****

**Lab no: 1 Date: 2081/12/20**

**Title: Write a program to demonstrate the Greatest Common Divisor (GCD) using the Iterative Approach and Recursive Approach.**

**GCD:**

The **Greatest Common Divisor (GCD)** of two numbers is the largest positive integer that divides both numbers without leaving a remainder. It can be found using:

1. **Iterative Approach** (Loop-based)
2. **Recursive Approach** (Function calls itself)

**Algorithm for GCD**

**Iterative Approach using Euclidean Algorithm**

1. Start with two numbers **a** and **b**.
2. Repeat until **b** becomes **0**:
   * Store the value of **b** in a temporary variable.
   * Set **b = a mod b** (remainder of division).
   * Set **a = temp**.
3. When **b = 0**, the remaining value of **a** is the **GCD**.
4. Return **a** as the final GCD.

**Pseudo Code:**

Function GCD\_

Return a Iterative(a, b):

While b ≠ 0:

temp = b

b = a mod b

a = temp

**Recursive Approach**

1. If **b == 0**, return **a** as the GCD.
2. Else, recursively call GCD\_Recursive(b, a mod b).
3. When base case **b == 0** is reached, return the last non-zero value of **a**.

**Pseudo Code:**

Function GCD\_Iterative(a, b):

While b ≠ 0:

temp = b

b = a mod b

a = temp

Return a

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 2 Date: 2081/12/20**

**Title: Write a program to demonstrate Fibonacci Series using the Iterative Approach**

**Fibonacci Series:**

The **Fibonacci Series** is a sequence where each number is the sum of the two preceding ones, starting from **0 and 1**.

**Formula:**

F0=0 and F1=1:

**Fn = Fn-1+Fn-2  n>1**

**Example of Fibonacci Sequence:**

0,1,1,2,3,5,8,13,21,34,55…

## **Algorithm for Fibonacci Series (Iterative Approach)**

1. Start with **n**, the number of terms.
2. Initialize two variables:
   * first = 0 (first Fibonacci number)
   * second = 1 (second Fibonacci number)
3. Print first and second as the first two terms.
4. Use a loop to generate the next terms:
   * Compute next = first + second
   * Update first = second
   * Update second = next
   * Repeat for (n-2) terms.
5. Print the generated Fibonacci series.

**Pseudo-code**

Function Fibonacci\_Iterative(n):

first = 0

second = 1

Print first, second

For i = 2 to n-1:

next = first + second

Print next

first = second

second = next

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 3 Date: 2081/12/20**

**Title: Write a program to demonstrate Linear Search.**

**Linear Search:**

**Linear Search** is a simple searching algorithm used to find the position of a target element in an **unsorted list or array**. It checks each element one by one until the desired element is found or the end of the list is reached.

### ****Characteristics of Linear Search:****

* Works on both sorted and unsorted data.
* Time Complexity: **O(n)** (worst case, when the element is at the end or not present).
* Space Complexity: **O(1)** (no extra memory required).
* Best suited for **small datasets**.

## **Algorithm for Linear Search**

### ****Steps:****

1. Start from the first element of the array.
2. Compare each element with the target value.
3. If a match is found, return the index of that element.
4. If no match is found, return -1 (element not present).

## **Pseudo-code for Linear Search**

Function LinearSearch(arr, n, key):

For i = 0 to n-1:

If arr[i] == key:

Return i // Element found at index i

Return -1 // Element not found

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 4 Date: 2081/12/20**

**Title: Write a program to demonstrate Binary Search using the Iterative Approach and Recursive Approach**

**Binary Search:**

**Binary Search** is an efficient searching algorithm used to find the position of a target element in a **sorted array**. It follows a **divide and conquer** approach by repeatedly dividing the search space in half until the element is found or the search space is exhausted.

### ****Characteristics of Binary Search:****

* **Works only on sorted data** (ascending or descending).
* Time Complexity:
  1. **O(log n)** in both **worst** and **average** cases.
  2. **O(1)** in the **best case** (when the middle element is the target).
* Space Complexity: **O(1) for Iterative**, **O(log n) for Recursive**.

## **Algorithm for Binary Search**

### ****Iterative Approach:****

1. Set low = 0 and high = n-1.
2. Repeat while low <= high:
   * Find mid = (low + high) / 2.
   * If arr[mid] == key, return mid (element found).
   * If arr[mid] > key, search in the **left half** (high = mid - 1).
   * If arr[mid] < key, search in the **right half** (low = mid + 1).
3. If low > high, return -1 (element not found).

## **Pseudo-code**

Function BinarySearch\_Iterative(arr, n, key):

low = 0

high = n - 1

While low <= high:

mid = (low + high) / 2

If arr[mid] == key:

Return mid // Element found

Else If arr[mid] > key:

high = mid - 1 // Search in the left half

Else:

low = mid + 1 // Search in the right half

Return -1 // Element not found

## **Recursive Approach**

1. If low > high, return -1 (element not found).
2. Compute mid = (low + high) / 2.
3. If arr[mid] == key, return mid.
4. If arr[mid] > key, search in the **left half** by calling the function recursively.
5. If arr[mid] < key, search in the **right half** by calling the function recursively.

**Pseudo-code**

Function BinarySearch\_Recursive(arr, low, high, key):

If low > high:

Return -1 // Element not found

mid = (low + high) / 2

If arr[mid] == key:

Return mid // Element found

Else If arr[mid] > key:

Return BinarySearch\_Recursive(arr, low, mid-1, key) // Search left half

Else:

Return BinarySearch\_Recursive(arr, mid+1, high, key) // Search right half

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 5 Date: 2081/12/20**

**Title: Write a program to implement Selection Sort.**

**Selection Sort:**

**Selection Sort** is a simple comparison-based sorting algorithm. It works by repeatedly selecting the **smallest (or largest)** element from the **unsorted** part of the array and swapping it with the first element of the unsorted portion.

### ****Characteristics of Selection Sort:****

* Works for both **small and large datasets**, but **not efficient for large inputs**.
* **Time Complexity:**
  + **O(n²)** in all cases (worst, average, and best).
* **Space Complexity:** **O(1)** (in-place sorting, no extra memory used).
* **Stable?** ❌ No (original order may not be preserved in case of duplicates).
* **Used when** the number of swaps should be minimized.

**Algorithm**

1. Start from index i = 0 and assume the **minimum element** is at i.
2. Traverse the rest of the array (i+1 to n-1) to find the smallest element.
3. Swap the smallest element with arr[i].
4. Repeat the process for i = 1 to n-2.
5. The array becomes **sorted** after n-1 iterations.

**Pseudo-code**

Function SelectionSort(arr, n):

For i = 0 to n-2:

minIndex = i

For j = i+1 to n-1:

If arr[j] < arr[minIndex]:

minIndex = j // Update minIndex

Swap arr[i] and arr[minIndex] *// Place the smallest element in correct position*

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 6 Date: 2081/12/20**

**Title: Write a program to implement Insertion Sort.**

**Insertion Sort:**

Insertion sort is a simple sorting algorithm that builds the final sorted list one element at a time. It works by iteratively taking an element from the unsorted part of the list and inserting it into its correct position within the sorted part of the list.

## **Algorithm for Insertion Sort**

1. Assume the **first element** is already sorted.
2. Pick the next element and compare it with the elements in the sorted portion.
3. Shift elements to the right until the correct position is found.
4. Insert the selected element into its correct position.
5. Repeat for all elements until the array is sorted.

**Pseudo-code for Insertion Sort**

Function InsertionSort(arr, n):

For i = 1 to n-1:

key = arr[i] *// Element to be inserted*

j = i - 1

While j >= 0 and arr[j] > key:

arr[j+1] = arr[j] *// Shift elements right*

j = j - 1

arr[j+1] = key *// Insert key at correct position*

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 7 Date: 2081/12/20**

**Title: Write a program to implement Min-Max Sort.**

**Min-Max Sort:**

**Min-Max Sort** is a **variant of Selection Sort** where in each pass, the **minimum** and **maximum** elements are selected **simultaneously** and placed at their correct positions. This reduces the number of passes compared to traditional Selection Sort.

**Characteristics of Min-Max Sort:**

* **Time Complexity:** **O(n²)** (same as Selection Sort).
* **Space Complexity:** **O(1)** (in-place sorting, no extra memory used).
* **Stable?** ❌ No (swaps may disrupt the order of duplicate elements).
* **Improvement over Selection Sort** as it places both minimum and maximum in one pass, reducing the number of iterations by half.

**Algorithm**

1. Start with two pointers: left (beginning of the array) and right (end of the array).
2. Find the **minimum** and **maximum** elements in the unsorted portion of the array.
3. Swap the **minimum** element with the first unsorted element (left).
4. Swap the **maximum** element with the last unsorted element (right).
5. Move left forward and right backward.
6. Repeat until all elements are sorted

**Pseudo-code**

Function MinMaxSort(arr, n):

left = 0

right = n - 1

While left < right:

minIndex = left

maxIndex = left

For i = left to right: *// Find min and max in unsorted part*

If arr[i] < arr[minIndex]:

minIndex = i

If arr[i] > arr[maxIndex]:

maxIndex = I *// Swap min with leftmost unsorted element*

Swap arr[left] and arr[minIndex]

*// If max element was swapped, update its position*

If maxIndex == left:

maxIndex = minIndex

*// Swap max with rightmost unsorted element*

Swap arr[right] and arr[maxIndex]

// Move towards center

left = left + 1

right = right – 1

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 8 Date: 2081/12/20**

**Title: Write a program to implement Merge Sort (Divide & Conquer approach).**

**Merge Sort:**

**Merge Sort** is a **Divide & Conquer** sorting algorithm that splits the array into smaller subarrays, sorts them, and then merges them back together in a sorted manner. It is **efficient** and works well for large datasets.

**Algorithm**

1. If the array has **one or zero elements**, return (already sorted).
2. Divide the array into two halves (**left** and **right**).
3. Recursively apply **Merge Sort** on both halves.
4. Merge the two sorted halves into a **single sorted array**.

**Pseudo-Code**

Function MergeSort(arr, left, right):

If left < right:

mid = (left + right) / 2 // Find middle index

MergeSort(arr, left, mid) // Recursively sort left half

MergeSort(arr, mid+1, right) // Recursively sort right half

Merge(arr, left, mid, right) // Merge the sorted halves

Function Merge(arr, left, mid, right):

Create temporary arrays Left[] and Right[]

Copy elements into Left[] (arr[left to mid]) and Right[] (arr[mid+1 to right])

i = 0, j = 0, k = left // i for Left[], j for Right[], k for arr[]

While i < size(Left[]) and j < size(Right[]):

If Left[i] <= Right[j]:

arr[k] = Left[i]

i = i + 1

Else:

arr[k] = Right[j]

j = j + 1

k = k + 1

While i < size(Left[]):

arr[k] = Left[i]

i = i + 1

k = k + 1

While j < size(Right[]):

arr[k] = Right[j]

j = j + 1

k = k + 1

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 9 Date: 2081/12/20**

**Title: Write a program to implement Quick Sort (Divide & Conquer approach).**

**Quick Sort:**

**Quick Sort** is a **Divide & Conquer** sorting algorithm that works by selecting a **pivot** element, partitioning the array around it, and then recursively sorting the subarrays. It is often the **fastest** sorting algorithm for large datasets due to its average **O(n log n)** time complexity.

**Algorithm:**

1. Choose a **pivot** element from the array. (Common strategies: first element, last element, middle element, or random element).
2. Partition the array into two subarrays:
   1. **Left subarray** contains elements smaller than the pivot.
   2. **Right subarray** contains elements greater than the pivot.
3. Recursively apply **Quick Sort** to the left and right subarrays.
4. Combine the subarrays by placing the pivot element between them.

**Pseudo-Code:**

Function QuickSort(arr, low, high):

If low < high:

// Partition the array and get the pivot index

pivotIndex = Partition(arr, low, high)

// Recursively sort the left and right subarrays

QuickSort(arr, low, pivotIndex - 1)

QuickSort(arr, pivotIndex + 1, high)

Function Partition(arr, low, high):

pivot = arr[high] // Choose last element as pivot

i = low - 1

For j = low to high - 1:

If arr[j] <= pivot:

i = i + 1

Swap arr[i] and arr[j]

Swap arr[i + 1] and arr[high] // Place pivot in its correct position

Return i + 1 // Return the pivot index

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 10 Date: 2081/12/20**

**Title: Write a program to implement Randomized Quick Sort**

**Randomized Quick Sort:**

**Randomized Quick Sort** is a **variant of Quick Sort** that selects the pivot element **randomly** rather than choosing a fixed element (such as the first, last, or middle element). This randomization reduces the likelihood of encountering worst-case performance (which happens with already sorted or nearly sorted arrays) and improves the algorithm's **average case performance**.

By randomizing the pivot selection, it helps to avoid the **O(n²)** worst-case time complexity, making the algorithm perform closer to **O(n log n)** on average.

**Algorithm:**

1. Choose a **random pivot** from the array.
2. Partition the array into two subarrays:
   1. **Left subarray**: Contains elements less than the pivot.
   2. **Right subarray**: Contains elements greater than the pivot.
3. Recursively apply **Randomized Quick Sort** on both subarrays.
4. Combine the subarrays by placing the pivot element between them.

**Pseudo-Code:**

Function RandomizedQuickSort(arr, low, high):

If low < high:

pivotIndex = RandomizedPartition(arr, low, high) // Randomly choose a pivot

RandomizedQuickSort(arr, low, pivotIndex - 1) // Recursively sort the left and right subarrays

RandomizedQuickSort(arr, pivotIndex + 1, high)

Function RandomizedPartition(arr, low, high):

// Select a random index between low and high

pivotIndex = Random(low, high)

// Swap the pivot element with the last element

Swap arr[pivotIndex] and arr[high]

// Partition the array and get the pivot index

Return Partition(arr, low, high)

Function Partition(arr, low, high):

pivot = arr[high] // Pivot is at the last index

i = low - 1

For j = low to high - 1:

If arr[j] <= pivot:

i = i + 1

Swap arr[i] and arr[j]

Swap arr[i + 1] and arr[high] // Place pivot in its correct position

Return i + 1 // Return the pivot index

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 11 Date: 2081/12/20**

**Title: Write a program to implement Heap Sort.**

**Heap Sort:**

**Heap Sort** is a comparison-based sorting algorithm that uses a **binary heap** data structure to organize the elements in an array. It works by first building a max-heap (or min-heap) and then repeatedly extracting the root element, which is either the maximum or minimum, depending on the heap structure. The heap property ensures that the largest (or smallest) element is always at the root.

**Time Complexity:**

* **Best, Worst, and Average Case:** O(n log n)
* **Space Complexity:** O(1) (in-place)

**Algorithm:**

1. **Build a Max-Heap** from the given array.

* Starting from the last non-leaf node, heapify the tree (make sure the tree follows the heap property).

1. **Extract the root (maximum)** element of the heap and swap it with the last element in the heap.
2. **Reduce the size of the heap** by one and **heapify the root** to maintain the max-heap property.
3. Repeat the extraction and heapifying process until all elements are sorted.

**Pseudo-Code**

Function HeapSort(arr):

n = length(arr)

// Build a max-heap

For i = n/2 - 1 down to 0:

Heapify(arr, n, i)

For i = n - 1 down to 1:

Swap arr[0] and arr[i]

// Heapify the root to maintain max-heap property

Heapify(arr, i, 0)

Function Heapify(arr, n, i):

largest = i

left = 2\*i + 1

right = 2\*i + 2

If left < n and arr[left] > arr[largest]:

largest = left

If right < n and arr[right] > arr[largest]:

largest = right

If largest != i:

Swap arr[i] and arr[largest]

// Recursively heapify the affected subtree

Heapify(arr, n, largest)

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 12 Date: 2081/12/20**

**Title: Write a program to solve the Fractional Knapsack Problem using the Greedy Approach.**

**Knapsack Problem:**

The **Fractional Knapsack Problem** is a variant of the 0/1 Knapsack Problem, where we can take fractions of an item (i.e., we are allowed to take parts of the items). In the **Greedy Approach**, the solution is to always choose the item with the highest **value per weight** ratio until the knapsack is full.

**Time Complexity:**

* **Best, Worst, and Average Case:** O(n log n) (due to sorting the items based on value per weight)
* **Space Complexity:** O(n) (for storing item information)

**Algorithm:**

1. **Calculate the value per weight** ratio for each item.
2. **Sort the items** in decreasing order based on the value per weight ratio.
3. **Iterate over the sorted items:**
4. If the item can be fully included in the knapsack, add it entirely.
5. If the item cannot be fully included, take the fractional part of it that can fit.
6. Continue until the knapsack reaches its capacity.

**Pseudo-Code:**

Function FractionalKnapsack(capacity, items):

n = length(items)

// Sort items based on value per weight in decreasing order

Sort items by value/weight ratio

totalValue = 0

For i = 0 to n - 1:

If capacity is 0:

Break

// Take the item fully if possible

If items[i].weight <= capacity:

totalValue += items[i].value

capacity -= items[i].weight

Else:

// Take fractional part of the item

totalValue += items[i].value \* (capacity / items[i].weight)

capacity = 0

Return totalValue

**Compiler: Dev C++**

**Language: C**

****

**Lab no: 13 Date: 2081/12/20**

**Title: Write a program to solve the Job Sequencing with Deadline Problem using the Greedy Approach**

**Job-Sequencing with Deadline**

The **Job Sequencing with Deadline Problem** is a classic optimization problem where we need to schedule jobs such that each job has a **deadline** and a **profit**. The goal is to schedule jobs in a way that maximizes the total profit. In this problem, only one job can be scheduled at a time, and the job must be completed before its deadline.

We use a **Greedy Approach** to solve this problem, where the jobs are selected based on **maximum profit** and scheduled as close to their deadlines as possible.

**Time Complexity:**

* **Best, Worst, and Average Case:** O(n log n) (due to sorting the jobs based on profit)
* **Space Complexity:** O(n) (for storing job information)

**Algorithm:**

1. **Sort the jobs** in decreasing order based on profit.
2. **Initialize a result array** to keep track of the jobs that are selected and their scheduled time slots.
3. **Iterate over the sorted jobs:**
   1. Find the latest available time slot for the current job.
   2. If a slot is found, schedule the job in that slot.
   3. Continue this until all jobs are considered or the slots are filled.
4. **Return the sequence of jobs and the total profit**

**Pseudo-Code:**

Function JobSequencing(jobs, n):

Sort jobs by profit in decreasing order

result = new array of size n (to store job sequence)

slot = new array of size n (to track available slots)

totalProfit = 0

// Iterate over each job

For i = 0 to n - 1:

// Find the latest available slot for the current job

For j = min(n, jobs[i].deadline) - 1 down to 0:

If slot[j] is empty:

slot[j] = jobs[i]

result[j] = jobs[i]

totalProfit += jobs[i].profit

Break

Return result, totalProfit

**Compiler: Dev C++**

**Language: C**