|  |  |
| --- | --- |
|  | Department of Computer Science and Engineering  Chandpur Science and Technology University |

**LAB-02**

**Course Title**: Algorithm Design and Analysis Sessional

**Course Code**:CSE 2202

**Submitted To-**

**Mostafiz Ahammed**

*Lecturer****Department of Computer Science and Engineering****Chandpur Science and Technology University*

**Submitted By-**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sl** | Class Roll | Name | Batch | Signature |
| 01 | B210101022 | Md. Rabbi Ahammed | CSE-21 | rabbi |

**Date of Submission: 19 July, 2025**

**Experiment 01: Divide and Conquer – Algorithms for Sorting and Searching**

# Objective

To explore and analyze various algorithms that utilize the Divide and Conquer approach, including sorting, searching, geometry, and multiplication problems. We will study their design, implementation, and complexity.

# Algorithm

This lab includes the following divide and conquer algorithms:  
- Merge Sort (Sorting)  
- Quick Sort (Sorting)  
- Binary Search (Searching)  
- Closest Pair of Points (Computational Geometry)  
- Strassen’s Matrix Multiplication (Matrix Operation)  
- Karatsuba’s Algorithm (Large Integer Multiplication)

# Theoretical Solution of Given Problem

Divide and Conquer is an algorithmic paradigm that breaks a problem into sub-problems, solves them recursively, and combines their results. Each algorithm applies this principle in a different context:  
- Merge Sort and Quick Sort recursively sort subarrays.  
- Binary Search halves the array on each step.  
- Closest Pair of Points uses recursive geometry division.  
- Strassen’s and Karatsuba’s methods improve mathematical operations using recursion.

# Practical Work

## a. Pseudocode

MERGE\_SORT(arr, low, high):  
 if low < high:  
 mid = (low + high) / 2  
 MERGE\_SORT(arr, low, mid)  
 MERGE\_SORT(arr, mid+1, high)  
 MERGE(arr, low, mid, high)  
  
QUICK\_SORT(arr, low, high):  
 if low < high:  
 pi = PARTITION(arr, low, high)  
 QUICK\_SORT(arr, low, pi - 1)  
 QUICK\_SORT(arr, pi + 1, high)  
  
BINARY\_SEARCH(arr, target):  
 low = 0  
 high = length(arr) - 1  
 while low <= high:  
 mid = (low + high) / 2  
 if arr[mid] == target:  
 return mid  
 else if arr[mid] < target:  
 low = mid + 1  
 else:  
 high = mid - 1  
 return -1

## b. Source Code in C++

#include <iostream>  
using namespace std;  
  
// Merge Sort  
void merge(int arr[], int l, int m, int r) {  
 int n1 = m - l + 1;  
 int n2 = r - m;  
 int L[n1], R[n2];  
 for (int i = 0; i < n1; i++) L[i] = arr[l + i];  
 for (int j = 0; j < n2; j++) R[j] = arr[m + 1 + j];  
  
 int i = 0, j = 0, k = l;  
 while (i < n1 && j < n2) {  
 arr[k++] = (L[i] <= R[j]) ? L[i++] : 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);  
 }  
}  
  
// Quick Sort  
int partition(int arr[], int low, int high) {  
 int pivot = arr[high];  
 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);  
}  
  
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);  
 }  
}  
  
// Binary Search  
int binarySearch(int arr[], int n, int x) {  
 int l = 0, r = n - 1;  
 while (l <= r) {  
 int m = l + (r - l) / 2;  
 if (arr[m] == x) return m;  
 if (arr[m] < x) l = m + 1;  
 else r = m - 1;  
 }  
 return -1;  
}  
  
// Main  
int main() {  
 int arr[] = {12, 11, 13, 5, 6, 7};  
 int n = sizeof(arr)/sizeof(arr[0]);  
  
 mergeSort(arr, 0, n - 1);  
 cout << "Sorted with Merge Sort: ";  
 for (int i = 0; i < n; i++) cout << arr[i] << " ";  
 cout << endl;  
  
 quickSort(arr, 0, n - 1);  
 cout << "Sorted with Quick Sort: ";  
 for (int i = 0; i < n; i++) cout << arr[i] << " ";  
 cout << endl;  
  
 int key = 13;  
 int result = binarySearch(arr, n, key);  
 cout << "Binary Search Result for " << key << ": ";  
 if (result != -1) cout << "Found at index " << result << endl;  
 else cout << "Not Found" << endl;  
  
 return 0;  
}

# Analysis Table

**Algorithm Types and Time Complexities:**

|  |  |  |
| --- | --- | --- |
| Algorithm | Problem Type | Time Complexity |
| Merge Sort | Sorting | O(n log n) |
| Quick Sort | Sorting | O(n log n) avg |
| Binary Search | Searching | O(log n) |
| Closest Pair of Points | Geometry | O(n log n) |
| Strassen’s Multiplication | Matrix | ~O(n^2.81) |
| Karatsuba’s Algorithm | Large Integer Multiplication | O(n^1.58) |

**Best, Worst, Average Case & Space Complexities:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Best Case | Worst Case | Avg Case | Space |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n) |
| Quick Sort | O(n log n) | O(n^2) | O(n log n) | O(log n) |
| Binary Search | O(1) | O(log n) | O(log n) | O(1) |
| Closest Pair of Points | O(n log n) | O(n log n) | O(n log n) | O(n) |
| Strassen’s Multiplication | O(n^2.81) | O(n^2.81) | O(n^2.81) | O(n^2) |
| Karatsuba’s Algorithm | O(n^1.58) | O(n^1.58) | O(n^1.58) | O(n) |

# Observations

- Merge Sort is stable and efficient for large datasets.  
- Quick Sort performs well on average but has a poor worst case.  
- Binary Search is very fast on sorted data.  
- Closest Pair of Points illustrates divide and conquer in geometry.  
- Strassen’s and Karatsuba’s algorithms are efficient for large inputs in matrix and number multiplication.

# Challenges

- Handling recursion depth in Python and C++.  
- Implementing geometric algorithms requires precision.  
- Balancing readability and performance.  
- Ensuring correct base and merge cases in recursive code.

# Conclusion

Divide and Conquer algorithms offer elegant solutions to complex problems by breaking them into subproblems. They provide optimal or near-optimal performance across different domains such as sorting, searching, and mathematical computations. Their analysis and implementation help develop a deeper understanding of efficient algorithms.