1. Implementation of Stack Using Array

1. **Definition & Operations:** What is a stack and what are its primary operations (push, pop, peek)?
2. **Array Implementation:** How do you implement a stack using an array in C? What are the key steps?
3. **LIFO Principle:** Can you explain the Last In First Out (LIFO) concept and how it applies to a stack?
4. **Overflow & Underflow:** What do stack overflow and stack underflow mean? How can these be detected and handled in an array-based stack?
5. **Top Pointer:** What is the role of the top pointer in an array-based stack, and how is it updated during push and pop operations?
6. **Error Handling:** How would you design your functions to check if the stack is empty or full?
7. **Practical Application:** How can a stack be used to reverse a string or to check balanced parentheses?
8. **Memory Efficiency:** What are the limitations of using a static array for implementing a stack and how can dynamic resizing be achieved?
9. **Time Complexity:** What is the time complexity of the push, pop, and peek operations in an array-based implementation?
10. **Comparison:** How does an array-based implementation of a stack compare to a linked list-based implementation in terms of performance and memory usage?

2. Implementation of Queue Using Array

1. **Definition & Operations:** What is a queue and what are its core operations (enqueue, dequeue, peek)?
2. **FIFO Principle:** Explain the First In First Out (FIFO) principle and how it applies to queues.
3. **Array Implementation:** How can you implement a queue using an array in C? What are the main considerations?
4. **Linear vs. Circular:** What are the drawbacks of a simple linear queue implemented with an array, and why is a circular queue often preferred?
5. **Circular Queue Mechanics:** How do you implement a circular queue using an array? What role do the front and rear pointers play?
6. **Overflow & Underflow:** How do you detect and handle queue overflow and underflow conditions in an array implementation?
7. **Space Utilization:** How can the issue of wasted space in a simple array-based queue be minimized?
8. **Time Complexity:** What is the time complexity of enqueue, dequeue, and peek operations in your implementation?
9. **Dynamic Resizing:** Is it possible to implement a dynamically resizing queue using an array? How?

10. **Comparison:** Compare the benefits and drawbacks of an array-based queue with a linked list-based queue.

3. Infix to Postfix Conversion

1. **Expression Forms:** What are infix, postfix, and prefix expressions? Why might postfix be preferred for evaluation?
2. **Conversion Algorithm:** Describe the step-by-step algorithm for converting an infix expression to a postfix expression.
3. **Operator Precedence:** How is operator precedence and associativity managed during the conversion?
4. **Role of Stack:** Why is a stack the ideal data structure for converting infix to postfix expressions?
5. **Handling Parentheses:** How do you handle parentheses in the conversion process?
6. **Error Handling:** What potential errors can occur during the conversion (such as mismatched parentheses) and how can they be detected?
7. **Pseudo-code:** Can you provide a pseudo-code or flowchart for the conversion algorithm?
8. **Evaluation:** Once converted, how would you evaluate a postfix expression?
9. **Complexity:** What is the time and space complexity of your conversion algorithm?
10. **Edge Cases:** How would your algorithm handle an expression with unary operators or other edge cases?

4. Doubly Linked List Implementation Using Structure

1. **Definition:** What is a doubly linked list and how does it differ from a singly linked list?
2. **Node Structure:** Describe the structure of a node in a doubly linked list.
3. **Traversal:** How do you traverse a doubly linked list in both forward and reverse directions?
4. **Insertion:** How do you insert a node at the beginning, middle, and end of a doubly linked list?
5. **Deletion:** Explain the process of deleting a node from a doubly linked list. What special cases need to be handled?
6. **Memory Management:** How do you manage dynamic memory allocation and deallocation for nodes in a doubly linked list?
7. **Advantages:** What are the advantages of using a doubly linked list over a singly linked list?
8. **Reversal:** How would you reverse a doubly linked list? Is it easier than reversing a singly linked list?
9. **Error Scenarios:** What potential pitfalls or errors (e.g., pointer mismanagement) should be considered when implementing a doubly linked list?
10. **Search Operation:** How do you search for an element in a doubly linked list and what is its time complexity?

5. Singly Linked List Using Structure

1. **Definition:** What is a singly linked list and what is its basic structure?
2. **Creation & Initialization:** How do you create and initialize a singly linked list using structures in C?
3. **Insertion at Beginning:** Describe the process of inserting a node at the beginning of a singly linked list.
4. **Insertion at End:** How do you insert a node at the end of a singly linked list?
5. **Deletion:** Explain how to delete a node from a singly linked list. What challenges might you encounter?
6. **Traversal:** How do you traverse a singly linked list to print all its elements?
7. **Reversal:** What is the algorithm to reverse a singly linked list?
8. **Comparison with Arrays:** What are the advantages and disadvantages of using a singly linked list compared to an array?
9. **Loop Detection:** How can you detect a loop in a singly linked list? Can you describe any algorithm for it?
10. **Time Complexity:** What is the time complexity of insertion, deletion, and search operations in a singly linked list?

6. Recursion (Fibonacci, Factorial, Tower of Hanoi)

1. **Basic Concept:** What is recursion? Explain with a simple example.
2. **Factorial Calculation:** How do you implement the factorial function recursively? What is the base case?
3. **Fibonacci Series:** Describe the recursive approach to generating the Fibonacci series. What are its drawbacks?
4. **Tower of Hanoi:** Explain the recursive solution to the Tower of Hanoi problem. How does the algorithm work?
5. **Base Case Importance:** Why is having a base case important in any recursive algorithm?
6. **Stack Usage:** How does recursion use the call stack and what are the implications for memory usage?
7. **Performance Issues:** Compare the performance of recursive vs. iterative solutions, particularly for the Fibonacci series.
8. **Optimization:** What techniques can be used to optimize recursive algorithms (e.g., memoization)?
9. **Conversion:** How would you convert a recursive algorithm into an iterative one? Provide an example.
10. **Debugging:** What strategies do you use to debug recursive functions?

7. Sorting (Bubble, Insertion, Selection)

1. **Purpose of Sorting:** Why is sorting an important operation in computer science?
2. **Bubble Sort Mechanics:** How does bubble sort work? Explain its basic algorithm and time complexity.
3. **Insertion Sort Mechanics:** Describe the insertion sort algorithm. How does it differ from bubble sort?
4. **Selection Sort Mechanics:** Explain how selection sort works and its main characteristics.
5. **Comparative Complexity:** Compare the best-case, average-case, and worst-case time complexities of bubble, insertion, and selection sort.
6. **Space Complexity:** What is the space complexity of these sorting algorithms?
7. **Stability:** Discuss the concept of stability in sorting algorithms. Which of these algorithms are stable?
8. **Performance on Nearly Sorted Data:** How do these algorithms perform when the input data is nearly sorted?
9. **Step-by-Step Example:** Can you walk through a step-by-step example of one of these sorting algorithms?
10. **Practical Applications:** In what scenarios might you choose one of these simple sorting algorithms over more complex ones like quicksort or mergesort?