Experiment 1 – Searching and Sorting

1. **Linear vs Binary Search:**
   Linear Search scans each element sequentially until the target is found — works on both sorted and unsorted data.
   Binary Search repeatedly divides a **sorted array** in half to locate the element quickly, giving O(log n) time complexity.
2. **Why Binary Search needs sorted data:**
   It compares the middle element and eliminates half of the list at each step.
   Without sorted order, the algorithm can't decide which half might contain the target element.
3. **Bubble Sort vs Selection Sort:**
   Bubble Sort repeatedly swaps adjacent elements until sorted — simple but slow.
   Selection Sort finds the smallest element and places it in order — fewer swaps.
   Both have $O(n^2)$ time, but Selection Sort performs better on large datasets.
4. **Role of realloc():**
   `realloc()` changes the size of an already allocated memory block dynamically.
   It preserves existing data and extends or shrinks memory without creating new pointers.
5. **Applications of sorting:**
   Sorting helps in searching contacts, ranking students, data visualization, file organization, and efficient database operations.

Experiment 2 – Stack (Infix to Postfix Conversion)

1. **Stack ADT:**
   A linear structure using **LIFO (Last-In, First-Out)** principle.
   Basic operations: `push()`, `pop()`, `peek()`, `isEmpty()`, and `isFull()`.
2. **Infix to Postfix Conversion:**
   Stack temporarily stores operators while operands are directly output.
   It ensures correct precedence and associativity using the Shunting Yard algorithm.
3. **Right-Associativity of ^ (Exponent):**
   Exponentiation is evaluated from right to left — `a^b^c` = `a^(b^c)`.
   So during conversion, right-associative operators are processed differently to maintain correct order.
4. **Handling Parentheses:**
   '(' is pushed to stack; when ')' is found, operators are popped until matching '('.
   This ensures sub-expressions are evaluated correctly.
5. **Example a + b * c:**
   Output sequence → a b c * +
   Explanation: * has higher precedence than +, so it's processed first.

6. **Postfix vs Prefix:**
   Postfix – operator follows operands (AB+), Prefix – operator precedes operands (+AB).
   Both remove the need for parentheses.
7. **Linked List vs Array Stack:**
   Linked list stack grows dynamically without overflow issues and uses memory efficiently.
   Array stack has fixed size and may overflow if not enough space.
8. **Multi-digit numbers & spaces:**
   Modify code to treat full numbers as single tokens and ignore whitespace when reading expressions.

Experiment 3 – Queue and Circular Queue

1. **Circular Queue:**
   Last position connects to the first to form a ring, efficiently using all spaces in the array.
2. **Linear vs Circular Queue:**
   Linear queue stops at the last index even if there's free space at front.
   Circular queue reuses space via wrap-around.
3. **Overflow & Underflow:**
   Overflow occurs when queue is full; underflow when it's empty and deletion is attempted.
   Both can be checked using front and rear pointers.
4. **Use of Modulo Operator:**
   Helps wrap the rear index back to 0 when it reaches the end, maintaining circular behavior.
5. **Applications:**
   Used in CPU process scheduling, printer spooling, traffic light control, and memory buffering.

Experiment 4 – Binary Search Tree (BST)

1. **Binary Search Tree (BST):**
   A binary tree where each node's left child < root < right child.
   Allows efficient searching, insertion, and deletion in O(log n) time.
2. **BST vs Binary Tree:**
   Binary Tree has no order rule; BST maintains a specific key order for quick access.
3. **Deletion in BST:**
   - If leaf → delete directly.
   - One child → link child to parent.
   - Two children → replace with inorder successor/predecessor, then delete that node.

1. **Tree Traversals:**
   - **Inorder (LNR):** Sorted output for BST.

- **Preorder (NLR):** Used to create copies of trees.
- **Postorder (LRN):** Used to delete trees.

1. **Depth of Tree:**
   Number of edges from root to the deepest leaf.
   It indicates the height and efficiency of the tree.
2. **Mirror of Tree:**
   Recursively swap left and right subtrees for all nodes to form the mirror image.
3. **Level-wise Traversal:**
   Implemented using a **queue** (Breadth-First Search).
   Visits all nodes level by level.
4. **Applications:**
   Used in databases, dictionaries, auto-suggest, file systems, and search engines.

Experiment 5 – Graphs and Minimum Spanning Tree

1. **Minimum Spanning Tree (MST):**
   Connects all vertices with minimum possible total edge weight and no cycles.
2. **Kruskal's vs Prim's Algorithm:**
   Kruskal's adds smallest edges globally; Prim's expands from one vertex.
   Kruskal uses sorting + Union-Find, Prim uses a priority queue.
3. **Graph Representation:**
   - **Adjacency Matrix:** 2D array, easy to check edges but uses more memory.
   - **Adjacency List:** Efficient, stores only connected vertices.

1. **Avoiding Cycles in MST:**
   Adding cycles increases total weight and violates tree property (V-1 edges).
2. **Applications:**
   Used in network design, road construction, telecommunication, and circuit layout optimization.
3. **Time Complexity:**
   Kruskal's – O(E log E); Prim's – O(E log V) using a priority queue.
4. **Union-Find in Kruskal:**
   Detects cycles and joins disjoint sets efficiently using union and find operations.
5. **Priority Queue in Prim:**
   Selects the minimum weighted edge connecting current MST to an unvisited vertex.

Experiment 6 – Heap Sort

Q1. What is a Heap?

A Heap is a **complete binary tree** that follows the **heap property**:

- In a **Max-Heap**, each parent node is greater than or equal to its children.

- In a **Min-Heap**, each parent node is smaller than or equal to its children.
  It is used to efficiently implement priority queues and heap sort.

Q2. What is Heap Sort?

Heap Sort is a **comparison-based sorting algorithm** that uses a binary heap structure to sort elements.

It builds a Max-Heap and repeatedly extracts the largest element to sort the array in ascending order.

Q3. Which type of heap is used for ascending order sorting?

A **Max-Heap** is used because the largest element is always at the root.

By repeatedly moving the root to the end and reducing the heap size, the array becomes sorted in ascending order.

Q4. What are the main steps of Heap Sort?

1. **Build a Max-Heap** from the given array.
2. **Swap** the first (largest) element with the last.
3. **Reduce** the heap size and call heapify again.
4. **Repeat** until the array is fully sorted.

Q5. What is the time complexity of Heap Sort?

The time complexity is **O(n log n)** in the best, average, and worst cases.

Building the heap takes O(n), and each of the n elements requires O(log n) for heapify.

Q6. What is the space complexity of Heap Sort?

The space complexity is **O(1)** since it sorts elements **in-place** without using extra memory.

Q7. Why is Heap Sort not stable?

Heap Sort is **not stable** because the relative order of equal elements may change during heapify operations due to swapping.

Q8. Applications of Heap Sort:

- Used in **priority queues** for scheduling tasks.
- **Job scheduling** in operating systems.
- Finding the **Kth largest or smallest element** in an array efficiently.

Q9. Advantages of Heap Sort:

- Provides **consistent O(n log n)** performance for all cases.
- **In-place sorting** (does not need extra memory).
- Suitable for **large datasets** where memory usage is critical.

Experiment 7 – Divide and Conquer (Merge Sort)

Q1. What is Divide and Conquer?

Divide and Conquer is a **problem-solving technique** that breaks a large problem into smaller subproblems, solves them independently, and then **combines their results** to form the final solution.

Merge Sort, Quick Sort, and Binary Search use this principle.

Q2. What is the basic idea of Merge Sort?

Merge Sort divides the array into halves until each subarray has one element, then **recursively merges** them in sorted order.

It ensures efficiency and stability for all data sizes.

Q3. What is the time complexity of Merge Sort?

It runs in **O(n log n)** time for best, average, and worst cases.

The array is divided log n times, and merging takes O(n) at each level.

Q4. Is Merge Sort a stable algorithm?

Yes ✓, Merge Sort is **stable**, meaning that equal elements retain their original relative order after sorting — important in record-based sorting.

Q5. What is the space complexity of Merge Sort?

Merge Sort needs **O(n)** extra space because temporary arrays are used to store divided sublists during merging.

Q6. Why is Merge Sort preferred for linked lists?

It works efficiently on linked lists since it doesn't require **random access**.

Nodes can be easily split and merged using pointers without extra space.

Q7. What is the merging process?

Merging combines **two sorted subarrays** into a single sorted array by repeatedly comparing and picking the smallest element from each list.

Q8. Compare Merge Sort and Quick Sort:

Parameter	Merge Sort	Quick Sort
Approach	Divide & Conquer	Divide & Conquer
Stability	Stable	Not Stable
Space	O(n)	O(log n)
Type	Non–in-place	In-place
Time	O(n log n)	O(n log n) avg, $O(n^2)$ worst

Q9. Applications of Merge Sort:

- Used in **external sorting** (sorting data from disks).
- **Database and file sorting** systems.
- Useful for large datasets that don't fit entirely in memory.

Q10. Advantages of Merge Sort:

- **Predictable O(n log n)** performance for all cases.
- **Stable** sorting method.
- Suitable for **linked lists** and large external data files.

Experiment 8 – Greedy Algorithm (Fractional Knapsack)

Q1. What is the Greedy Method?

It's a **problem-solving approach** where decisions are made by choosing the **best option at each step** (local optimum) in the hope of finding a **global optimum**.

It is simple, efficient, and widely used in optimization problems.

Q2. Difference between 0/1 Knapsack and Fractional Knapsack:

- **0/1 Knapsack:** You must take the entire item or leave it.
- **Fractional Knapsack:** Items can be **divided**, allowing partial selection to maximize profit.

Q3. What is the Greedy Choice Property?

This property means that making a **locally optimal choice at each step** leads to a **globally optimal solution**.

The Fractional Knapsack follows this property perfectly.

Q4. What is the Optimal Substructure Property?

A problem has an **optimal substructure** if the **optimal solution** of the main problem can be built from **optimal solutions of its subproblems**.

Both Knapsack problems exhibit this property.

Q5. Time Complexity of Fractional Knapsack:

The time complexity is **O(n log n)** because items are sorted based on their profit-to-weight ratio before selection.

Q6. Why does the Greedy approach work for Fractional Knapsack?

Because choosing items with the **highest profit/weight ratio first** always results in the **maximum possible profit**, ensuring an optimal solution.

Q7. Can Fractional Knapsack be solved using Dynamic Programming?

Yes, it can, but it's **not necessary**, as the Greedy approach already provides the **optimal solution** more efficiently.

Q8. Real-life Applications:

- **Cargo loading** and **resource allocation**.
- **Network bandwidth** or **memory optimization**.
- **Investment planning** or **portfolio management** for maximum returns.

Q9. Limitations of the Greedy Method:

- Doesn't always yield the global optimum for every problem (e.g., 0/1 Knapsack).
- Works only when the problem satisfies **Greedy Choice** and **Optimal Substructure** properties.

Q10. Advantages of Fractional Knapsack:

- **Fast and efficient** (O(n log n)).
- **Easy to implement.**
- Always yields **maximum profit** when fractional items are allowed.

Experiment 9 – Naïve String Matching Algorithm

Q1. What is String Matching?

String Matching is the process of finding all occurrences of a **pattern string** within a **text string**.

It's widely used in text editors, search engines, and DNA sequence analysis.

Q2. What is the Naïve String Matching Algorithm?

It is the **simplest pattern searching technique** that compares the pattern with every possible substring in the text, one character at a time.

Q3. What is the time complexity of the Naïve Algorithm?

- **Best Case:** O(n), when mismatches occur early.
- **Worst Case:** O(n × m), when almost all characters match except the last one.
  Here, n = text length, m = pattern length.

Q4. Is Naïve String Matching efficient?

No ✗, it's inefficient for **large texts or repetitive patterns** since it performs many unnecessary comparisons.

Q5. What are the limitations of Naïve String Matching?

- High time complexity (O(n × m))
- Not suitable for large datasets

- Doesn't use preprocessing like **KMP** or **Rabin–Karp** algorithms.

Q6. What are better alternatives to the Naïve Algorithm?

- **KMP Algorithm (Knuth–Morris–Pratt)** – avoids re-checking characters.
- **Rabin–Karp Algorithm –** uses hashing for faster comparison.
- **Boyer–Moore Algorithm** – skips sections using pattern heuristics.

Q7. What is the output of Naïve String Matching Algorithm?

It returns the **starting indices** where the pattern exactly matches the text.
Example: Text = "AABAACAADAABAABA", Pattern = "AABA" → Matches at **0, 9, 12**.

Q8. What is the space complexity of Naïve String Matching?

The algorithm uses **O(1)** space since it only needs a few variables for looping and comparison.

Q9. What is pattern shifting in String Matching?

After each mismatch or match, the **pattern shifts one position right** to check for new potential matches in the text.