Q1. What is recursion? Explain with an example.

Answer:
Recursion is a programming technique where a function calls itself to solve smaller sub-problems. It consists of a **base case** and a **recursive case**.
Example (Factorial):

```
int factorial(int n) {
  if (n == 0) return 1;
  else return n * factorial(n - 1);
}
```

Q2. Differentiate between recursion and iteration.

Answer:

Feature	Recursion	Iteration
Approach	Function calls itself	Loops (for, while)
Memory	More (stack frames)	Less
Speed	Slower (overhead)	Faster
Example	Factorial using recursion	Factorial using loop
Use Case	Tree traversal	Counting, summing

Q3. What are the essential components of a recursive function?

Answer:

1. **Base Case** – Stops recursion (e.g., if (n == 0) return 1;)
2. **Recursive Case** – Function calls itself with a smaller input.
3. **Progress toward Base Case** – Ensures termination (e.g., n - 1).

Q4. Explain tail recursion with an example.

Answer:
In tail recursion, the recursive call is the last operation.
Example:

```
int tailFactorial(int n, int acc) {
  if (n == 0) return acc;
  return tailFactorial(n - 1, acc * n);
}
```

Tail recursion is more memory-efficient and can be optimized by compilers.

Q5. What are the advantages and disadvantages of recursion?

Answer:
Advantages:

- Simplifies complex problems like tree traversal, backtracking.

Disadvantages:

- More memory (stack usage).
- Slower due to function calls.
- Risk of stack overflow if base case is missing.

🔢 FACTORIAL & FIBONACCI SERIES

Q6. Write a recursive function to calculate the factorial of a number and explain.

Answer:

```
int factorial(int n) {
  if (n == 0) return 1;
  return n * factorial(n - 1);
}
```

Explanation:

- Base case: factorial(0) = 1
- Recursive case: factorial(n) = n * factorial(n - 1)
  Each call multiplies current n with result of smaller factorial.

Q7. Write a recursive program to generate the nth Fibonacci number.

Answer:

```
int fibonacci(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Q8. Explain the drawbacks of the recursive Fibonacci approach.

Answer:

- **Redundant calls**: Many values are recomputed multiple times.
- **Time Complexity**: Exponential → $O(2^n)$
- **Memory overhead**: Deep recursion uses large call stack.

Solution: Use **Memoization** or **Iterative** version for efficiency.

🧠 **ACKERMANN FUNCTION**

Q9. What is the Ackermann function? Why is it significant?

Answer:
The Ackermann function is a **non-primitive recursive** function that grows very fast and cannot be expressed using loops.

A(m, n) =

n + 1 if m = 0

A(m - 1, 1) if m > 0, n = 0

A(m - 1, A(m, n-1)) if m > 0, n > 0

It demonstrates the power of recursion beyond what loops can do.

Q10. Write a recursive program for the Ackermann function.

Answer:

```
int ackermann(int m, int n) {
   if (m == 0) return n + 1;
   if (n == 0) return ackermann(m - 1, 1);
   return ackermann(m - 1, ackermann(m, n - 1));
}
```

⚡ QUICK SORT

Q11. Explain the working of Quick Sort using recursion.

Answer:

1. Choose a pivot.
2. Partition elements around pivot (smaller left, larger right).
3. Recursively sort left and right partitions.

Q12. Write a recursive program for Quick Sort.

Answer:

```
int partition(int arr[], int low, int high) {
   int pivot = arr[high], i = low - 1;
   for (int j = low; j < high; j++) {
      if (arr[j] < pivot) {
         i++; swap(&arr[i], &arr[j]);
      }
   }
   swap(&arr[i + 1], &arr[high]);
   return i + 1;
}
```

```
void quickSort(int arr[], int low, int high) {
  if (low < high) {
    int pi = partition(arr, low, high);
    quickSort(arr, low, pi - 1);
    quickSort(arr, pi + 1, high);
  }
}
```

Q13. Trace Quick Sort on array [10, 7, 8, 9, 1, 5].

Answer:

- Pivot = 5 → Partition → [1] [5] [10, 7, 8, 9]
- Recursively sort left and right parts
  Final sorted: [1, 5, 7, 8, 9, 10]

🧮 MERGE SORT

Q14. Explain merge sort with recursion.

Answer:

1. Divide array into two halves
2. Recursively sort each half
3. Merge the two sorted halves

Merge sort is a classic **Divide and Conquer** algorithm.

Q15. Write a recursive program for merge sort.

Answer:

```
void merge(int arr[], int l, int m, int r) {
  // merge logic with temp arrays
}
```

```
void mergeSort(int arr[], int l, int r) {
  if (l < r) {
    int m = (l + r) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
  }
}
```

Q16. Compare Quick Sort and Merge Sort.

Feature	Merge Sort	Quick Sort
Time (Best/Worst)	O(n log n)	O(n log n)/O(n^2)
Space	O(n)	O(log n)
Stability	Stable	Not Stable
Approach	Divide-Merge	Partitioning

📚 **ADDITIONAL THEORY QUESTIONS**

Q17. What is base case failure in recursion? Give example.

Answer:
Base case failure occurs when the termination condition is missing or incorrect, leading to infinite recursion and stack overflow.

Example:

```
int faulty(int n) {
  return n * faulty(n - 1); // No base case
}
```

Q18. What is stack overflow? How can it be avoided in recursion?

Answer:
Stack overflow occurs when too many recursive calls exhaust the stack memory.
Avoid by:

- Ensuring base case is correct
- Using iterative alternatives
- Applying tail recursion optimization

Q19. What is the time and space complexity of merge sort and quick sort?

Answer:

Sort	Time Complexity	Space Complexity
Merge Sort	O(n log n)	O(n)
Quick Sort	Avg: O(n log n), Worst: $O(n^2)$	O(log n) (stack space)

Q20. What is divide and conquer? Which sorting algorithms use it?

Answer:
Divide and Conquer breaks a problem into smaller subproblems, solves them independently, and combines results.
Used in: Merge Sort, Quick Sort

Q21. Write a recursive function to find the sum of digits of a number.

```
int sumDigits(int n) {
  if (n == 0) return 0;
  return (n % 10) + sumDigits(n / 10);
}
```

PART - 2

1. **What are the two main parts of a recursive function?**
   Answer:
   - Base Case: Terminates the recursion.
   - Recursive Case: Function calls itself with modified parameters.
2. **Explain the difference between recursion and iteration.**
   Answer:
   - **Recursion** uses function calls and the call stack; suitable for problems divisible into subproblems (e.g., tree traversal).
   - **Iteration** uses loops; more memory-efficient and faster for simple tasks.
3. **What is tail recursion? Give an example.**
   Answer: A tail-recursive function is where the recursive call is the last thing executed.
   Example:

```
int tailFactorial(int n, int acc = 1) {
  if (n == 0) return acc;
  return tailFactorial(n - 1, acc * n);
}
```

4. **What are the advantages and disadvantages of recursion?**
   Answer:
   Advantages: Cleaner code for complex problems (e.g., trees).
   Disadvantages: Higher memory usage, possible stack overflow.

🧮 Factorial Using Recursion

6. **Write a recursive program to find the factorial of a number.**
   Code same as Q1.
7. **Trace the execution of factorial(4) using recursion.**
   Answer:
   factorial(4) → 4 × factorial(3)
   factorial(3) → 3 × factorial(2)
   factorial(2) → 2 × factorial(1)

factorial(1) → 1 × factorial(0)
factorial(0) → 1
Final answer: 4 × 3 × 2 × 1 × 1 = 24

🔢 Fibonacci Series Using Recursion

8. **Write a recursive program to print the nth Fibonacci number.**

```
int fibonacci(int n) {
  if (n == 0) return 0;
  if (n == 1) return 1;
  return fibonacci(n - 1) + fibonacci(n - 2);
}
```

9. **What is the time complexity of recursive Fibonacci?**
   Answer: Exponential, O(2^n), due to repeated calls.
10. **How can the recursive Fibonacci be optimized?**
   Answer:

- Use **memoization** (store previous results)
- Or use **iterative approach**

🧠 Ackermann Function

11. **Define the Ackermann function.**
   Answer:
   A classic example of a recursive function that is **not primitive recursive**.

```
A(m, n) =
  n + 1              if m = 0
  A(m - 1, 1)        if m > 0 and n = 0
  A(m - 1, A(m, n - 1))   if m > 0 and n > 0
```

12. **Write a recursive program to implement the Ackermann function.**

```
int ackermann(int m, int n) {
  if (m == 0) return n + 1;
```

```
  if (n == 0) return ackermann(m - 1, 1);
  return ackermann(m - 1, ackermann(m, n - 1));
}
```

13. **Why is the Ackermann function significant in recursion?**
    Answer: It grows very fast and shows how recursion can express powerful computations that can't be done with loops alone.

🧮 Sorting Using Recursion

🟢 Merge Sort

14. **Explain merge sort algorithm using recursion.**
    Answer:

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

15. **Write a recursive program for merge sort.**
    (Include only function header and logic for merge + recursive split if needed.)

16. **What is the time and space complexity of merge sort?**
    Answer:

- Time: O(n log n)
- Space: O(n)

🔴 Quick Sort

17. **Explain quick sort using recursion.**
    Answer:

- Choose a pivot.
- Partition the array around the pivot.
- Recursively sort subarrays on both sides.

18. **Write a recursive program for quick sort.**
    (Show function with partitioning and recursive calls.)

🧠 Conceptual and Tracing Questions

20. **Trace the merge sort process on array [38, 27, 43, 3, 9, 82, 10].**

21. **Trace the quick sort partitioning step with pivot = last element.**

22. **Explain base case failure in recursion with example.**

23. **What is stack overflow in recursion? When does it happen?**
    Answer: Happens when recursion goes too deep, exceeding call stack size.

24. **What is a recursive data structure? Give examples.**
    Answer: A structure that contains a reference to itself.
    Example: Linked lists, trees.

25. **How does the system keep track of recursive calls?**
    Answer: Using the **call stack**; each call pushes new frame, popped when returning.
