Unit 1-2

Data Structures Units 1-3 Answers

Unit 1: Basic of Data Structures

1. What is Data Structure and explain various operations on data structures.

A data structure is a way of organizing and storing data in computer memory to enable efficient access and modification. It defines the relationship between data elements and the operations that can be performed on them.

Basic Operations on Data Structures:

1. Traversal: Accessing each element of the data structure exactly once for processing.

2. Searching: Finding the location of an element with a given value or key.

3. Insertion: Adding a new element to the data structure at a specific position.

4. Deletion: Removing an element from a specific position in the data structure.

5. Sorting: Arranging elements in ascending or descending order based on some criteria.

6. Merging: Combining elements from two similar data structures into one.

7. Splitting: Dividing a single data structure into multiple structures.

8. Update/Modification: Changing the value of an existing element.

These operations form the foundation for all data manipulation tasks and their efficiency determines the overall performance of algorithms and applications.

2. What is Data Structure and explain various types of data structures.

A data structure is a systematic way of organizing, managing, and storing data to facilitate efficient access, modification, and manipulation. It provides a means to manage large amounts of data efficiently.

Classification of Data Structures:

1. Primitive Data Structures:

- Integer, Float, Character, Boolean
- Basic data types provided by programming languages

- Directly operated upon by machine instructions

2. Non-Primitive Data Structures:

Linear Data Structures:

- Array: Elements stored in contiguous memory locations
- Linked List: Elements connected through pointers
- Stack: LIFO (Last In First Out) principle
- Queue: FIFO (First In First Out) principle

Non-Linear Data Structures:

- Tree: Hierarchical structure with parent-child relationships
- Graph: Network of vertices connected by edges
- Hash Table: Key-value pairs with hash function mapping

Static vs Dynamic:

- Static: Fixed size (Arrays)
- Dynamic: Variable size (Linked Lists, Trees)

3. What is algorithm? Explain its characteristics.

An algorithm is a finite set of well-defined instructions or steps designed to solve a specific problem or perform a particular task. It provides a systematic approach to problem-solving with clear input and output specifications.

Characteristics of Algorithm:

1. Finiteness: Algorithm must terminate after a finite number of steps. It cannot run indefinitely.

2. Definiteness: Each step must be clearly and unambiguously defined. No ambiguous instructions allowed.

3. Input: Algorithm can have zero or more inputs taken from a specified set of objects.

4. Output: Algorithm produces one or more outputs having a specific relation to inputs.

5. Effectiveness: All operations must be basic enough to be carried out exactly and in finite time.

6. Generality: Algorithm should be applicable to a set of inputs, not just specific cases.

7. Feasibility: Algorithm should be implementable with available resources and within reasonable time.

8. Independent: Algorithm should be independent of any programming language or computing platform.

These characteristics ensure that algorithms are reliable, understandable, and implementable solutions to computational problems.

4. What is algorithm? Write an algorithm to find factorial of given number.

An algorithm is a step-by-step procedure to solve a specific problem, consisting of finite, well-defined instructions that produce desired output from given input.

Algorithm to Find Factorial:

```
ALGORITHM: FACTORIAL
INPUT: N (a non-negative integer)
OUTPUT: FACT (factorial of N)

STEP 1: START
STEP 2: READ N
STEP 3: IF N < 0 THEN
        PRINT "Invalid input"
        GO TO STEP 8
STEP 4: SET FACT = 1
STEP 5: SET I = 1
STEP 6: REPEAT WHILE I ≤ N
        SET FACT = FACT * I
        SET I = I + 1
STEP 7: PRINT "Factorial of", N, "is", FACT
STEP 8: STOP
```

Example: For N = 5:

- FACT = 1 * 1 = 1
- FACT = 1 * 2 = 2
- FACT = 2 * 3 = 6
- FACT = 6 * 4 = 24
- FACT = 24 * 5 = 120

Time Complexity: O(N) **Space Complexity:** O(1)

5. What is algorithm? Write an algorithm to find greater number among three numbers.

An algorithm is a finite sequence of well-defined steps that solves a particular problem by transforming input into desired output through logical operations.

Algorithm to Find Greatest of Three Numbers:

```
ALGORITHM: FIND_GREATEST
INPUT: A, B, C (three numbers)
OUTPUT: GREATEST (largest among three)

STEP 1: START
STEP 2: READ A, B, C
STEP 3: IF A > B THEN
        IF A > C THEN
            SET GREATEST = A
        ELSE
            SET GREATEST = C
        END IF
        ELSE
        IF B > C THEN
            SET GREATEST = B
        ELSE
            SET GREATEST = C
        END IF
STEP 4: PRINT "Greatest number is", GREATEST
STEP 5: STOP
```

Alternative Method:

```
STEP 3: SET GREATEST = A
STEP 4: IF B > GREATEST THEN
        SET GREATEST = B
STEP 5: IF C > GREATEST THEN
        SET GREATEST = C
```

Example: For A=15, B=25, C=10, GREATEST = 25

6. Explain the difference between static and dynamic data structures.

Static Data Structures: Data structures with fixed size that is determined at compile time. Memory allocation is done during compilation and remains constant throughout program execution.

Characteristics:

- Fixed size declared at compile time
- Memory allocated in stack or data segment
- Size cannot be changed during runtime
- Faster access due to compile-time memory allocation
- Memory wastage if not fully utilized
- Limited flexibility

Examples: Arrays, Records/Structures

Dynamic Data Structures: Data structures whose size can change during runtime. Memory is allocated and deallocated as needed during program execution.

Characteristics:

- Variable size that can grow or shrink
- Memory allocated in heap during runtime
- Size can be modified during program execution
- Slower access due to runtime memory management
- Efficient memory utilization
- Greater flexibility and adaptability

Examples: Linked Lists, Trees, Stacks (using linked lists), Dynamic Arrays

Key Differences:

- Memory allocation timing: Compile-time vs Runtime
- Size flexibility: Fixed vs Variable
- Memory efficiency: May waste space vs Optimal usage
- Access speed: Faster vs Relatively slower

7. Differentiate between linear and non-linear data structures with examples

Linear Data Structures: Data elements are arranged in sequential order where each element has a unique predecessor and successor (except first and last elements). Elements are accessed in a linear sequence.

Characteristics:

- Elements stored in sequential manner
- Only one way to traverse (sequential)
- Each element connected to its previous and next element
- Memory utilization is not efficient
- Implementation is easier

Examples:

- **Array:** [10, 20, 30, 40, 50]
- **Linked List:** 10 → 20 → 30 → NULL
- **Stack:** Push/Pop from top only
- **Queue:** Insert at rear, delete from front

Non-Linear Data Structures: Data elements are not arranged in sequential order. Elements are connected in hierarchical or network manner with multiple paths for traversal.

Characteristics:

- Elements not stored sequentially
- Multiple ways to traverse
- Elements connected in hierarchical manner
- Efficient memory utilization
- Complex implementation

Examples:

- **Tree:** Hierarchical structure with parent-child relationship
- **Graph:** Network of vertices connected by edges

Key Differences:

- Traversal: Sequential vs Multiple paths
- Memory usage: Less efficient vs More efficient
- Implementation: Simple vs Complex

8. Define algorithm complexity and explain best case, average case and worst case complexity with examples.

Algorithm complexity measures the amount of computational resources (time and space) required by an algorithm as a function of input size. It helps analyze and compare algorithm efficiency.

Types of Complexity Analysis:

1. Best Case Complexity: Minimum time/space required when input is most favorable. Represents optimal scenario for algorithm performance.

2. Average Case Complexity: Expected time/space required for typical random input. Considers all possible inputs and their probabilities.

3. Worst Case Complexity: Maximum time/space required when input is least favorable. Represents performance guarantee under all conditions.

Example - Linear Search:

Best Case: O(1)

- Element found at first position
- Only one comparison needed

- Input: [5, 10, 15, 20], Search: 5

Average Case: O(n/2) → O(n)

- Element found at middle position on average
- Approximately n/2 comparisons needed

Worst Case: O(n)

- Element at last position or not found
- All n elements must be examined
- Input: [5, 10, 15, 20], Search: 20 or 25

Worst case analysis is most commonly used as it provides performance guarantee.

9. What is the difference between time complexity and space complexity? Provide examples.

Time Complexity: Measures the amount of computational time an algorithm takes to complete as a function of input size. It indicates how execution time grows with increasing input.

Space Complexity: Measures the amount of memory space an algorithm requires during execution as a function of input size. It includes both auxiliary space and input space.

Key Differences:

Time Complexity:

- Measures execution time
- Analyzed by counting operations
- Critical for performance optimization
- Affected by processor speed and system load
- Cannot be improved by adding more memory

Space Complexity:

- Measures memory usage
- Analyzed by counting memory allocations
- Critical for memory-constrained systems
- Independent of processor characteristics
- Can sometimes trade-off with time complexity

Examples:

Linear Search:

- Time Complexity: O(n) - may need to check all elements
- Space Complexity: O(1) - only uses few variables

Merge Sort:

- Time Complexity: O(n log n) - divide and conquer approach
- Space Complexity: O(n) - requires additional array for merging

Matrix Addition (n×n matrices):

- Time Complexity: $O(n^2)$ - nested loops for all elements
- Space Complexity: $O(n^2)$ - stores result matrix

Recursive Fibonacci:

- Time Complexity: $O(2^n)$ - exponential due to repeated calculations
- Space Complexity: O(n) - recursion stack depth

10. How to calculate time complexity of an algorithm. Explain with the help of example.

Time complexity calculation involves counting the number of basic operations performed by an algorithm as a function of input size 'n'. Focus on dominant operations that contribute most to execution time.

Steps to Calculate Time Complexity:

1. **Identify basic operations:** Assignments, comparisons, arithmetic operations
2. **Count operations in each statement**
3. **Analyze loops and their iterations**
4. **Consider nested structures**
5. **Find dominant term and drop constants**
6. **Express in Big-O notation**

Example - Bubble Sort Algorithm:

```
for i = 0 to n-1:           // Outer loop: n iterations
    for j = 0 to n-i-2:     // Inner loop: (n-1), (n-2), ..., 1 iterations
        if arr[j] > arr[j+1]:   // Comparison: 1 operation
            swap(arr[j], arr[j+1])  // Swap: 3 operations
```

Analysis:

- Outer loop executes n times
- Inner loop executes: (n-1) + (n-2) + ... + 1 = n(n-1)/2 times
- Each inner loop iteration: 1 comparison + 3 swaps (worst case) = 4 operations

- Total operations: $n \times n(n-1)/2 \times 4 = 2n^2(n-1) = 2n^3 - 2n^2$

Dominant term: $2n^3$ **Dropping constants:** n^3 **Time Complexity:** $O(n^3) \rightarrow$ Actually $O(n^2)$ for bubble sort

Common Time Complexities:

- O(1): Constant time
- O(log n): Logarithmic time
- O(n): Linear time
- O(n log n): Linearithmic time
- $O(n^2)$: Quadratic time

Unit 2: Searching and Sorting Techniques

1. Write an algorithm for linear search and explain its working using example.

Linear search is a sequential search algorithm that checks each element of the array one by one until the target element is found or the end of array is reached.

Algorithm:

```
LINEAR_SEARCH(ARR, N, ITEM)
STEP 1: SET I = 0
STEP 2: REPEAT WHILE I < N
STEP 3:   IF ARR[I] = ITEM THEN
STEP 4:     RETURN I
STEP 5:   SET I = I + 1
STEP 6: RETURN -1 (Item not found)
```

Working Example: Array: [15, 23, 8, 42, 31, 7, 19] Search for ITEM = 42

Step-by-step execution:

- I = 0: ARR[0] = 15 ≠ 42, continue
- I = 1: ARR[1] = 23 ≠ 42, continue
- I = 2: ARR[2] = 8 ≠ 42, continue
- I = 3: ARR[3] = 42 = 42, found! Return index 3

Characteristics:

- Works on both sorted and unsorted arrays
- Time Complexity: O(n)
- Space Complexity: O(1)
- Best Case: O(1) - element at first position

- Worst Case: O(n) - element at last position or not found
- Simple implementation but inefficient for large datasets

2. Write an algorithm for binary search and explain its working using example.

Binary search is an efficient search algorithm that works on sorted arrays by repeatedly dividing the search interval in half and comparing the target with the middle element.

Algorithm:

```
BINARY_SEARCH(ARR, LOW, HIGH, ITEM)
STEP 1: IF LOW > HIGH THEN
STEP 2:   RETURN -1
STEP 3: SET MID = (LOW + HIGH) / 2
STEP 4: IF ARR[MID] = ITEM THEN
STEP 5:   RETURN MID
STEP 6: ELSE IF ARR[MID] > ITEM THEN
STEP 7:   RETURN BINARY_SEARCH(ARR, LOW, MID-1, ITEM)
STEP 8: ELSE
STEP 9:   RETURN BINARY_SEARCH(ARR, MID+1, HIGH, ITEM)
```

Working Example: Sorted Array: [5, 12, 18, 23, 31, 45, 67, 89] Search for ITEM = 31

Execution:

- LOW=0, HIGH=7, MID=3: ARR[3]=23 < 31, search right half
- LOW=4, HIGH=7, MID=5: ARR[5]=45 > 31, search left half
- LOW=4, HIGH=4, MID=4: ARR[4]=31 = 31, found! Return 4

Characteristics:

- Requires sorted array
- Time Complexity: O(log n)
- Space Complexity: O(1) iterative, O(log n) recursive
- Much more efficient than linear search for large datasets

3. Differentiate between linear and Binary Search

Linear Search:

Working Principle: Sequential examination of each element from beginning to end until target is found or array ends.

Prerequisites: No special requirement; works on both sorted and unsorted arrays.

Time Complexity:

- Best Case: O(1) - element at first position
- Average Case: O(n/2) = O(n)
- Worst Case: O(n) - element at last position

Space Complexity: O(1) - uses constant extra space

Implementation: Simple and straightforward

Use Cases: Small datasets, unsorted arrays, when simplicity is preferred

Binary Search:

Working Principle: Divide and conquer approach; repeatedly divides search space in half by comparing with middle element.

Prerequisites: Array must be sorted in ascending or descending order.

Time Complexity:

- Best Case: O(1) - element at middle position
- Average Case: O(log n)
- Worst Case: O(log n)

Space Complexity: O(1) iterative, O(log n) recursive

Implementation: More complex due to boundary conditions

Use Cases: Large sorted datasets, when efficiency is critical

Key Differences:

- Efficiency: Linear O(n) vs Binary O(log n)
- Array requirement: Any vs Sorted only
- Complexity: Simple vs Complex implementation

6. Write algorithm for binary search. Apply the algorithm on the array A containing 9, 17, 23, 38, 45, 50, 57, 76, 90, 100 to search items 10 and 100.

Binary Search Algorithm:

```
BINARY_SEARCH(A, N, ITEM)
STEP 1: SET LOW = 0, HIGH = N-1
STEP 2: REPEAT WHILE LOW ≤ HIGH
STEP 3:   SET MID = (LOW + HIGH) / 2
STEP 4:   IF A[MID] = ITEM THEN
STEP 5:     RETURN MID
STEP 6:   ELSE IF A[MID] > ITEM THEN
STEP 7:     SET HIGH = MID - 1
STEP 8:   ELSE
STEP 9:     SET LOW = MID + 1
STEP 10: RETURN -1
```

Array A: [9, 17, 23, 38, 45, 50, 57, 76, 90, 100] **Indices:** [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

Search for ITEM = 10:

- LOW=0, HIGH=9, MID=4: A[4]=45 > 10, search left half
- LOW=0, HIGH=3, MID=1: A[1]=17 > 10, search left half
- LOW=0, HIGH=0, MID=0: A[0]=9 < 10, search right half
- LOW=1, HIGH=0: LOW > HIGH, item not found, return -1

Search for ITEM = 100:

- LOW=0, HIGH=9, MID=4: A[4]=45 < 100, search right half
- LOW=5, HIGH=9, MID=7: A[7]=76 < 100, search right half
- LOW=8, HIGH=9, MID=8: A[8]=90 < 100, search right half
- LOW=9, HIGH=9, MID=9: A[9]=100 = 100, found! Return 9

Results: ITEM 10 not found (-1), ITEM 100 found at index 9.

7. What do you mean by sorting? List all sorting techniques. Write an algorithm of bubble sort.

Sorting is the process of arranging data elements in a particular order, typically ascending or descending, based on some comparison criteria. It's a fundamental operation in computer science that makes data searching and processing more efficient.

Classification of Sorting Techniques:

Internal Sorting (In-memory):

- Bubble Sort, Selection Sort, Insertion Sort
- Quick Sort, Merge Sort, Heap Sort
- Shell Sort, Radix Sort, Counting Sort

External Sorting (File-based):

- Used when data is too large for memory
- Merge-based external sorting

Stable vs Unstable:

- Stable: Maintains relative order of equal elements
- Unstable: May change relative order of equal elements

Bubble Sort Algorithm:

```
BUBBLE_SORT(A, N)
STEP 1: FOR I = 0 TO N-2 DO
STEP 2:   FOR J = 0 TO N-I-2 DO
STEP 3:     IF A[J] > A[J+1] THEN
STEP 4:       TEMP = A[J]
STEP 5:       A[J] = A[J+1]
STEP 6:       A[J+1] = TEMP
STEP 7:   END FOR
STEP 8: END FOR
```

Characteristics:

- Time Complexity: $O(n^2)$
- Space Complexity: O(1)
- Stable sorting algorithm
- In-place sorting
- Simple but inefficient for large datasets

9. Write an algorithm for bubble sort with example.

Bubble Sort Algorithm:

```
BUBBLE_SORT(ARR, N)
STEP 1: FOR I = 0 TO N-2 DO
STEP 2:   FOR J = 0 TO N-I-2 DO
STEP 3:     IF ARR[J] > ARR[J+1] THEN
STEP 4:       SWAP ARR[J] AND ARR[J+1]
STEP 5:   END FOR
STEP 6: END FOR
```

Working Principle: Bubble sort repeatedly compares adjacent elements and swaps them if they are in wrong order. The largest element "bubbles" to the end in each pass.

Example: Sort array [64, 34, 25, 12, 22, 11, 90]

Pass 1: (6 comparisons)

- 64,34 → 34,64,25,12,22,11,90
- 64,25 → 34,25,64,12,22,11,90
- 64,12 → 34,25,12,64,22,11,90

- 64,22 → 34,25,12,22,64,11,90
- 64,11 → 34,25,12,22,11,64,90
- 64,90 → 34,25,12,22,11,64,90 (no swap)

Pass 2: (5 comparisons)

- Result: 25,12,22,11,34,64,90

Continue until no swaps needed...

Final Result: [11, 12, 22, 25, 34, 64, 90]

Optimization: Add flag to detect if array is already sorted and terminate early.

Time Complexity: $O(n^2)$ worst and average case, $O(n)$ best case with optimization.

12. Write an algorithm for selection sort with example.

Selection Sort Algorithm:

```
SELECTION_SORT(ARR, N)
STEP 1: FOR I = 0 TO N-2 DO
STEP 2:   SET MIN_INDEX = I
STEP 3:   FOR J = I+1 TO N-1 DO
STEP 4:     IF ARR[J] < ARR[MIN_INDEX] THEN
STEP 5:       SET MIN_INDEX = J
STEP 6:   END FOR
STEP 7:   SWAP ARR[I] AND ARR[MIN_INDEX]
STEP 8: END FOR
```

Working Principle: Selection sort divides array into sorted and unsorted portions. In each iteration, it selects the minimum element from unsorted portion and places it at the beginning of unsorted portion.

Example: Sort array [64, 25, 12, 22, 11]

Pass 1: Find minimum in [64, 25, 12, 22, 11]

- Minimum = 11 (index 4)
- Swap: [11, 25, 12, 22, 64]

Pass 2: Find minimum in [25, 12, 22, 64]

- Minimum = 12 (index 2)
- Swap: [11, 12, 25, 22, 64]

Pass 3: Find minimum in [25, 22, 64]

- Minimum = 22 (index 3)
- Swap: [11, 12, 22, 25, 64]

Pass 4: Find minimum in [25, 64]

- Minimum = 25 (already in position)
- No swap needed: [11, 12, 22, 25, 64]

Final Result: [11, 12, 22, 25, 64]

Characteristics:

- Time Complexity: O(n²) in all cases
- Space Complexity: O(1)
- Unstable sorting algorithm
- Performs fewer swaps than bubble sort

15. Write an algorithm for insertion sort with example.

Insertion Sort Algorithm:

```
INSERTION_SORT(ARR, N)
STEP 1: FOR I = 1 TO N-1 DO
STEP 2:   SET KEY = ARR[I]
STEP 3:   SET J = I - 1
STEP 4:   WHILE J ≥ 0 AND ARR[J] > KEY DO
STEP 5:     SET ARR[J+1] = ARR[J]
STEP 6:     SET J = J - 1
STEP 7:   END WHILE
STEP 8:   SET ARR[J+1] = KEY
STEP 9: END FOR
```

Working Principle: Insertion sort builds the sorted array one element at a time by repeatedly taking an element from unsorted portion and inserting it at correct position in sorted portion.

Example: Sort array [12, 11, 13, 5, 6]

Initial: [12, 11, 13, 5, 6]

Pass 1: KEY = 11, insert in [12]

- Compare 11 with 12: 11 < 12, shift 12 right
- Insert 11: [11, 12, 13, 5, 6]

Pass 2: KEY = 13, insert in [11, 12]

- Compare 13 with 12: 13 > 12, no shift needed
- Result: [11, 12, 13, 5, 6]

Pass 3: KEY = 5, insert in [11, 12, 13]

- Shift 13, 12, 11 right and insert 5
- Result: [5, 11, 12, 13, 6]

Pass 4: KEY = 6, insert in [5, 11, 12, 13]

- Shift 13, 12, 11 right and insert 6 after 5
- Result: [5, 6, 11, 12, 13]

Characteristics:

- Time Complexity: $O(n^2)$ worst case, $O(n)$ best case
- Space Complexity: O(1)
- Stable and adaptive sorting algorithm

18. Write an algorithm for merging two sorted array.

Algorithm for Merging Two Sorted Arrays:

```
MERGE_SORTED_ARRAYS(A, M, B, N, C)
STEP 1: SET I = 0, J = 0, K = 0
STEP 2: WHILE I < M AND J < N DO
STEP 3:   IF A[I] ≤ B[J] THEN
STEP 4:     SET C[K] = A[I]
STEP 5:     SET I = I + 1
STEP 6:   ELSE
STEP 7:     SET C[K] = B[J]
STEP 8:     SET J = J + 1
STEP 9:   SET K = K + 1
STEP 10: END WHILE
STEP 11: WHILE I < M DO
STEP 12:   SET C[K] = A[I]
STEP 13:   SET I = I + 1, K = K + 1
STEP 14: END WHILE
STEP 15: WHILE J < N DO
STEP 16:   SET C[K] = B[J]
STEP 17:   SET J = J + 1, K = K + 1
STEP 18: END WHILE
```

Working Principle: Compare elements from both arrays and place smaller element in result array. Continue until one array is exhausted, then copy remaining elements from other array.

Example: Array A: [1, 5, 9, 12] Array B: [2, 6, 8, 11, 15]

Merging Process:

- Compare 1,2: 1 smaller → C=[1]
- Compare 5,2: 2 smaller → C=[1,2]
- Compare 5,6: 5 smaller → C=[1,2,5]
- Compare 9,6: 6 smaller → C=[1,2,5,6]
- Continue process...

Final Result: C=[1,2,5,6,8,9,11,12,15]

Time Complexity: O(M+N) **Space Complexity:** O(M+N)

20. Write an algorithm for merge sort.

Merge Sort Algorithm:

```
MERGE_SORT(ARR, LEFT, RIGHT)
STEP 1: IF LEFT < RIGHT THEN
STEP 2:   SET MID = (LEFT + RIGHT) / 2
STEP 3:   CALL MERGE_SORT(ARR, LEFT, MID)
STEP 4:   CALL MERGE_SORT(ARR, MID+1, RIGHT)
STEP 5:   CALL MERGE(ARR, LEFT, MID, RIGHT)
STEP 6: END IF

MERGE(ARR, LEFT, MID, RIGHT)
STEP 1: SET I = LEFT, J = MID+1, K = LEFT
STEP 2: CREATE TEMP[RIGHT-LEFT+1]
STEP 3: WHILE I ≤ MID AND J ≤ RIGHT DO
STEP 4:   IF ARR[I] ≤ ARR[J] THEN
STEP 5:     SET TEMP[K] = ARR[I], I = I+1
STEP 6:   ELSE
STEP 7:     SET TEMP[K] = ARR[J], J = J+1
STEP 8:   SET K = K+1
STEP 9: END WHILE
STEP 10: COPY REMAINING ELEMENTS TO TEMP
STEP 11: COPY TEMP BACK TO ARR
```

Working Principle: Merge sort follows divide-and-conquer strategy. It recursively divides array into two halves until single elements remain, then merges them back in sorted order.

Example: [38, 27, 43, 3, 9, 82, 10]

Divide Phase:

```
[38,27,43,3,9,82,10]
    /        \
[38,27,43]  [3,9,82,10]
   /    \      /       \
[38,27] [43] [3,9]  [82,10]
  /  \    |   /  \    /   \
[38][27] [43][3][9][82][10]
```

Merge Phase:

- Merge [38],[27] → [27,38]
- Merge [27,38],[43] → [27,38,43]
- Continue merging...

Final Result: [3, 9, 10, 27, 38, 43, 82]

Characteristics:

- Time Complexity: O(n log n) in all cases
- Space Complexity: O(n)
- Stable sorting algorithm
- Not in-place

23. Write an algorithm for quick sort.

Quick Sort Algorithm:

```
QUICK_SORT(ARR, LOW, HIGH)
STEP 1: IF LOW < HIGH THEN
STEP 2:   PI = PARTITION(ARR, LOW, HIGH)
STEP 3:   CALL QUICK_SORT(ARR, LOW, PI-1)
STEP 4:   CALL QUICK_SORT(ARR, PI+1, HIGH)
STEP 5: END IF

PARTITION(ARR, LOW, HIGH)
STEP 1: SET PIVOT = ARR[HIGH]
STEP 2: SET I = LOW - 1
STEP 3: FOR J = LOW TO HIGH-1 DO
STEP 4:   IF ARR[J] < PIVOT THEN
STEP 5:     SET I = I + 1
```