**DAA Lab Manual**

**PART-A**

**1. Design and Implement an algorithm for computing Greatest Common divisor (GCD) of 2 numbers, say m and n, using the following approaches. i. ii. iii. Middle school procedure Euclid's Algorithm by Recursion Consecutive Integer Checking Method**

**Compute the Time Complexity for each and Display the GCD (m, n) where m>n.**

**1. Middle School Procedure**

**Algorithm**

1. Find all divisors of **m** and **n**.
2. Identify the common divisors of **m** and **n**.
3. The largest common divisor is the **GCD**.

**Algorithm Steps**

Step 1: Input two numbers m and n (where m > n).

Step 2: Initialize an empty list for factors of m and n.

Step 3: Find all divisors of m and store them in a list.

Step 4: Find all divisors of n and store them in a list.

Step 5: Find the common divisors from both lists.

Step 6: Return the maximum value from the common divisors list as GCD.

Step 7: End.

**Time Complexity: O(min(m,n))**

**2. Euclidean Algorithm (Recursive)**

**Algorithm**

1. If **n = 0**, return **m** as the **GCD**.
2. Otherwise, compute GCD(n, m % n).
3. Continue until **n** becomes **0**.

**Algorithm Steps**

Step 1: Input two numbers m and n (where m > n).

Step 2: If n == 0, return m as GCD.

Step 3: Else, compute GCD(n, m % n).

Step 4: Repeat until n becomes 0.

Step 5: Return the final value of m as GCD.

Step 6: End.

**Time Complexity: O(log(min(m,n))) (Most efficient)**

**3. Consecutive Integer Checking Method**

**Algorithm**

1. Find the smaller number between **m** and **n**.
2. Start from that number and decrement by **1**.
3. Check if the number divides both **m** and **n**.
4. The first number that divides both is the **GCD**.

**Algorithm Steps**

Step 1: Input two numbers m and n (where m > n).

Step 2: Set gcd = min(m, n).

Step 3: While gcd > 0:

- If m % gcd == 0 and n % gcd == 0, return gcd.

- Else, decrement gcd by 1.

Step 4: Return 1 if no common divisor is found.

Step 5: End.

**Time Complexity: O(min(m,n)) (Slowest in worst cases)**

**Conclusion**

* **Use Euclidean Algorithm** for the fastest computation.
* **Middle School Procedure** is simple but slower.
* **Consecutive Integer Method** is the slowest.

**Note:**

 **Middle School Procedure**

* **Best for**: Teaching basic concepts in GCD computation.
* **Complexity**: O(min(m,n)), but slower due to listing factors.
* **Good for**: Small numbers and educational purposes.

 **Euclidean Algorithm (Recursive)**

* **Best for**: Fast computation, even for large numbers.
* **Complexity**: O(log(min(m,n))) (most efficient).
* **Good for**: Practical applications in computing.

 **Consecutive Integer Checking Method**

* **Best for**: Understanding how GCD works step-by-step.
* **Complexity:**(min(m, n)) (slowest in worst cases).
* **Good for**: Learning purposes, not recommended for large numbers.

**Program:c**

**Output:**

Middle School GCD: 8

Time Taken: 0.000041 sec

Euclidean GCD: 8

Time Taken: 0.000003 sec

Consecutive Integer GCD: 8

Time Taken: 0.000007 sec

### ****Analysis of Execution Time:****

| **Method** | **GCD Output** | **Time Taken (sec)** | **Efficiency** |
| --- | --- | --- | --- |
| **Middle School Procedure** | 8 | 0.000041 | **Slowest** (O(min(m, n))) |
| **Euclidean Algorithm** | 8 | 0.000003 | **Fastest** (O(log(min(m, n)))) |
| **Consecutive Integer Check** | 8 | 0.000007 | **Inefficient** (O(min(m, n))) |

**2.Design and Implement algorithm for searching techniques Linear Search and Binary Search(iterative/recursive). Compute the Time Complexity and Display.**

**1. Linear Search**

**Algorithm**

1. Start from the first element of the array.
2. Compare each element with the target value.
3. If found, return the index.
4. If not found, return **-1**.

**Time Complexity**

* **Best Case**: O(1) (if the element is at the first position).
* **Worst Case**: O(n) (if the element is at the last position or not present).
* **Average Case**: O(n).

**2. Binary Search (Iterative & Recursive)**

**Algorithm (Binary Search)**

1. Sort the array (if not sorted).
2. Find the middle element.
3. If the middle element is the target, return the index.
4. If the target is smaller, search in the left half.
5. If the target is larger, search in the right half.
6. Repeat until the target is found or the search range is empty.

**Time Complexity**

* **Best Case**: O(1) (if the middle element is the target).
* **Worst Case**: O(logn).
* **Average Case**: O(logn).

**Implementation in C**

**Output:**

Linear Search Index: 3

Binary Search (Iterative) Index: 3

Binary Search (Recursive) Index: 3

Yes! The correct index for the searched element (assuming **zero-based indexing**) is **3** in all three methods. This confirms that:

✅ **Linear Search** correctly finds the index.  
✅ **Binary Search (Iterative)** also finds the correct index.  
✅ **Binary Search (Recursive)** works as expected.

**Time Complexity Recap:**

| **Search Algorithm** | **Best Case** | **Worst Case** |
| --- | --- | --- |
| **Linear Search** | O(1) | O(n) |
| **Binary Search (Iterative)** | O(1) | O(logn) |
| **Binary Search (Recursive)** | O(1) | O(logn) |

**Key Observations:**

* **Binary Search (Iterative & Recursive) is much faster** than Linear Search for large datasets.
* **Linear Search works on unsorted arrays**, while **Binary Search requires a sorted array**.

**3. Consider the problem: You have a row of binary digits arranged randomly. Arrange them in such an order that all 0's precede all l's or vice-versa. The only constraint in arranging them is that you are allowed to interchange the positions of binary digits if they are not similar.**

**" Implement an algorithm for Merge-sort for binary value as input like 110101000.**

**"Compute the Time Complexity and Display the output as 0 0 0001111.**

**Merge Sort Steps:**

1. **Divide** the binary sequence into two halves recursively.
2. **Sort** each half separately.
3. **Merge** the two halves while ensuring all 0s appear before 1s.

Since **Merge Sort** is a stable sorting algorithm, it maintains the relative order of elements (important if we extend this to other cases).

**Time Complexity:**

* **Merge Sort runs in** O(nlogn)**time complexity**.
* Since it sorts recursively and merges efficiently, it works well for large input sizes.

**Output:**

Sorted Binary Array: 0 0 0 0 0 1 1 1 1

=== Code Execution Successful ===

1. **Dsesign and Implement a Quick Sort algorithm to sort a given set of elements and determine the time required to sort the elements. Repeat the experiment for different values of n, the number of elements in the list to be sorted. The elements can be read from the user or can be generated using the random number generator.**

## ****Quick Sort Algorithm****

### ****Algorithm****

1. **Choose a pivot** (last element in this case).
2. **Partition** the array such that:
   * Elements **smaller** than the pivot go to the left.
   * Elements **greater** than the pivot go to the right.
3. **Recursively apply Quick Sort** to both partitions.
4. **Base Case**: If the sub-array has one or zero elements, it is already sorted.

**Time Complexity:**

* **Best & Average Case**: O(nlogn)
* **Worst Case** (when the pivot is always the smallest or largest): O(n2)

## ****Implementation in C****

**Output:**

Enter number of elements: 6

Generated array: 69 2 76 85 46 72

Sorted array: 2 46 69 72 76 85

Time taken: 0.000001 sec

=== Code Execution Successful ===

**5.Design and Implement an algorithm to print all the nodes reachable from a given starting node in a graph using Breadth First Search (BFS). Use Queue for constructing BFS spanning tree. Display the BFS traversal order**

Here’s an implementation of **Breadth-First Search (BFS)** for graph traversal using a **Queue** to construct the BFS spanning tree. The program will:

1. **Accept user input for the graph (adjacency list).**
2. **Perform BFS traversal starting from a given node.**
3. **Display all reachable nodes in BFS order.**
4. **Use a queue to construct the BFS spanning tree.**
5. **Compute time complexity.**

## ****1️ BFS Algorithm****

### ****Algorithm for BFS****

1. **Initialize a queue** and enqueue the starting node.
2. **Mark the starting node as visited.**
3. While the queue is not empty:
   * Dequeue a node.
   * Print the node (BFS order).
   * Enqueue all its **unvisited** neighbors and mark them **visited**.
4. Repeat until all reachable nodes are visited.

## ****BFS Implementation in C****

**Enter number of nodes: 3**

**Enter adjacency matrix:**

**3 4**

**5 6**

**6 7**

**7 8**

**3 5**

**Enter start node: BFS Traversal: 5**

**=== Code Execution Successful ===3**

**6.** **Design and Implement an algorithm to check whether a given graph is connected or not using Depth First Search (DFS). Use stack for constructing DFS spanning tree traversal.**

**Display the DFS traversal order.**

**Algorithm: DFS (Recursive)**

1. **Start from a source node** (or any unvisited node in a disconnected graph).
2. **Mark the current node as visited**.
3. **Recursively visit all unvisited adjacent nodes**.
4. **Backtrack when no unvisited adjacent nodes remain**.

**Output:**

**Enter the number of nodes: 4**

**Enter the adjacency matrix:**

**0 1 1 0**

**1 0 0 1**

**1 0 0 1**

**0 1 1 0**

**Enter the starting node: 0**

**DFS Traversal: 0 2 3 1**

**The graph is Connected.**

**=== Code Execution Successful ===**

1. **Design and Implement Topological sort algorithnm for a directed graph (DAG) using anyone of the following approaches.**

**i)DFS-based**

**ii)Source-removal**

## ****1. DFS-Based Topological Sort****

### ****Approach****

* Use **DFS** to traverse the graph.
* Maintain a **stack** to store nodes in reverse finishing order.
* When a node has **no outgoing edges left**, push it to the stack.
* Pop from the stack to get the **topological order**.

### ****C Implementation (DFS-based)****

**Output:**

**Enter the number of nodes: 6**

**Enter the adjacency matrix (only for DAG):**

**0 1 1 0 0 0**

**0 0 0 1 0 0**

**0 0 0 1 1 0**

**0 0 0 0 0 1**

**0 0 0 0 0 1**

**0 0 0 0 0 0**

**Topological Sort Order (DFS-based): 0 2 4 1 3 5**

**=== Code Execution Successful ===**

## ****2. Source Removal (Kahn’s Algorithm - BFS-based)****

### ****Approach****

* Compute **in-degree** (number of incoming edges) for each node.
* Add **nodes with in-degree = 0** to a queue (they have no dependencies).
* Remove nodes from the queue, process them, and reduce the in-degree of their neighbors.
* If a neighbor’s in-degree becomes **0**, add it to the queue.
* Continue until all nodes are processed.

### ****C Implementation (Source Removal - Kahn’s Algorithm)****

Output

Enter the number of nodes: 6

Enter the adjacency matrix (only for DAG):

0 1 1 0 0 0

0 0 0 1 0 0

0 0 0 1 1 0

0 0 0 0 0 1

0 0 0 0 0 1

0 0 0 0 0 0

Topological Sort Order (Kahn's Algorithm): 0 1 2 3 4 5