← swaps + 1
        2. Swap arr[i] and arr[largest]
        3. heapifyWithCount(arr, n, largest, comparisons, swaps)
    9. End If
end heapifyWithCount
```

For 10 elements: ~25-30 comparisons, ~15-20 swaps (approx)