PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY

Faculty of Engineering & Technology

Department of Information and Communication Engineering

LAB REPORT

Course name: Data Structure And Alghoritm Sessional

Course Code: ICE-2202

Submitted By:

Name: Sohel Ahamed

Roll: 220634

Session: 2021-22

2nd Year 2st Semester

Department of ICE, PUST

Submitted To:

Md. Anwar Hossain

Professor

Department of ICE, PUST

Date of Submission:01.03.2023

Index

Sl.	Problem Statement
1.	Write a program to sort a linear array using the bubble sort algorithm.
2.	Write a program to find an element using a linear search algorithm.
3.	Write a program to sort a linear array using the merge sort algorithm.
4.	Write a program to find an element using the binary search algorithm.
5.	Write a program to find a given pattern from text using the pattern matching algorithm.
6.	Write a program to implement a queue data structure along with its typical operations.
7.	Write a program to solve **n** queen's problem using backtracking.
8.	Consider a set **S = {5,10,12,13,15,18}** and **d = 30**. Write a program to solve the sum of subset problem.
9.	Write a program to solve the following **0/1 Knapsack** using dynamic programming approach **P = (15,25,13,23), weight W = (2,6,12,9), Knapsack C = 20**, and the number of items **n=4**.
10	Write a program to solve the **Tower of Hanoi** problem for the **N** disk.

Data Structure And Alghoritm Sessional

Problem No-1. Write a program to sort a linear array using the bubble sort algorithm

Theory:

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted.

Time Complexity:

Best Case: $O(n)$ (when the array is already sorted).
Worst Case: $O(n^2)$ (when the array is in reverse order).
Average Case: $O(n^2)$

Objective:

To implement the Bubble Sort algorithm to sort a linear array in ascending order.

Algorithm:

Step-1: Start with an unsorted array.
Step-2: Compare each pair of adjacent elements.
Step-3: Swap them if they are in the wrong order.
Step-3: Repeat the process for each element in the array.
Step-4: Continue until no swaps are needed, indicating the array is sorted.

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n;
    cin >> n;  // Input the number of elements in the array

    int arr[n];  // Declare an array of size n

    // Input the elements of the array
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
```

```
    }
    // Bubble Sort: Sorting the array in ascending order
    for (int i = 1; i <= n; i++)   // Outer loop for n iterations
    {
        for (int j = 0; j < n - i; j++)    // Inner loop to compare adjacent
elements
        {
            // If the current element is greater than the next one, swap them
            if (arr[j] > arr[j + 1])
            {
                swap(arr[j], arr[j + 1]);  // Swap the elements
            }
        }
    }
    // Output the sorted array
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";   // Print each element of the sorted array
    }
    cout << endl;  // Print a newline after outputting the sorted array

    return 0;  // Return from main
}
```

Input:

5

1 4 29 2 9

Output

1 2 4 9 29

Problem No-2. Write a program to find an element using a linear search algorithm.

Theory:

Linear Search is a method for finding a particular value in a list. It checks each element of the list one by one until a match is found or the whole list has been searched.

Time Complexity:

- **Best Case:** O(1) (when the target is the first element).
- **Worst Case:** O(n) (when the target is the last element or not present).

• **Average Case:** O(n).

Objective:
To implement the Linear Search algorithm to find an element in a linear array.

Algorithm:
Step-1: Start from the first element of the array.
Step-2: Compare the target element with each element of the array.
Step-3: If a match is found, return the index of the element.
Step-4: If no match is found, return -1.

Code:

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n;
    cin >> n;  // Input the number of elements in the array
    int arr[n]; // Declare an array of size n

    // Input the elements of the array
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }

    int K;
    cin >> K;  // Input the value to search for in the array

    int idx = -1;  // Initialize the index to -1 (indicating not found initially)

    // Linear search to find the value K in the array
    for (int i = 0; i < n; i++)
    {
        if (arr[i] == K)  // If the current element equals K
        {
            idx = i;  // Store the index of the element
            break;  // Exit the loop once the element is found
        }
    }

    // Output the result
```

```
    if (idx != -1)
    {
        cout << "Find The Value at Index: " << idx << endl;  // If found, print
the index
    }
    else
    {
        cout << "Value could not be found" << endl;  // If not found, print this
message
    }

    return 0;
}
```

Input:
5

12 39 349 29 3

3
Output
Find The Value Is:3

Problem No-3. Write a program to sort a linear array using the merge sort algorithm..

Theory:
Merge Sort is a divide-and-conquer algorithm that divides the array into two halves, sorts each half recursively, and then merges the two sorted halves.

Time Complexity:
- Best Case: O(nlogn).
- Worst Case: O(nlogn).
- Average Case: O(nlogn).

Objective:
To sort a linear array in ascending or descending order using the Merge

Sort algorithm.
Algorithm:

Step-1. Divide the array into two halves.

Step-2. Recursively sort each half.

Step-3. Merge the two sorted halves into a single sorted array.

.

Code

```cpp
#include <bits/stdc++.h>
using namespace std;

// Merge function to combine two sorted subarrays
void merge(int arr[], int l, int mid, int r)
{
    // Calculate the size of the two subarrays
    int n1 = mid - l + 1;
    int n2 = r - mid;

    // Temporary arrays to store the subarrays
    int a[n1];
    int b[n2];

    // Copy data into temporary arrays
    for (int i = 0; i < n1; i++)
    {
        a[i] = arr[l + i];
    }
    for (int i = 0; i < n2; i++)
    {
        b[i] = arr[mid + 1 + i];
    }

    int i = 0, j = 0, k = l;

    // Merge the temporary arrays back into the original array
    while (i < n1 && j < n2)
    {
        if (a[i] < b[j])
        {
```

```
            arr[k] = a[i];
            k++;
            i++;
        }
        else
        {
            arr[k] = b[j];
            k++;
            j++;
        }
    }

    // If there are any remaining elements in array 'a', copy them
    while (i < n1)
    {
        arr[k] = a[i];
        k++;
        i++;
    }
    // If there are any remaining elements in array 'b', copy them
    while (j < n2)
    {
        arr[k] = b[j];
        k++;
        j++;
    }
}
// Recursive mergesort function
void mergesort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Find the middle point
        int mid = (l + r) / 2;
        // Recursively sort the left and right halves
        mergesort(arr, l, mid);
        mergesort(arr, mid + 1, r);

        // Merge the two sorted halves
        merge(arr, l, mid, r);
    }
}
int main()
{
    int n;
```

```
    cin >> n;  // Input the number of elements in the array

    int arr[n];  // Declare an array of size n

    // Input the elements of the array
    for (int i = 0; i < n; i++)
        cin >> arr[i];

    // Call the mergesort function to sort the array
    mergesort(arr, 0, n - 1);  // Pass the correct range (0 to n-1)

    // Output the sorted array
    for (int i = 0; i < n; i++)
    {
        cout << arr[i] << " ";
    }
    cout << endl;

    return 0;
}
```

Input:

5

7 9 1 3 2

Output

1 2 3 7 9

Problem No-4. Write a program to find an element using the binary search algorithm.

Theory:

Binary Search is an efficient searching algorithm that works on sorted arrays. It repeatedly divides the search interval in half and compares the middle element with the target.

Time Complexity:

- Best Case: O(1) (when the target is the middle element).

- Worst Case: O(logn).
- Average Case: O(logn).

Objective:

To find an element in a sorted linear array using the Binary Search algorithm.

Algorithm:

Step-1. Start with the entire sorted array.

Step-2. Find the middle element of the array.

Step-3. If the middle element matches the target, return its index.

Step-4. If the middle element is greater than the target, search the left half.

Step-5. If the middle element is less than the target, search the right half.

Step-6. Repeat steps 2-5 until the target is found or the search interval is empty.

Code

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    int n;
    cin >> n;  // Input the size of the array
    int a[n];   // Declare an array of size n
    for (int i = 0; i < n; i++)
    {
        cin >> a[i];  // Input the elements of the array
    }

    int x;
    cin >> x;  // Input the element to be searched (x)

    int l = 0, r = n - 1;  // Set the left (l) and right (r) pointers for binary
search
    int idx = -1;  // Initialize the index to -1 (element not found by default)

    // Binary Search Loop
    while (l <= r)
    {
        int mid_index = (l + r) / 2;  // Find the middle index
```

```
        if (a[mid_index] == x)  // If element at mid_index is the target, found
the element
        {
            idx = mid_index;  // Set the index to the middle index
            break;  // Break the loop as the element is found
        }
        else if (x > a[mid_index])  // If target is greater, search in the right
half
        {
            l = mid_index + 1;  // Move left pointer to mid_index + 1
        }
        else  // If target is smaller, search in the left half
        {
            r = mid_index - 1;  // Move right pointer to mid_index - 1
        }
    }

    // Output the result
    if (idx != -1)  // If element was found, print the index
    {
        cout << "Index No: " << idx << endl;
    }
    else  // If element was not found, print -1
    {
        cout << "-1" << endl;
    }

    return 0;
}
```

Input:

5

1 6 8 9 11

11

Output

Index No:4

Problem No-5. Write a program to find a given pattern from text using the pattern matching algorithm.

Theory:

Pattern Matching is the process of finding a substring (pattern) within a larger string (text). A common algorithm for this is the Naive Pattern Matching algorithm.

Time Complexity:

- **Best Case:** O(n) (when the pattern is found at the beginning of the text).
- **Worst Case:** O(m(n−m+1)) (where mm is the length of the pattern and n is the length of the text).

Objective:

To find a given pattern in a text using the Pattern Matching algorithm.

Algorithm:

Step-1. Start from the first character of the text.

Step-2. Compare the pattern with the substring of the text starting at the current position.

Step-3. If the pattern matches, return the starting index.

Step-4. If the pattern does not match, move to the next character in the text.

Step-5. Repeat steps 2-4 until the pattern is found or the end of the text is reached.

Code

```
#include <bits/stdc++.h>
using namespace std;

int main()
{
    string t, p;
    cin >> t >> p;  // Input the text and pattern

    int tlen = t.size();  // Get the length of the text
    int plen = p.size();  // Get the length of the pattern

    bool flag = false;  // Flag to check if any match is found

    // Loop through the text to find all occurrences of the pattern
for (int i = 0; i < tlen; i++)
// Make sure there's enough space for the pattern
    {
        // Compare characters of the pattern with the corresponding part of the
text
        for (int j = 0; j < plen; j++)
        {
```

```
            if (t[i + j] != p[j])  // If characters do not match, break the inner
loop
            {
                break;
            }
            // If the entire pattern has been matched
            if (j == plen - 1)
            {
                cout << i << " ";   // Print the index where the pattern starts
in the text
                flag = true;  // Mark that
```

Input:

ababcabcababc

aba

Output

0 8

Problem No-6. Write a program to implement a queue data structure along with its typical operations.

Theory:

A Queue is a linear data structure that follows the First In First Out (FIFO) principle. It supports two main operations: enqueue (add an element to the rear) and dequeue (remove an element from the front).

Time Complexity:

- Enqueue: O(1).
- Dequeue: O(1).
- Display: O(n).

Objective:

To implement a Queue data structure and perform its typical operations (enqueue, dequeue, and display).

Algorithm:

Step-1. Define a queue using an array or linked list.

Step-2. Implement the enqueue operation to add elements to the rear of the queue.

Step-3. Implement the dequeue operation to remove elements from the front of the queue.

Step-4. Implement a function to display the elements of the queue.

Code:

```
#include <bits/stdc++.h>

using namespace std;

class my_queue
{
public:
  list<int> l;
  void push(int val)
  {
    l.push_back(val);
  }
  void pop()
  {
    l.pop_front();
  }
  int front()
  {
    return l.front();
  }
  int size()
  { #include <bits/stdc++.h>

using namespace std;

// Custom queue class using list container
class my_queue
{
public:
  list<int> l;  // List to hold the queue elements

  // Method to add an element to the back of the queue
  void push(int val)
  {
    l.push_back(val);  // Add the element at the back of the list
  }

  // Method to remove an element from the front of the queue
  void pop()
  {
    l.pop_front();  // Remove the element from the front of the list
  }
```

```
  // Method to get the element at the front of the queue
  int front()
  {
    return l.front();  // Return the element at the front of the list
  }

  // Method to get the current size of the queue
  int size()
  {
    return l.size();  // Return the size of the list
  }

  // Method to check if the queue is empty
  bool empty()
  {
    return l.empty();  // Return true if the list is empty, false otherwise
  }
};

int main()
{
  my_queue Q;  // Create an instance of the custom queue
  int n;
  cin >> n;  // Input the number of elements to be added to the queue

  // Input the elements and push them into the queue
  for (int i = 0; i < n; i++)
  {
    int x;
    cin >> x;  // Read an element
    Q.push(x);  // Push the element into the queue
  }

  // Output and pop elements until the queue is empty
  while (!Q.empty())
  {
    cout << Q.front() << " ";  // Print the front element of the queue
    Q.pop();  // Remove the front element from the queue
  }

  return 0;  // End of the program
}
```

Input:

5

6 7 8 9 10

Output

6 7 8 9 10 8

Problem No-7. Write a program to solve the n-queens problem using backtracking.

Theory:

The N-Queens problem is a classic backtracking problem where the goal is to place N queens on an N×N chessboard such that no two queens threaten each other.

Time Complexity:

- Worst Case: O(N!) (exponential time complexity due to backtracking).

Objective:

To solve the N-Queens problem using backtracking.

Algorithm:

Step-1. Start placing queens row by row.

Step-2. For each row, try placing the queen in each column.

Step-3. Check if the current placement is safe (no conflicts with previously placed queens).

Step-4. If safe, move to the next row and repeat the process.

Step-5. If not safe, backtrack and try the next column.

Step-6. Repeat until all queens are placed or no solution is found.

Code:

```
#include <bits/stdc++.h>
using namespace std;

// Function to check if a queen can be placed at board[row][col]
bool isSafe(const vector<vector<int>> &board, int row, int col, int n)
{
    // Check the same column
    for (int i = 0; i < row; i++)
    {
        if (board[i][col] == 1)
            return false;
```

```
    }
    // Check the upper-left diagonal
    for (int i = row, j = col; i >= 0 && j >= 0; i--, j--)
    {
        if (board[i][j] == 1)
            return false;
    }
    // Check the upper-right diagonal
    for (int i = row, j = col; i >= 0 && j < n; i--, j++)
    {
        if (board[i][j] == 1)
            return false;
    }
    return true;
}
// Function to print the chessboard
void printBoard(const vector<vector<int>> &board, int n)
{
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < n; j++)
        {
            cout << (board[i][j] == 1 ? "Q " : ". ");
        }
        cout << endl;
    }
    cout << endl;
}
// Function to solve the N-Queens problem and find all solutions
void solveNQueens(vector<vector<int>> &board, int row, int n, int &solutions)
{
    // Base case: If all queens are placed
    if (row == n)
    {
        solutions++;
        cout << "Solution " << solutions << ":\n";
        printBoard(board, n);
        return;
    }
    // Try placing a queen in each column of the current row
    for (int col = 0; col < n; col++)
    {
        if (isSafe(board, row, col, n))
        {
            // Place the queen
```

```
            board[row][col] = 1;

            // Recur to place the rest of the queens
            solveNQueens(board, row + 1, n, solutions);

            // Backtrack: Remove the queen
            board[row][col] = 0;
        }
    }
}
// Main function
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cout << "Enter the value of N: ";
    cin >> n;
    // Create an NxN chessboard initialized to 0
    vector<vector<int>> board(n, vector<int>(n, 0));

    int solutions = 0; // Count the number of solutions
    solveNQueens(board, 0, n, solutions);
    if (solutions == 0)
    {
        cout << "No solutions exist for " << n << "-Queens problem." << endl;
    }
    else
    {
        cout << "Total solutions for " << n << "-Queens problem: " << solutions
<< endl;
    }
    return 0;
}
```

Input:

Enter the value of N:4

Output

Solution 1:

. Q . .

. . . Q

Q . . .

. . Q .

Solution 2:

. . Q .

Q . . .

. . . Q

. Q . .

Problem No-8 Consider a set S={5,10,12,13,15,18} and d=30. Write a program to solve the sum of subsets problem.

Theory:

The Sum of Subsets problem involves finding a subset of a given set of numbers that adds up to a target sum dd.

Time Complexity:

- Worst Case: O(2n) (exponential time complexity due to backtracking).

Objective:

To solve the Sum of Subsets problem using backtracking.

Algorithm:

Step-1. Start from the first element of the set.

Step-2. Include the current element in the subset and check if the sum equals the target.

Step-3. If the sum equals the target, print the subset.

Step-4. If the sum is less than the target, move to the next element and repeat the process.

Step-5. If the sum exceeds the target, backtrack and exclude the current element.

Code:

```cpp
#include <bits/stdc++.h>
using namespace std;
void suubsum(vector<int> v, vector<int> subset, int index, int sum, int target)
{
    if (sum == target)
    {
        cout << "{";
        {
            for (int val : subset)
            {
                cout << val << " ";
            }
            cout << "}" << endl;
        }
        return;
    }
    if (sum > target || index >= v.size())
    {
        return;
    }
    subset.push_back(v[index]);
    suubsum(v, subset, index + 1, sum + v[index], target);

    subset.pop_back();
    suubsum(v, subset, index + 1, sum, target);
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    vector<int> v(n);
    for (int i = 0; i < n; i++)
    {
        cin >> v[i];
    }
    int d = 30;
    vector<int> subset;

    suubsum(v, subset, 0, 0, d);
    return 0;
}
```

Input:

6

5 10 12 13 15 18

Output

{5 10 15 }

{5 12 13 }

{12 18 }

Problem No-9 Write a program to solve the 0/1 Knapsack problem using a dynamic programming approach with profits P=(15,25,13,23) weights W=(2,6,12,9) knapsack capacity C=20, and the number of items n=4.

Theory:

The 0/1 Knapsack problem is a classic optimization problem where the goal is to maximize the total profit by selecting items with given weights and profits, without exceeding the knapsack's capacity.

Time Complexity:

- Time Complexity: O(n×C) (where n is the number of items and CC is the knapsack capacity).

Objective:

To solve the 0/1 Knapsack problem using dynamic programming.

Algorithm:

Step-1. Create a 2D DP table where dp[i][j] represents the maximum profit for the first i items and capacity j.

Step-2. Initialize the table such that dp[0][j] = 0 and dp[i][0] = 0.

Step-3.Fill the table using the following rules:

- If the weight of the current item is less than or equal to the capacity, choose the maximum between including and excluding the item.
- If the weight exceeds the capacity, exclude the item.

Step-4. The value dp[n][C] will give the maximum profit.

Code:

```cpp
// Date: 25-02-2025 at 22:12 BST
// Link:
#include <bits/stdc++.h>
using namespace std;

int knapsack(int C, const vector<int> &W, const vector<int> &P, int n)
{
    // Create a 2D DP table with (n+1) rows and (C+1) columns
    vector<vector<int>> dp(n + 1, vector<int>(C + 1, 0));
    // Build the DP table
    for (int i = 1; i <= n; i++)
    {
        for (int j = 1; j <= C; j++)
        {
            if (W[i - 1] <= j)
            {
                // Include the item if it fits in the knapsack
                dp[i][j] = max(P[i - 1] + dp[i - 1][j - W[i - 1]], dp[i - 1][j]);
            }
            else
            {
                // Exclude the item if it doesn't fit
                dp[i][j] = dp[i - 1][j];
            }
        }
    }

    // The maximum profit is stored in dp[n][C]
    return dp[n][C];
}

int main()
{
    vector<int> P = {15, 25, 13, 23}; // Profits
    vector<int> W = {2, 6, 12, 9};    // Weights
    int C = 20;                       // Knapsack capacity
    int n = P.size();                 // Number of items

    int maxProfit = knapsack(C, W, P, n);
    cout << "Maximum Profit: " << maxProfit << endl;

    return 0;
}
```

OUTPUT

Maximum Profit: 63

Problem No-10 Write a program to solve the Tower of Hanoi problem for N disks.

Theory:

The Tower of Hanoi is a mathematical puzzle where the goal is to move NN disks from the source rod to the destination rod using an auxiliary rod, following specific rules.

Time Complexity:

- Time Complexity: O(2n)O(2n) (exponential time complexity due to recursion).

Objective:

To solve the Tower of Hanoi problem for NN disks.

Algorithm:

Step-1. Move N−1N−1 disks from the source rod to the auxiliary rod.

Step-2. Move the NN-th disk from the source rod to the destination rod.

Step-3. Move the N−1N−1 disks from the auxiliary rod to the destination rod.

Code:

```
#include <iostream>
using namespace std;

// Function to solve the Tower of Hanoi problem
void towerOfHanoi(int N, char source, char auxiliary, char destination)
{
    if (N == 1)
    {
        // Base case: Move the top disk from source to destination
        cout << "Move disk 1 from " << source << " to " << destination << endl;
        return;
    }

    // Move N-1 disks from source to auxiliary using destination as the auxiliary
rod
    towerOfHanoi(N - 1, source, destination, auxiliary);

    // Move the Nth disk from source to destination
    cout << "Move disk " << N << " from " << source << " to " << destination <<
endl;
```

```
    // Move N-1 disks from auxiliary to destination using source as the auxiliary
rod
    towerOfHanoi(N - 1, auxiliary, source, destination);
}

int main()
{
    int N;
    cout << "Enter the number of disks: ";
    cin >> N;

    // Solve the Tower of Hanoi problem for N disks
    towerOfHanoi(N, 'A', 'B', 'C');

    return 0;
}
```

INPUT:

Enter the number of disks: 3

OUTPUT:

Enter the number of disks: 3

Move disk 1 from A to C

Move disk 2 from A to B

Move disk 1 from C to B

Move disk 3 from A to C

Move disk 1 from B to A

Move disk 2 from B to C

Move disk 1 from A to C