**Pabna University of Science and Technology**

A logo with a computer and symbols

AI-generated content may be incorrect.

**Faculty of Engineering and Technology**

**Department of Information and Communication Engineering**

**Lab Report**

Course Code: **ICE-2202**

Course title: **Data Structure and Algorithm Sessional**

|  |  |
| --- | --- |
| **Submitted By:**  **Name:** Shahriyar  **Roll:** 220616  **Reg:** 1065463  **Session:**2021-2022  2 nd Year 2 nd semester  Department of Information and  Communication Engineering,  Pabna University of Science And Technology,PUST | **Submitted To:**  **MD. Anwar Hossain**  professor  Department of  Information and Communication Engineering,  Pabna University of Science And Technology, PUST. |

**Date of Submission: 01/03/2025**

**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 **profits 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. | |

**Lab-1: Write a program to sort a linear array using the Bubble Sort Algorithm.**

**Objective:**

The objective of this lab is to implement the Bubble Sort algorithm in C++ to sort a linear array of integers in ascending order.

**Introduction:**

Sorting is a fundamental operation in computer science and is essential in many applications. 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. This process continues until the list is sorted.

**Algorithm:**

1. Start at the beginning of the array.
2. Compare each pair of adjacent elements.
3. Swap the elements if they are in the wrong order.
4. Repeat the process for n-1 passes, where n is the size of the array.
5. The largest unsorted element "bubbles up" to its correct position after each pass.

**Source code:**

#include <bits/stdc++.h>

using namespace std;

void bubbleSort(int arr[], int n) {

    for (int i = 0; i < n - 1; i++)

        bool swapped = false;

        for (int j = 0; j < n - i - 1; j++) {

            if (arr[j] > arr[j + 1]) {

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

                swapped = true;

            }

        }

        if (!swapped)

            break;

    }

}

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

    for (int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

}

int main() {

    int n;

    cout << "Enter the number of elements in the array: ";

    cin >> n;

    int arr[n];

    cout << "Enter " << n << " elements: ";

    for (int i = 0; i < n; i++) {

        cin >> arr[i];

    }

    cout << "Original array: ";

    printArray(arr, n);

    bubbleSort(arr, n);

    cout << "Sorted array: ";

    printArray(arr, n);

    return 0;

}

**Output:**

**Input Array**: 64 34 25 12 22 11 90  
**Sorted Array**: 11 12 22 25 34 64 90

**Lab-2: write a program to find an element using a linear search Algorithm .**

**Objective:**

To write a C++ program that implements the linear search algorithm to find an element in an array.

**Introduction:**

Linear search is a straightforward algorithm used to find a specific value in an array. It checks each element of the array sequentially until the desired value is found or the array is exhausted. This algorithm is particularly useful for small datasets or unsorted arrays where simplicity is prioritized over efficiency.

**Algorithm:**

1. Begin at the first element of the array.
2. Compare the target value with the current element.
3. If a match is found, return the index of the element.
4. If the end of the array is reached without finding the target, conclude that the element is not in the array.

**Program Code:**

#include <iostream>

using namespace std;

int linearSearch(int arr[], int size, int target) {

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

if (arr[i] == target) {

return i;

}

}

return -1;

}

int main() {

int n, target;

cout << "Enter the number of elements in the array: ";

cin >> n;

int arr[n];

cout << "Enter the elements of the array: ";

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

cin >> arr[i];

}

cout << "Enter the element to search for: ";

cin >> target;

int result = linearSearch(arr, n, target);

if (result != -1) {

cout << "Element found at index " << result << "." << endl;

} else {

cout << "Element not found in the array." << endl;

}

return 0;

}

**Test Cases:**

| **Test Case** | **Array** | **Target Element** | **Expected Output** | **Actual Output** |
| --- | --- | --- | --- | --- |
| 1 | {10, 20, 30, 40} | 30 | Element found at index 2 | Element found at index 2 |
| 2 | {5, 15, 25, 35} | 50 | Element not found | Element not found |
| 3 | {1, 2, 3, 4, 5} | 1 | Element found at index 0 | Element found at index 0 |
| 4 | {9, 8, 7, 6, 5} | 5 | Element found at index 4 | Element found at index 4 |

**Lab-3: Write a program to sort a linear array using the Merge Sort Algorithm .**

**Objective:**

The purpose of this lab is to implement the Merge Sort algorithm in C++ to sort a linear array. This experiment demonstrates the concept of divide-and-conquer and evaluates the efficiency of Merge Sort in terms of time complexity.

**Theory:**

Merge Sort is a recursive algorithm based on the divide-and-conquer paradigm. The algorithm divides the input array into two halves, recursively sorts them, and then merges the two sorted halves.

**Algorithm:**

1. Define a function to merge two sorted subarrays.
2. Implement the recursive function for sorting the array.
3. Create a driver function to input the array, call the merge sort function, and display the sorted array.

**Source Code:**

#include <iostream>

#include <vector>

using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {

int n1 = mid - left + 1; // Size of the left subarray

int n2 = right - mid; // Size of the right subarray

vector<int> leftArr(n1), rightArr(n2);

for (int i = 0; i < n1; ++i)

leftArr[i] = arr[left + i];

for (int i = 0; i < n2; ++i)

rightArr[i] = arr[mid + 1 + i];

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

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

if (leftArr[i] <= rightArr[j]) {

arr[k++] = leftArr[i++];

} else {

arr[k++] = rightArr[j++];

}

}

while (i < n1) {

arr[k++] = leftArr[i++];

}

while (j < n2) {

arr[k++] = rightArr[j++];

}

}

void mergeSort(vector<int>& arr, int left, int right) {

if (left < right) {

int mid = left + (right - left) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

int main() {

int n;

cout << "Enter the number of elements in the array: ";

cin >> n;

vector<int> arr(n);

cout << "Enter the elements of the array:\n";

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

cin >> arr[i];

}

mergeSort(arr, 0, n - 1);

cout << "Sorted array:\n";

for (const auto& num : arr) {

cout << num << " ";

}

cout << endl;

return 0;

}

**Sample Input/Output:**

**Input:**

Enter the number of elements in the array: 5

Enter the elements of the array:

38 27 43 3 9

**Output:**

Sorted array:

3 9 27 38 43

**Lab-4: Write a program to find an element using the binary search algorithm.**

**Objective:**

To implement the binary search algorithm in C++ to find an element in a sorted array efficiently.

**Theory:**

Binary search is an efficient searching algorithm that finds an element in a sorted array by repeatedly dividing the search interval in half. If the target element is smaller than the middle element, the search continues in the left half; otherwise, it continues in the right half. The process repeats until the element is found or the interval is empty.

**Algorithm:**

1. Start with a sorted array and define two pointers: low (initially 0) and high (initially last index of the array).
2. Calculate the middle index: mid = (low + high) / 2.
3. If arr[mid] is equal to the target, return the index.
4. If the target is smaller than arr[mid], search in the left half (high = mid - 1).
5. If the target is larger than arr[mid], search in the right half (low = mid + 1).
6. Repeat steps 2-5 until low is greater than high.
7. If the element is not found, return -1.

**Source code:**

#include <iostream>

using namespace std;

int binarySearch(int arr[], int size, int target) {

int low = 0, high = size - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

if (arr[mid] == target)

return mid;

else if (arr[mid] < target)

low = mid + 1;

else

high = mid - 1;

}

return -1;

}

int main() {

int arr[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91};

int size = sizeof(arr) / sizeof(arr[0]);

int target;

cout << "Enter the element to search: ";

cin >> target;

int result = binarySearch(arr, size, target);

if (result != -1)

cout << "Element found at index " << result << endl;

else

cout << "Element not found in the array" << endl;

return 0;

}

**I*nput-Output:***

***Example 1:***

***Input:***

Enter the element to search: 23

**Output:**

Element found at index 5

**Example 2:**

**Input:**

Enter the element to search: 50

**Output:**

Element not found in the array

**Lab-5: Write a program to find a given pattern from text using the pattern matching algorithm.**

**Objective:** The objective of this lab is to implement a pattern-matching algorithm in C++ that finds occurrences of a given pattern in a text using the Knuth-Morris-Pratt (KMP) algorithm.

**Theory:** Pattern matching is a fundamental problem in computer science, where we search for a pattern within a larger text.

**Algorithm :**

1. Compute the LPS array for the given pattern.
2. Traverse the text and compare characters using the LPS array.
3. If a match is found, store the position.
4. Continue until the entire text is processed.

**Source code:**

#include <iostream>

#include <vector>

using namespace std;

void computeLPSArray(string pattern, vector<int>& lps) {

int m = pattern.length(), j = 0;

lps[0] = 0;

for (int i = 1; i < m; ) {

if (pattern[i] == pattern[j]) {

lps[i] = j + 1;

j++; i++;

} else {

if (j != 0) {

j = lps[j - 1];

} else {

lps[i] = 0;

i++;

}

}

}

}

void KMPsearch(string text, string pattern) {

int n = text.length(), m = pattern.length();

vector<int> lps(m, 0);

computeLPSArray(pattern, lps);

int i = 0, j = 0;

while (i < n) {

if (text[i] == pattern[j]) {

i++; j++;

}

if (j == m) {

cout << "Pattern found at index " << i - j << endl;

j = lps[j - 1];

} else if (i < n && text[i] != pattern[j]) {

if (j != 0) {

j = lps[j - 1];

} else {

i++;

}

}

}

}

int main() {

string text, pattern;

cout << "Enter the text: ";

getline(cin, text);

cout << "Enter the pattern to search: ";

getline(cin, pattern);

KMPsearch(text, pattern);

return 0;

}

**Input & Output :**

**Input:**

Enter the text: abcabcabcabcd

Enter the pattern to search: abcabcd

**Output:**

Pattern found at index 6.

**Lab-6: Write a program to implement a queue data structure along with its typical operations.**

**Objective:** The objective of this lab is to implement a queue data structure in C++ and perform standard queue operations such as enqueue (insertion), dequeue (removal), peek (front element), and checking if the queue is empty or full.

**Theory:** A queue is a linear data structure that follows the First-In-First-Out (FIFO) principle. Elements are added to the back (rear) and removed from the front. The primary operations of a queue include:

* Enqueue: Add an element to the rear of the queue.
* Dequeue: Remove an element from the front of the queue.
* Peek: Get the front element without removing it.
* isEmpty: Check if the queue is empty.
* isFull: Check if the queue is full (for an array-based queue).

**Algorithm:**

1. Define a class Queue with attributes for the front, rear, size, and an array to store elements.
2. Implement the enqueue operation by adding elements at the rear and updating the rear index.
3. Implement the dequeue operation by removing elements from the front and updating the front index.
4. Implement helper functions like isEmpty, isFull, and peek.
5. Provide a menu-driven program to demonstrate the working of the queue.

**Source code**:

#include <iostream>

#define SIZE 5

using namespace std;

class Queue {

private:

int front, rear;

int arr[SIZE];

public:

Queue() {

front = -1;

rear = -1;

}

bool isEmpty() {

return (front == -1 || front > rear);

}

bool isFull() {

return (rear == SIZE - 1);

}

void enqueue(int value) {

if (isFull()) {

cout << "Queue is full!\n";

return;

}

if (front == -1) front = 0;

arr[++rear] = value;

cout << "Inserted " << value << " into the queue.\n";

}

void dequeue() {

if (isEmpty()) {

cout << "Queue is empty!\n";

return;

}

cout << "Removed " << arr[front] << " from the queue.\n";

front++;

if (front > rear) {

front = -1;

rear = -1;

}

}

void peek() {

if (isEmpty()) {

cout << "Queue is empty!\n";

} else {

cout << "Front element: " << arr[front] << "\n";

}

}

void display() {

if (isEmpty()) {

cout << "Queue is empty!\n";

return;

}

cout << "Queue elements: ";

for (int i = front; i <= rear; i++) {

cout << arr[i] << " ";

}

cout << "\n";

}

};

int main() {

Queue q;

int choice, value;

do {

cout << "\nQueue Operations:\n1. Enqueue\n2. Dequeue\n3. Peek\n4. Display\n5. Exit\nEnter choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert: ";

cin >> value;

q.enqueue(value);

break;

case 2:

q.dequeue();

break;

case 3:

q.peek();

break;

case 4:

q.display();

break;

case 5:

cout << "Exiting...\n";

break;

default:

cout << "Invalid choice!\n";

}

} while (choice != 5);

return 0;

}

**Input-Output**:

**Input:**

10

20

30

**Output:**

Inserted 10 into the queue.

Inserted 20 into the queue.

Inserted 30 into the queue.

Front element: 10

Removed 10 from the queue.

Queue elements: 20 30

**Lab-7: Write a program to solve n queen’s problem using backtracking.**

**Objective:**

To implement and understand the N-Queens problem using the backtracking algorithm in C++.

**Theory:**

The N-Queens problem involves placing N chess queens on an N×N board so that no two queens threaten each other. This means:

1. No two queens should be in the same row.
2. No two queens should be in the same column.
3. No two queens should be in the same diagonal.

Backtracking is used to find a valid arrangement by placing queens one by one and checking constraints recursively.

**Algorithm:**

1. Start with an empty board.
2. Place a queen in the first column.
3. Move to the next column and place a queen such that it does not attack the previous ones.
4. If no valid position is found, backtrack to the previous column and try the next row.
5. Repeat until all queens are placed.
6. Print the solution.

**Source code:**

#include <iostream>

#include <vector>

using namespace std;

#define N 8

void printSolution(vector<vector<int>> &board) {

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

for (int j = 0; j < N; j++) {

cout << (board[i][j] ? "Q " : "- ");

}

cout << endl;

}

cout << endl;

}

bool isSafe(vector<vector<int>> &board, int row, int col) {

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

if (board[row][i])

return false;

}

for (int i = row, j = col; i >= 0 && j >= 0; i--, j--) {

if (board[i][j])

return false;

}

for (int i = row, j = col; i < N && j >= 0; i++, j--) {

if (board[i][j])

return false;

}

return true;

}

bool solveNQueensUtil(vector<vector<int>> &board, int col) {

if (col >= N)

return true;

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

if (isSafe(board, i, col)) {

board[i][col] = 1;

if (solveNQueensUtil(board, col + 1))

return true;

board[i][col] = 0;

}

}

return false;

}

void solveNQueens() {

vector<vector<int>> board(N, vector<int>(N, 0));

if (!solveNQueensUtil(board, 0)) {

cout << "Solution does not exist" << endl;

return;

}

printSolution(board);

}

int main() {

solveNQueens();

return 0;

}

**Input-Output** :

**Input:**

N = 8 (predefined in the program)

**Output:** An 8×8 board representation where 'Q' marks a queen's position, and '-' marks an empty space.

**Example:**

Q - - - - - - -

- - - - Q - - -

- - - - - - - Q

- - - - - Q - -

- - Q - - - - -

- - - - - - Q -

- Q - - - - - -

- - - Q - - - -

**Lab-8: Consider a set S={5,10,12,13,15,18} and d=30. Write a program to solve the sum of subset problem.**

**Objective:**

The goal of this experiment is to find all subsets of a given set whose sum equals a specified target using a backtracking approach.

**Problem Statement:**

Given a set S={5,10,12,13,15,18} and a target sum d=30, find all subsets of S that sum to 30.

**Algorithm :**

The solution employs a backtracking approach:

1. Start with an empty subset.
2. Recursively add elements while ensuring the sum does not exceed the target.
3. If the sum matches the target, print the subset.
4. Backtrack to explore other possibilities.

**Source code:**

#include <iostream>

#include <vector>

using namespace std;

void sumOfSubsets(vector<int>& set, vector<int>& subset, int index, int sum, int target) {

if (sum == target) {

cout << "Subset: ";

for (int num : subset) {

cout << num << " ";

}

cout << endl;

return;

}

for (int i = index; i < set.size(); i++) {

if (sum + set[i] <= target) {

subset.push\_back(set[i]);

sumOfSubsets(set, subset, i + 1, sum + set[i], target);

subset.pop\_back();

}

}

}

int main() {

vector<int> S = {5, 10, 12, 13, 15, 18};

int d = 30;

vector<int> subset;

cout << "Subsets with sum " << d << ":" << endl;

sumOfSubsets(S, subset, 0, 0, d);

return 0;

}

**Input and Output:**

**Input:**

* The set S={5,10,12,13,15,18}
* Target sum d=30

**Output:** The program prints all subsets whose sum equals 30.

**Output:**

Subsets with sum 30:

Subset: 5 10 15

Subset: 5 12 13

Subset: 12 18

**Lab-9: Write a program to solve the following 0/1 knapsack using dynamic programming approach profits P=(15,25,13,23), weight W=(2,6,12,9), knapsack C=20,and the number of items n=4.**

**Objective:**

The objective of this lab is to implement and analyze the 0/1 knapsack problem using a dynamic programming approach in C++.

**Problem Statement:**

Given:

* Profits: P={15,25,13,23}
* Weights: W={2,6,12,9}
* Knapsack capacity: C=20
* Number of items: n=4

Determine the maximum profit that can be obtained by selecting items without exceeding the knapsack capacity.

**Algorithm :**

1. Create a 2D table dp[n+1][C+1], where dp[i][w] stores the maximum profit for i items and weight w.
2. Initialize dp[0][w] = 0 (no items).
3. Use the recurrence relation: dp[i][w]=max⁡(P[i−1]+dp[i−1][w−W[i−1]],dp[i−1][w])
4. If the current item's weight is within the capacity, choose the maximum between including and excluding it.

**Source code:**

#include <iostream>

#include <vector>

using namespace std;

int knapsack(int C, vector<int>& weights, vector<int>& profits, int n) {

vector<vector<int>> dp(n + 1, vector<int>(C + 1, 0));

for (int i = 1; i <= n; i++) {

for (int w = 0; w <= C; w++) {

if (weights[i - 1] <= w)

dp[i][w] = max(profits[i - 1] + dp[i - 1][w - weights[i - 1]], dp[i - 1][w]);

else

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

}

}

return dp[n][C];

}

int main() {

vector<int> profits = {15, 25, 13, 23};

vector<int> weights = {2, 6, 12, 9};

int C = 20;

int n = profits.size();

int maxProfit = knapsack(C, weights, profits, n);

cout << "Maximum Profit: " << maxProfit << endl;

return 0;

}

**Input/Output:**

**Input**

* Profits: {15, 25, 13, 23}
* Weights: {2, 6, 12, 9}
* Knapsack Capacity: 20

**Output:**

Maximum Profit: 40

**Lab-10: Write a program to solve the Tower of Hanoi problem for the N disk.**

**Introduction** :

The Tower of Hanoi is a mathematical puzzle that involves moving a set of disks from one peg to another using a third peg as an auxiliary. The challenge is to move all the disks following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or on an empty peg.
3. The objective is to move all disks from the source peg to the destination peg using the auxiliary peg.

**Algorithm** :

The Tower of Hanoi problem can be efficiently solved using recursion. The recursive approach follows these steps:

1. Move the top disks from the source peg to the auxiliary peg.
2. Move the th disk (largest disk) from the source peg to the destination peg.
3. Move the disks from the auxiliary peg to the destination peg.

**Source code:**

#include <iostream>

using namespace std;

int m=0;

void towerOfHanoi(int n, char source, char auxiliary, char destination) {

if (n == 1) {

m++;

cout << m<<" Move disk 1 from " << source << " to " << destination << endl;

return;

}

towerOfHanoi(n - 1, source, destination, auxiliary);

m++;

cout << m<<" Move disk " << n << " from " << source << " to " << destination << endl;

towerOfHanoi(n - 1, auxiliary, source, destination);

}

int main() {

int n;

cout << "Enter the number of disks: ";

cin >> n;

towerOfHanoi(n, 'A', 'B', 'C');

return 0;

}

**4. Input and Output:**

**Sample Input:**

Enter the number of disks: 3

**Sample Output:**

1 Move disk 1 from A to C

2 Move disk 2 from A to B

3 Move disk 1 from C to B

4 Move disk 3 from A to C

5 Move disk 1 from B to A

6 Move disk 2 from B to C

7 Move disk 1 from A to C