PRESIDENCY UNIVERSITY

GAIN MORE KNOWLEDGE
REACH GREATER HEIGHTS

Department of Computer Science and Engineering

Academic Year 2025 – 2026

Even Semester

Lab Manual

CSE2263 ANALYSIS OF ALGORITHMS

Prepared by,Dr. S. Aarif Ahamed-Asst. Prof-SCSE

TABLE OF CONTENTS

Lab Sheet 1: Measuring Running Time of an Algorithm

Aim:

To measure the execution time of different algorithms using the clock() function in Turbo C, and to study the effect of algorithm complexity on running time for:

- Simple loop (O(n))
- Nested loop ($O(n^2)$)
- Recursive function (Factorial)

Procedure:

- Open Turbo C and create a new C program.
- Include the required header files.
- Write the program to record the start time using clock().
- Execute the given loop or recursive function.
- Record the end time after execution.
- Calculate the execution time using CLOCKS_PER_SEC.
- Compile and run the program.
- Observe the time taken displayed on the screen.

Program 1: Simple Loop

```
#include <stdio.h>

#include <time.h>

int main() {

   int i, n = 1000000;

   clock_t start, end;

   double cpu_time;

   start = clock();

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

   end = clock();

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

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

   return 0;

}
```

Program 2: Measuring Running Time of Nested Loops ($O(n^2)$)

```
#include <stdio.h>

#include <time.h>
```

```
int main() {
    int i, j;
    int n = 2000;
    clock_t start, end;
    double cpu_time;
    start = clock();
    for(i = 0; i < n; i++) {
        for(j = 0; j < n; j++) {
            // Empty loop body
        }
    }
    end = clock();
    cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken for O(n^2) loop: %f seconds\n", cpu_time);
    return 0;
}
```

Program 3: Measuring Running Time of Recursive Function (Factorial)

```
#include <stdio.h>
#include <time.h>
long factorial(int n) {
    if(n == 0)
        return 1;
    return n * factorial(n - 1);
}
int main() {
    int n = 20;
    clock_t start, end;
    double cpu_time;
    start = clock();
    factorial(n);
```

```
    end = clock();
    cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("Time taken for recursive factorial: %f seconds\n", cpu_time);
    return 0;
}
```

Result:

The execution time of different algorithms was measured successfully using Turbo C. It was observed that the running time increases as the algorithm complexity increases from O(n) to $O(n^2)$.

Lab Sheet 2: Compare Running Time of Algorithms

Aim:

To compare the running time of two algorithms for calculating the sum of first n natural numbers using:

- Loop-based method
- Formula-based method

Procedure:

- Open Turbo C and create a new C program.
- Include the required header files such as stdio.h and time.h.
- Write a program to calculate the sum of first n natural numbers using a loop-based approach.
- Use the clock() function to record the start time before executing the loop.
- Record the end time after the loop execution and calculate the total execution time using CLOCKS_PER_SEC.
- Write another program to calculate the sum using the formula-based approach.
- Measure the execution time of the formula-based method in the same manner.
- Compile and run both programs separately.
- Observe and compare the execution time obtained for both approaches.

Program 1: Loop-Based Approach (O(n))

```
#include <stdio.h>
#include <time.h>
int main() {
    int n = 1000000;
    long sum = 0;
    clock_t start, end;
```

```
    start = clock();
    for(int i = 1; i <= n; i++)
        sum += i;
    end = clock();
    printf("Loop Method Time: %f seconds\n",
           (double)(end-start)/CLOCKS_PER_SEC);
    return 0;
}
```

Program 2: Formula-Based Approach (O(1))

```
#include <stdio.h>
#include <time.h>
int main() {
    int n = 1000000;
    long sum;
    clock_t start, end;
    start = clock();
    sum = n * (n + 1) / 2;
    end = clock();
    printf("Formula Method Time: %f seconds\n",
           (double)(end-start)/CLOCKS_PER_SEC);
    return 0;
}
```

Result:

The formula-based approach executed faster than the loop-based approach, proving that algorithms with lower time complexity perform better.

Lab Sheet 3: Implement Sorting Algorithms – Bubble Sort and Selection Sort

Aim:

To implement Bubble Sort and Selection Sort algorithms in C and to understand their working and time complexity.

Procedure:

• Open Turbo C and create a new C program.

• Include the required header file stdio.h.

• Read the number of elements and array values from the user.

• Implement the Bubble Sort algorithm to sort the array in ascending order.

• Display the sorted array obtained using Bubble Sort.

• Implement the Selection Sort algorithm on the same set of elements.

• Display the sorted array obtained using Selection Sort.

• Compile and run the program.

• Observe and compare the results of both sorting techniques.

Program: Bubble Sort and Selection Sort

```
#include <stdio.h>
#include <time.h>
// Function to perform Bubble Sort
void bubbleSort(int a[], int n) {
   int i, j, temp;
   for(i = 0; i < n - 1; i++) {
      for(j = 0; j < n - i - 1; j++) {
         if(a[j] > a[j + 1]) {
            temp = a[j];
            a[j] = a[j + 1];
            a[j + 1] = temp;
         }
      }
   }
}
// Function to perform Selection Sort
void selectionSort(int a[], int n) {
   int i, j, minIndex, temp;
```

```
    for(i = 0; i < n - 1; i++) {
        minIndex = i;
        for(j = i + 1; j < n; j++) {
            if(a[j] < a[minIndex])
                minIndex = j;
        }
        temp = a[i];
        a[i] = a[minIndex];
        a[minIndex] = temp;
    }
}
// Function to display array
void displayArray(int a[], int n) {
    for(int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}
int main() {
    int a[20], n, original[20];
    clock_t start, end;
    double time_taken;
    // Input array elements
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter array elements:\n");
    for(int i = 0; i < n; i++) {
        scanf("%d", &a[i]);
        original[i] = a[i]; // store original array for fair comparison
    }
```

```
    // Bubble Sort
    start = clock();
    bubbleSort(a, n);
    end = clock();
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nSorted array using Bubble Sort:\n");
    displayArray(a, n);
    printf("Time taken by Bubble Sort: %f seconds\n", time_taken);
    // Restore original array for Selection Sort
    for(int i = 0; i < n; i++)
        a[i] = original[i];
    // Selection Sort
    start = clock();
    selectionSort(a, n);
    end = clock();
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    printf("\nSorted array using Selection Sort:\n");
    displayArray(a, n);
    printf("Time taken by Selection Sort: %f seconds\n", time_taken);
    return 0;
}
```

Result:

Bubble Sort and Selection Sort algorithms were implemented successfully. Both algorithms sorted the given array correctly. It was observed that both algorithms have $O(n^2)$ time complexity and are suitable for small input sizes.

Lab Sheet 4: Compare Searching Algorithms – Linear Search and Binary Search

Aim:

To implement Linear Search and Binary Search algorithms in C and to compare their performance for searching an element in an array.

Procedure:

• Open Turbo C and create a new C program.

• Include the required header file stdio.h.

• Read the number of elements and array values from the user.

• Read the search key to be found.

• Implement the Linear Search algorithm to search for the key in the array.

• Display whether the element is found and its position.

• Sort the array before applying Binary Search.

• Implement the Binary Search algorithm on the sorted array.

• Display the search result obtained using Binary Search.

• Compile and run the program.

• Observe and compare the results of both searching techniques.

Program: Linear Search and Binary Search

```
#include <stdio.h>
#include <time.h>
// Linear Search Function
int linearSearch(int a[], int n, int key) {
    for (int i = 0; i < n; i++) {
        if (a[i] == key)
            return i;
    }
    return -1;
}
// Binary Search Function
int binarySearch(int a[], int n, int key) {
    int low = 0, high = n - 1, mid;
    while (low <= high) {
        mid = (low + high) / 2;
        if (a[mid] == key)
            return mid;
        else if (a[mid] < key)
```

```
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
// Function to sort array (for binary search)
void sortArray(int a[], int n) {
    int temp;
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (a[i] > a[j]) {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
}
int main() {
    int a[20], n, key, pos;
    clock_t start, end;
    double time_taken;
    // Input array
    printf("Enter number of elements: ");
    scanf("%d", &n);
    printf("Enter array elements:\n");
    for (int i = 0; i < n; i++)
        scanf("%d", &a[i]);
    printf("Enter element to search: ");
```

```
    scanf("%d", &key);
    // Linear Search
    start = clock();
    pos = linearSearch(a, n, key);
    end = clock();
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    if (pos != -1)
        printf("Linear Search: Element found at position %d\n", pos + 1);
    else
        printf("Linear Search: Element not found\n");
    printf("Time taken by Linear Search: %f seconds\n", time_taken);
    // Sort array for Binary Search
    sortArray(a, n);
    // Binary Search
    start = clock();
    pos = binarySearch(a, n, key);
    end = clock();
    time_taken = ((double)(end - start)) / CLOCKS_PER_SEC;
    if (pos != -1)
        printf("Binary Search: Element found at position %d\n", pos + 1);
    else
        printf("Binary Search: Element not found\n");
    printf("Time taken by Binary Search: %f seconds\n", time_taken);
    return 0;
}
```

Result:

Linear Search and Binary Search algorithms were implemented successfully. Linear Search works on both sorted and unsorted arrays but takes more time for large inputs, whereas Binary Search is faster with O(log n) time complexity but requires the array to be sorted.

Lab Sheet 5: Compare Sorting Algorithms – Insertion Sort and Merge Sort

Aim:

To implement Insertion Sort and Merge Sort algorithms in C and to compare their performance on the same set of input data.

Procedure:

• Open Turbo C and create a new C program.

• Include the required header file stdio.h.

• Read the number of elements and array values from the user.

• Implement the Insertion Sort algorithm and display the sorted array.

• Implement the Merge Sort algorithm using divide and conquer technique.

• Display the sorted array obtained using Merge Sort.

• Compile and run the program.

• Observe the execution and compare the performance of both algorithms.

Program: Insertion Sort and Merge Sort

```
#include <stdio.h>
#include <time.h>
void insertionSort(int a[], int n) {
   int i, key, j;
   for(i = 1; i < n; i++) {
      key = a[i];
      j = i - 1;
      while(j >= 0 && a[j] > key) {
         a[j + 1] = a[j];
         j--;
      }
      a[j + 1] = key;
   }
}
void merge(int a[], int low, int mid, int high) {
   int i = low, j = mid + 1, k = 0;
```

```
    int temp[100];
    while(i <= mid && j <= high) {
        if(a[i] < a[j])
            temp[k++] = a[i++];
        else
            temp[k++] = a[j++];
    }
    while(i <= mid)
        temp[k++] = a[i++];
    while(j <= high)
        temp[k++] = a[j++];

    for(i = low, k = 0; i <= high; i++, k++)
        a[i] = temp[k];
}
void mergeSort(int a[], int low, int high) {
    int mid;
    if(low < high) {
        mid = (low + high) / 2;
        mergeSort(a, low, mid);
        mergeSort(a, mid + 1, high);
        merge(a, low, mid, high);
    }
}
int main() {
    int a[100], b[100], n, i;
    clock_t start, end;
    double time_insert, time_merge;
    printf("Enter number of elements: ");
    scanf("%d", &n);
```

```
    printf("Enter array elements:\n");
    for(i = 0; i < n; i++) {
        scanf("%d", &a[i]);
        b[i] = a[i];    // Copy same input for Merge Sort
    }
    start = clock();
    insertionSort(a, n);
    end = clock();
    time_insert = (double)(end - start) / CLOCKS_PER_SEC;
    start = clock();
    mergeSort(b, 0, n - 1);
    end = clock();
    time_merge = (double)(end - start) / CLOCKS_PER_SEC;
    printf("\nSorted array using Insertion Sort:\n");
    for(i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n\nSorted array using Merge Sort:\n");
    for(i = 0; i < n; i++)
        printf("%d ", b[i]);
    printf("\n\nInsertion Sort Time: %f seconds", time_insert);
    printf("\nMerge Sort Time: %f seconds\n", time_merge);
    return 0;
}
```

Result:

Insertion Sort and Merge Sort algorithms were implemented successfully. Insertion Sort is efficient for small input sizes, while Merge Sort performs better for large datasets with O(n log n) time complexity.