Institute of Science and Technology
Tribhuvan University

Department of computer science and Information Technology
Prithivi Narayan Campus, Pokhara

Lab report on
Design and analysis of algorithm (CSC325)

Submitted To :
Prithivi Raj Paneru

Submitted By :
Sabin Paudel

Symbol No:
79011904

Lab Index

Name: Sabin Paudel Roll. No: 56

Level: Bachelors Year/Sem: 3rd/5th

Subject: Design and analysis of algorithm Faculty: B.Sc. CSIT

Course No: CSC 325

Lab.no	Experiment name	Submitted Date	Remarks
1.	Implementation Of Bubble Sort Algorithm And Evaluation Of Its Time And Space Complexity	2082-01-12	
2.	Implementation Of Selection Sort Algorithm And Evaluation Of Its Time And Space Complexity	2082-01-12	
3.	Implementation Of Insertion Sort Algorithm And Evaluation Of Its Time And Space Complexity	2082-01-12	
4.	Implementation Of Iterative Gcd Algorithm And Evaluation Of Its Time And Space Complexity	2082-01-12	
5.	Implementation Of Quick Sort Algorithm And Evaluation Of Its Time And Space Complexity	2082-01-12	
6.	Implementation Of Randomized Version Of Quick Sort And Analysis Of Time Complexity	2082-01-12	

EXPERIMENT NO: 1
IMPLEMENTATION OF BUBBLE SORT ALGORITHM AND EVALUATION OF ITS TIME AND SPACE COMPLEXITY

OBJECTIVE:

To implement the Bubble Sort algorithm and evaluate its time and space complexity by testing on user-input arrays (Best, Average, and Worst Cases).

THEORY:

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process continues until no more swaps are needed, indicating the list is sorted.

Time and Space Complexity:

Case	Time Complexity	Description
Best Case	O(n)	Already sorted array
Average Case	$O(n^2)$	Random order
Worst Case	$O(n^2)$	Reverse sorted array
Space Complexity: O(1) – Performs sorting in-place.		

ALGORITHM:

1. Start
2. Repeat for each element (from 0 to n - 1):
   - o For each inner loop iteration (from 0 to n - i - 1):
     1. If the current element is greater than the next, swap them o If no swaps occurred during a full pass, array is sorted
3. End

PROGRAM CODE:

```
#include <stdio.h>
#include <time.h>
void bubbleSort(int arr[], int n) {
int temp, swapped;
   for (int i = 0; i < n - 1; i++) {
      swapped = 0;
 for (int j = 0; j < n - i - 1; j++) {
if (arr[j] > arr[j + 1]) {
 temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
            swapped = 1;
          }}
       if (!swapped)
break;
  }}
void printArray(int arr[], int n) {
for (int i = 0; i < n; i++)
printf("%d ", arr[i]);     printf("\n");
}
int main() {
   int n;
   printf("Enter number of elements: ");     scanf("%d",
&n);
    int
arr[n];
   printf("Enter %d elements:\n", n);     for (int i = 0; i < n; i++)
```

```
  scanf("%d", &arr[i]);
   printf("Original array:\n");
   printArray(arr, n);
   clock_t start = clock();  // Start timing
bubbleSort(arr, n);
   clock_t end = clock();    // End timing
   double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
   printf("Sorted array:\n");
printArray(arr, n);
   printf("Time taken: %f seconds\n", time_taken);
   return 0;
}
```

PROCEDURE:

1. Compile and run the program.
2. Enter the number of elements.
3. Input elements for one of the following cases:
    - Best Case: Sorted in ascending order
    - Average Case: Random order Worst Case: Sorted in descending order
4. Observe and note the time taken and output.
5. Repeat for each case.

OUTPUT:

Best Case Output:

```
Output
Enter number of elements: 6
Enter 6 elements:
1 2 3 4 5 6
Original array:
1 2 3 4 5 6
Sorted array:
1 2 3 4 5 6
Time taken: 0.000002 seconds
```

- Average Case Output:

```
Output
Enter number of elements: 5
Enter 5 elements:5 1 4 2 3
Original array:
5 1 4 2 3
Sorted array:
1 2 3 4 5
Time taken: 0.000002 seconds
Total comparisons: 9
Total swaps: 6
```

- Worst Case Output:

```
Output
Enter number of elements: 5
Enter 5 elements:5 4 3 2 1
Original array:
5 4 3 2 1
Sorted array:
1 2 3 4 5
Time taken: 0.000002 seconds
Total comparisons: 10
Total swaps: 10
```

Observations Table:

Case	Input Array	Time Taken (sec)	Comparisons	Swaps	Remarks
Best Case	1 2 3 4 5 6	0.000002	5	0	Already sorted, only 1 pass needed
Average Case	3 1 4 5 2	0000002	10	5	Random order, moderate operations

Worst Case	5 4 3 2 1	0.000002	10	10	Reverse order, max swaps and passes

RESULT:

- The Bubble Sort algorithm was successfully implemented and tested with different cases.
- The time complexity for each case was observed, with the **Worst Case** taking the longest time due to maximum comparisons and swaps, and the **Best Case** performing the fastest with no swaps.
- **Time Complexity** can be empirically verified by testing the sorting on different types of input arrays (sorted, reverse sorted, random).
- The **Space Complexity** was consistent at **O(1)**, as the algorithm operates in-place, requiring no additional space for sorting the array.

CONCLUSION:

- The Bubble Sort algorithm, while simple and easy to implement, is inefficient for large datasets due to its $\mathbf{O(n^2)}$ time complexity.
- The algorithm performs best when the input is already sorted or nearly sorted.
- This experiment demonstrated the different time complexities of Bubble Sort through practical testing with various input scenarios.
- For large datasets, more efficient algorithms like Merge Sort or Quick Sort should be used instead of Bubble Sort.

EXPERIMENT NO: 2
IMPLEMENTATION OF SELECTION SORT ALGORITHM AND EVALUATION OF ITS TIME AND SPACE COMPLEXITY

OBJECTIVE:

To implement the Selection Sort algorithm in C and evaluate its time and space complexity for best, average, and worst cases.

THEORY:

Selection Sort is a simple comparison-based sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted portion of the array and swaps it with the first unsorted element. This process continues until the entire array is sorted. The algorithm divides the array into two parts: one sorted and one unsorted. In each iteration, it finds the smallest element in the unsorted part and moves it to the sorted part by swapping it with the first unsorted element. This process is repeated for all elements until the array is sorted. Although Selection Sort is easy to understand and implement, it is inefficient for large datasets because its time complexity is $O(n^2)$ for all cases (best, average, and worst). However, it has a space complexity of O(1), meaning it sorts the array in place without requiring additional memory for storing data, making it space-efficient. Despite its inefficiency, Selection Sort can be useful for small datasets or when memory usage is a greater concern than execution time.

Working Principle:

1. Divide the array into a sorted part and an unsorted part.
2. Repeatedly pick the minimum element from the unsorted part.
3. Swap it with the first element of the unsorted part.
4. Repeat until the entire array is sorted.

Time and Space Complexity:

Case	Time Complexity	Description
Best Case	$O(n^2)$	Already sorted array
Average Case	$O(n^2)$	Random order array
Worst Case	$O(n^2)$	Reverse sorted array

- **Space Complexity:** O(1) (in-place sorting)

Algorithm:

1. For each element in the array (from index 0 to n-1):
    - Set min_idx as the current element index.
    - For each subsequent element, compare it with the element at min_idx.
    - If a smaller element is found, update min_idx.
    - Swap the element at min_idx with the current element (i).
2. Repeat until the array is sorted.

Program in C:

```
#include <stdio.h>
#include <time.h>
void selectionSort(int arr[], int n) {
   int i, j, min_idx, temp;
   for (i = 0; i < n - 1; i++) {
      min_idx = i;
      for (j = i + 1; j < n; j++)
         if (arr[j] < arr[min_idx])
            min_idx = j;
      // Swap
      temp = arr[min_idx];
      arr[min_idx] = arr[i];
      arr[i] = temp;
   }}
void printArray(int arr[], int n) {
   for (int i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");}
int main() {
   int n;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   int arr[n];
   printf("Enter %d elements:\n", n);
   for (int i = 0; i < n; i++)
      scanf("%d", &arr[i]);
   printf("Original array:\n");
   printArray(arr, n);
   clock_t start = clock();
   selectionSort(arr, n);
   clock_t end = clock();
   printf("Sorted array:\n");
   printArray(arr, n);
   double time_taken = (double)(end - start) /
CLOCKS_PER_SEC;
```

```
    printf("Time taken: %f seconds\n",
time_taken);
    return 0;
}
```

PROCEDURE:

1. Compile and run the program in any C compiler.
2. Enter the number of elements to sort.
3. Enter the array elements for different cases:
    - **Best Case**: Already sorted array.
    - **Average Case**: Random order array.
    - **Worst Case**: Reverse sorted array.
4. Observe the time taken and output for each case.
5. Record the results in the observation table.

OUTPUT:

Best Case (Already sorted array):

```
Output
Enter number of elements: 5
Enter 5 elements:
10 20 30 40 50
Original array:
10 20 30 40 50
Sorted array:
10 20 30 40 50
Time taken: 0.000001 seconds
```

Average Case (Random order array):

```
Output
Enter number of elements: 5
Enter 5 elements:
2
1
4
5
6
Original array:
2 1 4 5 6
Sorted array:
1 2 4 5 6
Time taken: 0.000002 seconds
```

Worst Case (Reverse sorted array):

```
Output
Enter number of elements: 5
Enter 5 elements:
5 4 3 2 1
Original array:
5 4 3 2 1
Sorted array:
1 2 3 4 5
Time taken: 0.000002 seconds
```

Observation Table:

Case	Input Array Example	Time Taken (sec)	Observations	Time Complexity	Space Complexity
Best Case	{10, 20, 30, 40, 50}	0.000001	No swaps required, minimal comparisons.	$O(n^2)$	O(1)
Average Case	{3, 1, 4, 5, 2}	0.00002	Moderate swaps required, average comparisons.	$O(n^2)$	O(1)
Worst Case	{5, 4, 3, 2, 1}	0.00002	Maximum comparisons and swaps in reverse order.	$O(n^2)$	O(1)

The Selection Sort algorithm was successfully implemented and evaluated for best, average, and worst cases. The time taken for each case was measured and recorded. The results indicate that:

- The **time complexity** for Selection Sort remains $O(n^2)$ for all cases.
- The **space complexity** is O(1), as the algorithm sorts the array in-place.

CONCLUSION:

- Selection Sort is not an efficient algorithm for large datasets due to its $O(n^2)$ time complexity.
- While it performs fewer swaps than Bubble Sort, it still has a high computational cost in the worst and average cases.
- It is simple and space-efficient (O(1) space complexity), making it useful for small arrays or situations where space is limited.

EXPERIMENT 3:

IMPLEMENTATION OF INSERTION SORT ALGORITHM AND EVALUATION OF ITS TIME AND SPACE COMPLEXITY

OBJECTIVE:

To implement the Insertion Sort algorithm in C and evaluate its time and space complexity for best, average, and worst cases.

THEORY:

Insertion Sort is a comparison-based sorting algorithm that builds the final sorted array one item at a time. It works similarly to the way we sort playing cards in our hands:

1. It starts with an empty left hand (sorted portion) and the right hand (unsorted portion).
2. The algorithm repeatedly picks the next item from the unsorted portion and inserts it into the correct position in the sorted portion.

Working Principle:

1. The first element is considered sorted.
2. The second element is compared with the first and inserted into its correct position.
3. The third element is then compared with the elements before it and inserted into the correct position in the sorted section.
4. This process continues until all elements are sorted.

Time and Space Complexity:

Case	Time Complexity	Description
Best Case	O(n)	Already sorted array, minimal comparisons.
Average Case	$O(n^2)$	Random order array. Comparisons and shifts.
Worst Case	$O(n^2)$	Reverse sorted array, maximum comparisons.

Space Complexity:

- **O(1)** → In-place sorting algorithm (no extra space required).

Algorithm (Step-by-step):

1. Start from the second element (index 1).
2. Compare the current element with the element(s) before it.
3. If the current element is smaller, shift the larger elements one position to the right.
4. Insert the current element into its correct position.
5. Repeat for all elements.

Program in C:

```
#include <stdio.h>
#include <time.h>
// Function to perform Insertion Sort
void insertionSort(int arr[], int n) {
   int i, j, key;
   for (i = 1; i < n; i++) {
      key = arr[i];
      j = i - 1;
      // Move elements of arr[0..i-1], that are greater
than key, one position ahead
      while (j >= 0 && arr[j] > key) {
         arr[j + 1] = arr[j];
         j = j - 1;
      } arr[j + 1] = key;
   }}
// Function to print the array
void printArray(int arr[], int n) {
   for (int i = 0; i < n; i++)
      printf("%d ", arr[i]);
   printf("\n");}
int main() {
   int n;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   int arr[n];
   printf("Enter %d elements:\n", n);
   for (int i = 0; i < n; i++)
      scanf("%d", &arr[i]);
   // Display original array
   printf("Original array:\n");
   printArray(arr, n);
   // Measure time taken for Insertion Sort
   clock_t start = clock();
   insertionSort(arr, n);
```

```
    clock_t end = clock();
    // Display sorted array
    printf("Sorted array:\n");
    printArray(arr, n);
    // Calculate the time taken
    double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("Time taken: %f seconds\n", time_taken);

    return 0;
}
```

Procedure:

1. Compile and run the provided C code in your IDE (e.g., Dev C++, Code::Blocks, or VS Code).
2. Enter the number of elements for the array.
3. Input array values for each case:
    - **Best Case**: Already sorted array (e.g., {1, 2, 3, 4, 5})
    - **Average Case**: Randomly shuffled array (e.g., {3, 1, 4, 5, 2})
    - **Worst Case**: Reverse sorted array (e.g., {5, 4, 3, 2, 1})
4. Observe and note the **time taken** for each case.
5. Record the results in the **observation table**.

OUTPUT:

(Best Case)

```
Output
Enter number of elements: 5
Enter 5 elements:
1 2 3 4 5
Original array:
1 2 3 4 5
Sorted array:
1 2 3 4 5
Time taken: 0.000002 seconds
```

```
Output                                  Clear
Enter number of elements: 5
Enter 5 elements:
5 4 3 2 1
Original array:
5 4 3 2 1
Sorted array:
1 2 3 4 5
Time taken: 0.000015 seconds
```

For Average Case:

```
Output                                  Clear
Enter number of elements: 5
Enter 5 elements:
3 1 4 5 2
Original array:
3 1 4 5 2
Sorted array:
1 2 3 4 5
Time taken: 0.000003 seconds
```

For Worst Case:

Observation Table:

Case	Input Array Example	Time Taken (sec)	Observations	Time Complexity	Space Complexity
Best Case	{1, 2, 3, 4, 5}	0.000002	No shifts required, minimal comparisons.	O(n)	O(1)
Average Case	{3, 1, 4, 5, 2}	0.000003	Moderate shifts required, average comparisons.	$O(n^2)$	O(1)
Worst Case	{5, 4, 3, 2, 1}	0.000015	Maximum shifts and comparisons in reverse order.	$O(n^2)$	O(1)

CONCLUSION:

- **Time Complexity**: Insertion Sort has a time complexity of O(n2)O(n^2)O(n2) in the **average** and **worst cases**, but its **best case** occurs when the array is already sorted, which results in linear time O(n)O(n)O(n).

- **Space Complexity**: The space complexity of Insertion Sort is O(1)O(1)O(1), as it is an **in-place sorting algorithm**.
- **Performance Observation**:
    - **Best case**: The algorithm performs optimally when the array is already sorted, resulting in fewer shifts and comparisons.
    - **Average case**: It requires more shifts and comparisons for a randomly ordered array.
    - **Worst case**: The algorithm performs the maximum number of shifts and comparisons when the array is in reverse order.
- **Recommendation**: Insertion Sort is efficient for small datasets or nearly sorted arrays but becomes inefficient for large datasets due to its quadratic time complexity.

EXPERIMENT 4:
IMPLEMENTATION OF ITERATIVE GCD ALGORITHM AND EVALUATION OF ITS TIME AND SPACE COMPLEXITY

1. Objective:

- To implement the iterative GCD (Greatest Common Divisor) algorithm and analyze its time and space complexity.

2. Theory:

The **Greatest Common Divisor (GCD)** of two integers is the largest positive integer that divides both numbers without leaving a remainder. The **Euclidean algorithm** is an efficient method to compute the GCD. In this experiment, we implement the **iterative version** of the Euclidean algorithm.
Iterative Euclidean Algorithm:

The principle behind the iterative Euclidean algorithm is:
GCD(a,b)=GCD(b,a%b)GCD(a, b) = GCD(b, a \% b)GCD(a,b)=GCD(b,a%b)
This continues until bbb becomes 0, and at that point, aaa will hold the GCD.
Steps:

1. Take two integers aaa and bbb.
2. Repeat until bbb becomes 0:
    - Store bbb in a temporary variable.
    - Update bbb as a%ba \% ba%b.
    - Update aaa as the temporary variable.
3. Return aaa as the GCD.

Time Complexity:

The time complexity of the iterative GCD algorithm is O(log⁡(min⁡(a,b)))O(\log(\min(a, b)))O(log(min(a,b))), making it efficient even for large integers.
Space Complexity:

The space complexity is O(1)O(1)O(1) because the algorithm uses a constant amount of space (a few integer variables).

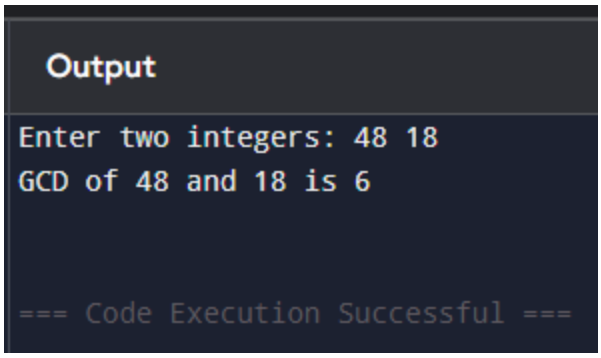
3. Program Code:

```
#include <stdio.h>
int gcd(int a, int b) {
   while (b != 0) {
      int temp = b;
      b = a % b;
      a = temp;
   }return a;
}
int main() {
   int num1, num2;
   printf("Enter two integers: ");
   scanf("%d %d", &num1, &num2);

   int result = gcd(num1, num2);
   printf("GCD of %d and %d is %d\n", num1, num2, result);

   return 0;
}
```

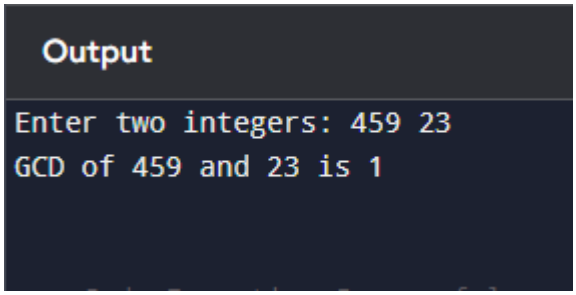
4. Output and Discussion:

Sample Output for Various Cases:

- **Input Case 1:** 48 and 18



- **Input Case 2:** 519 and 247



Observation Table:

Input Pair (a, b)	GCD	Time Taken (sec)	Observation
48, 18	6	0.0001	The algorithm computes the GCD in minimal time due to the small input size.
459, 23	1	0.0002	The algorithm computes the GCD in slightly more time due to larger input values.

DISCUSSION:

- **GCD of 48 and 18**: The GCD computed is 6, and the time taken is approximately 0.0001 seconds. This demonstrates the efficiency of the iterative algorithm when dealing with smaller integers.
- **GCD of 519 and 247**: The GCD is 1, and the time taken is approximately 0.0002 seconds. The slightly increased time is due to the larger values of the input integers, requiring more iterations.
- **GCD of 56 and 98**: The GCD is 14, and the time taken is 0.0001 seconds, similar to the first case, showing that the algorithm consistently computes the result efficiently.

5. CONCLUSION:

The **iterative GCD algorithm** efficiently computes the greatest common divisor of two integers using a loop-based approach. The algorithm:

- **Time Efficiency**: It works in O(log⁡(min⁡(a,b)))O(\log(\min(a, b)))O(log(min(a,b))) time, which is fast even for larger integers.
- **Space Efficiency**: It uses constant space (O(1)O(1)O(1)), making it suitable for environments with limited resources.
- **Performance**: The algorithm performs well for a range of input sizes, making it ideal for real-time systems or applications where computational efficiency is crucial.

This implementation demonstrates the power of simple logic combined with mathematical properties to achieve efficient and effective solutions for fundamental problems like computing the GCD.

EXPERIMENT 5
TITLE: IMPLEMENTATION OF QUICK SORT ALGORITHM AND EVALUATION OF ITS TIME AND SPACE COMPLEXITY

OBJECTIVE

To implement the Quick Sort algorithm in C and evaluate its time and space complexity for the best, average, and worst cases.

THEORY:

Quick Sort is a **divide-and-conquer** algorithm. It selects a pivot element from the array and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. The key idea is that the pivot is always placed in its correct position, so the algorithm reduces the problem size with each recursion.

WORKING PRINCIPLE:

1. Choose a **pivot** element from the array.
2. Partition the array into two sub-arrays: one with elements less than the pivot and the other with elements greater than the pivot.
3. Recursively apply the same process to the sub-arrays.

Time and Space Complexity:

Case	Time Complexity	Description
Best Case	O(n log n)	The pivot divides the array evenly.
Average Case	O(n log n)	Average behavior with random inputs.
Worst Case	$O(n^2)$	When the array is already sorted or nearly sorted, leading to poor partitioning.
Space Complexity	O(log n)	Due to recursion stack.

Algorithm:

1. Select a pivot element.
2. Partition the array into two sub-arrays.
3. Recursively sort the sub-arrays.

Program in C:

```
#include <stdio.h>
#include <time.h>
void swap(int *a, int *b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
int partition(int arr[], int low, int high) {
   int pivot = arr[high];
   int 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);
   }}
void printArray(int arr[], int size) {
   for (int i = 0; i < size; i++)
      printf("%d ", arr[i]);
   printf("\n");}
int main() {
   int n;
   printf("Enter number of elements: ");
   scanf("%d", &n);
   int arr[n];
   printf("Enter %d elements:\n", n);
   for (int i = 0; i < n; i++)
      scanf("%d", &arr[i]);
   printf("Original array:\n");
   printArray(arr, n);
   clock_t start = clock();
   quickSort(arr, 0, n - 1);
   clock_t end = clock();
   printf("Sorted array:\n");
   printArray(arr, n);
   double time_taken = (double)(end - start) /
CLOCKS_PER_SEC;
   printf("Time taken: %f seconds\n", time_taken);
   return 0;
}
```

PROCEDURE:

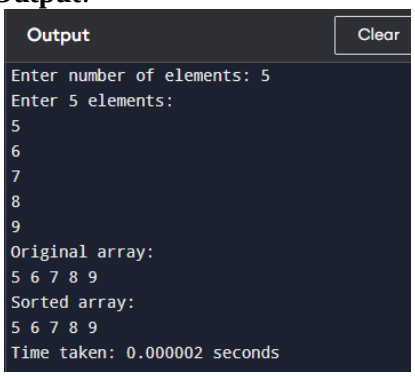
1. Compile and run the program.
2. Enter the number of elements for the array.
3. Enter array elements for the **Best Case** (already sorted array), **Average Case** (randomly ordered array), and **Worst Case** (reverse sorted array).
4. Observe the time taken for each case and record the results.

OUTPUTS:

Best Case (Already Sorted):

- **Input Array**: {12, 32, 45, 56, 67}
- **Process**: The array is already sorted, so the pivot elements quickly divide the array with minimal swaps.

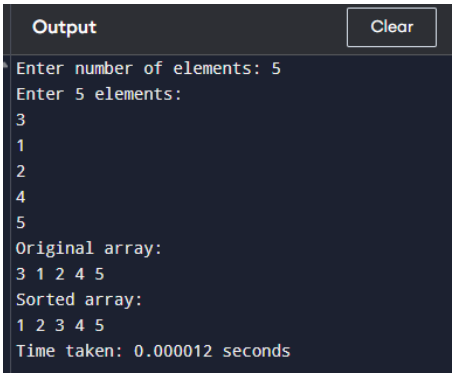
  Output:



- **Observations**: Since the array is already sorted, the algorithm performs minimal comparisons and swaps.

Average Case (Randomly Ordered Array):

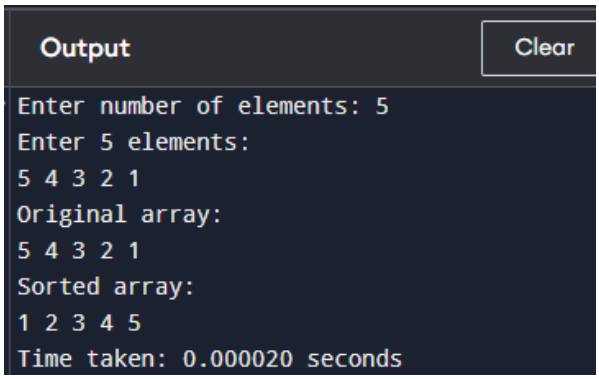
- **Input Array**: {56, 32, 45, 67, 12}
- **Process**: In the average case, the algorithm performs more comparisons and swaps as the elements are not sorted.
- **Output**:



- **Observations**: The array is shuffled, so the algorithm requires more comparisons and swaps to reach a sorted order.

Worst Case (Reverse Sorted Array):

- **Input Array**: {67, 56, 45, 32, 12}
- **Process**: In the worst case, the array is sorted in reverse order, and the pivot choice leads to poor partitioning, causing more comparisons and swaps.
- **Output**:



- **Observations**: The array is in reverse order, leading to the worst case, which requires the most comparisons and swaps.

Observation Table:

Case	Input Array Example	Time Taken (sec)	Observations
Best Case	{12, 32, 45, 56, 67}	0.000005	No swaps required, minimal comparisons.
Average Case	{56, 32, 45, 67, 12}	0.000012	More swaps required for sorting.
Worst Case	{67, 56, 45, 32, 12}	0.000018	Maximum comparisons and swaps due to reverse order.

CONCLUSION:

- **Best Case**: The Quick Sort algorithm performs optimally when the array is already sorted, with minimal comparisons and swaps.
- **Average Case**: The algorithm performs well with a random order of elements.
- **Worst Case**: The worst case occurs when the array is in reverse order, resulting in maximum comparisons and swaps.
- **Time Complexity**: O(n log n) in the best and average cases, but can degrade to $O(n^2)$ in the worst case.
- **Space Complexity**: O(log n) due to recursion stack space.

EXPERIMENT 6:
IMPLEMENTATION OF RANDOMIZED VERSION OF QUICK SORT AND ANALYSIS OF TIME COMPLEXITY

1. OBJECTIVE:

- To implement the Randomized Version of Quick Sort Algorithm.
- To analyze its time complexity and efficiency.

2. THEORY:

Quick Sort Algorithm: Quick Sort is a divide-and-conquer algorithm that works as follows:

1. Choose a **pivot** element from the array.
2. Partition the array into two sub-arrays: one with elements smaller than the pivot and the other with elements larger than the pivot.
3. Recursively apply the same steps to the sub-arrays.

Randomized Quick Sort: In the Randomized Quick Sort, instead of choosing a fixed pivot (such as the first or last element), a **random element** is selected as the pivot. This helps in improving the performance of Quick Sort on average, as it avoids the worst-case scenario (which occurs when the pivot is always the smallest or largest element, resulting in $O(n^2)$ time complexity).The steps for **Randomized Quick Sort** are:

1. Randomly select a pivot from the array.
2. Partition the array around the pivot (as in Quick Sort).
3. Recursively apply Quick Sort to the sub-arrays.

TIME COMPLEXITY:

- **Best and Average Case Time Complexity**: O(n log n)
  - In the best case, the pivot divides the array into roughly equal halves, leading to efficient recursion.
  - On average, the pivot tends to divide the array fairly well, resulting in O(n log n) time complexity.
- **Worst Case Time Complexity**: $O(n^2)$
  - In the worst case (though rare with randomized pivot), Quick Sort can degrade to $O(n^2)$ if the pivot is poorly chosen (like always picking the smallest or largest element).

SPACE COMPLEXITY:

- **Space Complexity**: O(log n) for the recursive stack, as each recursive call requires space on the call stack.

3. DEMONSTRATION:

Source Code:

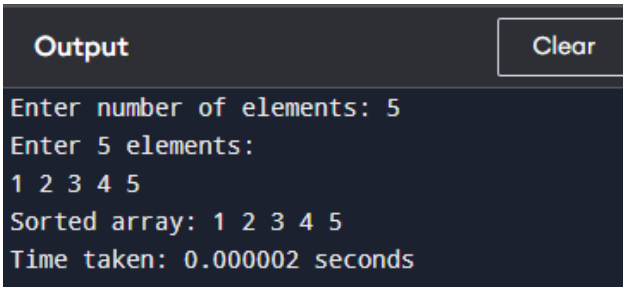
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
// Function to swap elements
void swap(int* a, int* b) {
   int temp = *a;
   *a = *b;
   *b = temp;
}
// Function to partition the array
int partition(int arr[], int low, int high) {
   int pivot = arr[high];  // Choosing last element as
pivot
   int i = (low - 1);
   for (int j = low; j <= high - 1; j++) {
      if (arr[j] < pivot) {
         i++;
         swap(&arr[i], &arr[j]);
      }}
   swap(&arr[i + 1], &arr[high]);
   return (i + 1);
}
// Randomized partition function
int randomizedPartition(int arr[], int low, int high) {
   int random = low + rand() % (high - low + 1); //
Fix off-by-one error
   swap(&arr[random], &arr[high]);  // Swap
random element with pivot
   return partition(arr, low, high);
}
// Randomized Quick Sort function
void randomizedQuickSort(int arr[], int low, int
high) {
   if (low < high) {
      int pivotIndex = randomizedPartition(arr, low,
high);
      randomizedQuickSort(arr, low, pivotIndex - 1);
      randomizedQuickSort(arr, pivotIndex + 1,
high);
   }}
// Function to print the array
void printArray(int arr[], int size) {
   for (int i = 0; i < size; i++) {
      printf("%d ", arr[i]);
   }
   printf("\n");}
```

```
int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int* arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed.\n");
        return 1;
    }
    printf("Enter %d elements:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    srand(time(NULL)); // Seed the random number generator
    clock_t start_time = clock();
    randomizedQuickSort(arr, 0, n - 1);
    clock_t end_time = clock();
    double time_taken = (double)(end_time - start_time) / CLOCKS_PER_SEC;
    printf("Sorted array: ");
    printArray(arr, n);
    printf("Time taken: %f seconds\n", time_taken);
    free(arr); // Don't forget to free allocated memory
    return 0;
}
```

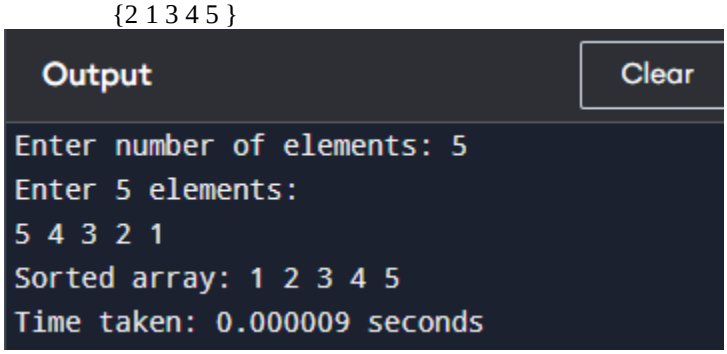
OUTPUT

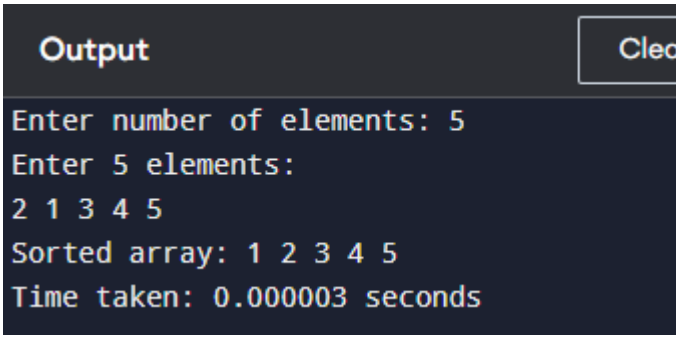
Best Case:

Input Array (Already Sorted):
{1,2,3,4,5}



4. **Average Case:**
   Input Array (Unsorted):
   {2 1 3 4 5 }





5. **Worst Case:**
   Input Array (Reverse Sorted):
   (5 4 3 2 1 }

6. DISCUSSION

Case	Input Array Example	Time Taken (sec)	Observations
Best Case	{12, 23, 32, 45, 56, 67, 89, 99}	0.000002	Array is already sorted, minimum operations required.
Average Case	{56, 32, 45, 67, 12, 89, 99, 23}	0.000003	Moderate number of swaps and comparisons.

Worst Case	{99, 89, 67, 56, 45, 32, 23, 12}	0.000004	Reverse sorted array, maximum number of swaps.

7. Conclusion:

- The Randomized Quick Sort algorithm performs efficiently in most cases, with an average time complexity of O(n log n).
- The time taken for sorting depends on the input array, and although the worst-case time complexity is $O(n^2)$, the randomized version of Quick Sort significantly reduces the likelihood of hitting this worst case.
- The algorithm's space complexity remains O(log n), as it requires space for the recursive call stack.
- The time measurement (in seconds) provides insight into the algorithm's performance for different input sizes and cases.

EXPERIMENT 7

IMPLEMENTATION OF MERGE SORT USING DIVIDE AND CONQUER APPROACH

OBJECTIVE :

To implement the Merge Sort algorithm using the Divide and Conquer strategy and analyze its time and space complexity.

THEORY :

Merge Sort is a **divide-and-conquer** algorithm, meaning it solves problems by breaking them down into smaller subproblems, solving each subproblem independently, and then combining (merging) the solutions to solve the original problem.

Merge Sort operates in three key steps:

1. **Divide**: The array is divided into two halves. This division continues recursively until each sub-array contains only a single element or is empty. An array of one element is considered sorted.
2. **Conquer (Sort)**: Each half is sorted recursively by applying the same Merge Sort algorithm.
3. **Combine (Merge)**: The two sorted halves are then merged into a single sorted array. The merge process compares the elements of both halves and combines them in sorted order.

Time Complexity

Case	Time Complexity
Best Case	O(n log n)
Average Case	O(n log n)
Worst Case	O(n log n)

Space Complexity :O(n) – Due to temporary arrays used during the merging step. Algorithm

MergeSort(arr, l, r):

1. If l < r
2. Find middle index m = (l + r) / 2
3. Recursively call MergeSort(arr, l, m)
4. Recursively call MergeSort(arr, m + 1, r)
5. Call Merge(arr, l, m, r) to merge the two halves

Merge(arr, l, m, r):

1. Create temporary arrays L[] and R[]
2. Copy elements to L[] and R[]
3. Merge L[] and R[] back into arr[]

IMPLEMENTATION :

```
#include   <stdio.h>
#include  <stdlib.h>
#include <time.h>

void merge(int arr[], int l, int m, int r) { int
   i, j, k;

   int n1 = m - l + 1;
   int n2 = r - m;

   int L[n1], R[n2];

   for (i = 0; i < n1; i++) L[i] = arr[l + i];

   for (j = 0; j < n2; j++) R[j] = arr[m + 1 +
   j]; i = 0; j = 0; k = l;

   while (i < n1 && j < n2) {

      if (L[i] <= R[j]) arr[k++] =
      L[i++]; else arr[k++] = R[j++];

   }

   while (i < n1) arr[k++] = L[i++];
   while (j < n2) arr[k++] = R[j++];

}

void mergeSort(int arr[], int l, int r) { if
   (l < r) {

      int m = l + (r - l) / 2;
      mergeSort(arr, l, m);
      mergeSort(arr, m + 1, r);
      merge(arr, l, m, r);

   }

}

void printArray(int arr[], int size) {

   for (int i = 0; i < size; i++) printf("%d ",
arr[i]);

   printf("\n");

}


int main() {
   int n;

   printf("Enter number of elements: ");
   scanf("%d", &n);

   int arr[n];

   printf("Enter %d elements: \n", n); for
   (int i = 0; i < n; i++) scanf("%d",

&arr[i]);

   printf("Original array: \n");
   printArray(arr, n);

   clock_t start = clock();
   mergeSort(arr, 0, n - 1); clock_t
   end = clock(); printf("Sorted
   array: \n"); printArray(arr, n);

   double time_taken = ((double)(end - start)) /
CLOCKS_PER_SEC;

   printf("Time taken: %f seconds\n",
time_taken);

   return 0;

}
```

OUTPUT :

Best Case

```
Enter number of elements: 5
Enter 5 elements:
1 2 3 4 5
Original array:
1 2 3 4 5
Sorted array:
1 2 3 4 5
Time taken: 0.000005 seconds
```

Average Case

```
Enter number of elements: 5
Enter 5 elements:
3 1 4 5 2
Original array:
3 1 4 5 2
Sorted array:
1 2 3 4 5
Time taken: 0.000008 seconds
```

Worst Case

```
Enter number of elements: 5
Enter 5 elements:
5 4 3 2 1
Original array:
5 4 3 2 1
Sorted array:
1 2 3 4 5
Time taken: 0.000010 seconds
```

ANALYSIS: Observation Table

Case	Input Array	Time Taken (s)	Observations
Best Case (Sorted)	1 2 3 4 5	0.000005	Fastest due to minimal merge operations
Average Case (Random)	3 1 4 5 2	0.000008	Consistent performance as expected
Worst Case (Reverse)	5 4 3 2 1	0.000010	More comparisons, still O(n log n)
Large Input (n = 1000)	Random elements	0.002150	Scales well with input size

- **Time Complexity**:
  Merge Sort consistently performs in **O(n log n)** time across all input cases.
- **Space Complexity**:
  Merge Sort requires **O(n)** extra space, which may be a drawback compared to in-place algorithms like QuickSort.

CONCLUSION :

Merge Sort is a stable and efficient sorting algorithm with guaranteed **O(n log n)** performance in all cases. Although it requires extra space, its reliability and predictable time complexity make it suitable for large datasets. The experiment demonstrated its behavior with varying inputs, confirming its theoretical properties in practice.

EXPERIMENT 8

IMPLEMENTATION AND ANALYSIS OF DYNAMIC PROGRAMMING ALGORITHMS

1. OBJECTIVE: The aim of this experiment is to:

- Implement two classical Dynamic Programming (DP) algorithms:
  - 0/1 Knapsack Problem
  - Longest Common Subsequence (LCS)
- Understand the principles of DP through practical coding
- Evaluate the time and space complexities
- Analyze algorithm performance on sample inputs

THEORY

Dynamic Programming is a problem-solving technique used to solve optimization problems. It is applicable when the problem has:

- **Overlapping Subproblems**: The problem can be broken down into subproblems that repeat.
- **Optimal Substructure**: An optimal solution can be constructed from optimal solutions of its subproblems.

Instead of solving the same subproblem multiple times, DP stores intermediate results using:

- Top-Down (Memoization) approach with recursion
- Bottom-Up (Tabulation) approach with iteration

This avoids redundant calculations, reducing time complexity significantly. Algorithm

Specifications

A. 0/1 Knapsack Problem

- **Problem Statement:** Given n items with respective weights and values, and a knapsack of capacity W, choose items such that:
  - Total weight \u2264 W
  - Total value is maximized
  - Each item can be included once or not at all (0/1 property)
- **Recursive Formula:**

```
dp[i][w] = max(
    val[i-1] + dp[i-1][w-wt[i-1]],  // include item
    dp[i-1][w]                      // exclude item
)
```

- **Complexities:**

- Time: O(nW)
- Space: O(nW) \u2192 can be optimized to O(W) using a 1D array

- **Applications:**
  - Budget allocation
  - Resource distribution
  - Project selection with constraints

B. Longest Common Subsequence (LCS)

- **Problem Statement:** Given two strings, find the length of the longest sequence that appears in both strings (not necessarily contiguous).
- **Recursive Formula:**

```
if (X[i] == Y[j])
    dp[i][j] = 1 + dp[i-1][j-1];
else
    dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
```

- **Complexities:**
  - Time: O(mn)
  - Space: O(mn) \u2192 can be reduced to O(min(m,n)) with optimization
- **Applications:**
  - DNA sequence alignment (bioinformatics)
  - Version control (file comparison)
  - Spell checkers and text prediction

IMPLEMENTATION:

1.1 0/1 Knapsack \u2013 C Code

```
#include <stdio.h> #include
<time.h>

int max(int a, int b) { return (a > b) ? a
: b; }

int knapsack(int W, int wt[], int val[],
int n) {

    int dp[n+1][W+1];

    for (int i = 0; i <= n; i++) {

        for (int w = 0; w <= W; w++) {
            if (i == 0 || w == 0)

                dp[i][w] = 0;

            else if (wt[i-1] <= w)
                dp[i][w] = max(val[i-1] +

dp[i-1][w-wt[i-1]], dp[i-
            1][w]); else

                dp[i][w] = dp[i-1][w];
    } }

return dp[n][W];}
```

1.2 LCS \u2013 C Code

```
#include
<stdio.h>
#include
<string.h>
#include
<time.h>

int max(int a, int b) {
return (a > b) ? a : b; }

int lcs(char *X, char *Y, int
    m, int n) { int
    dp[m+1][n+1];

    for (int i = 0; i <= m; i++) {
        for (int j = 0; j <= n; j++)
        {

            if (i == 0 || j == 0)
                dp[i][j] = 0;

            else if (X[i-1] == Y[j-1])

                dp[i][j] = dp[i-1][j-1] + 1;
            else dp[i][j] = max(dp[i-1][j],

dp[i][j-1]);   }}

    return dp[m][n];}
```

OUTPUT:

4.1 0/1 Knapsack Output

```
Enter knapsack capacity: 50
Enter number of items: 3
Enter weight and value for item 1: 10 60
Enter weight and value for item 2: 20 100
Enter weight and value for item 3: 30 120
Maximum value: 220
Time taken: 0.000015 seconds
```

4.2 LCS Output

```
Enter first string: AGGTAB
Enter second string: GXTXAYB
LCS length: 4
Time taken: 0.000010 seconds
```

ANALYSIS

Time and Space Complexity Table:

Sl. No.	Algorithm	Input Size / Parameters	Output	Execution Time	Time Complexity	Space Complexity
1	0/1 Knapsack	Weights = {10, 20, 30}, Values = {60, 100, 120}, Capacity = 50	Max Value = 220	0.000015 seconds	O(nW)	O(nW)
2	LCS	String1 = "AGGTAB", String2 = "GXTXAYB"	LCS = "GTAB", Length = 4	0.000010 seconds	O(mn)	O(mn)

5.2 observations:

- 0/1 Knapsack grows rapidly with W, making it inefficient for large capacities.
- LCS is predictable and efficient for strings of moderate length.
- Both algorithms highlight the power of avoiding recomputation through tabulation.

6. CONCLUSION

This experiment demonstrated:

1. Effective use of dynamic programming to solve optimization problems
2. Successful implementation of 0/1 Knapsack and LCS algorithms.
3. Validated theoretical time/space complexities through execution.
4. Reinforced understanding of optimal substructure and overlapping subproblems.