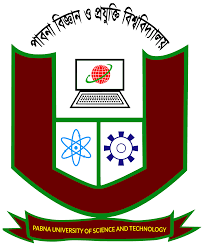
**Pabna University of Science and Technology**



**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:** Saklam Sakib  **Roll:** 220609  **Session:**2021-2022  2 nd Year 2 nd semester  Department of Information and  Communication Engineering,  PUST | **Submitted To:**  **MD. Anwar Hossain**  Chairman and Associate professor,  Department of  Information and Communication Engineering,  Pabna University of Science And Technology, Pabna. |

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

|  |  |
| --- | --- |
| **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. | |

## **Problem 1: Bubble Sort**

### Problem Statement:

Write a program to sort a linear array using the bubble sort algorithm.

### Illustration of the Problem:

Bubble sort works by repeatedly swapping adjacent elements if they are in the wrong order. This process is repeated until the array is sorted.

### Step-by-Step Algorithm

1.Start from the first element.

2.Compare the current element with the next element.

3.Swap them if they are in the wrong order.

4.Repeat steps 2 and 3 for all elements in the array.

5.Continue the process for all elements until no swaps are required.

Source Code:

#include <iostream>

using namespace std;

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

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

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

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

swap(arr[j], arr[j + 1]);

}

}

}

}

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

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

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

}

cout << endl;

}

int main() {

int arr[] = {64, 34, 25, 12, 22, 11, 90};

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

cout << "Unsorted array: ";

printArray(arr, n);

bubbleSort(arr, n);

cout << "Sorted array: ";

printArray(arr, n);

return 0;

}

**Input :**

Unsorted array: 64 34 25 12 22 11 90

**OutPut:**

Sorted array: 11 12 22 25 34 64 90

## Problem 2: Linear Search

### Problem Statement

Write a program to find an element using a linear search algorithm.

### Illustration of the Problem

Linear search scans each element of the array sequentially until the desired element is found.

### Step-by-Step Algorithm

1.Start from the first element of the array.

2.Compare the element with the target value.

3.If a match is found, return the index.

4.If no match is found, continue to the next element.

5.If the element is not found, return -1.

### **Source Code**

#include <iostream>

using namespace std;

int linearSearch(int arr[], int n, int x) {

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

if (arr[i] == x)

return i;

}

return -1;

}

int main() {

int arr[] = {10, 20, 30, 40, 50};

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

int x = 30;

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

if (result == -1)

cout << "Element not found";

else

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

return 0;

}

**Input:**

Array: 10 20 30 40 50

Element to search: 30

**Output:**

Element found at index 2

## Problem 3: Merge Sort

### Problem Statement

Write a program to sort a linear array using the merge sort algorithm.

### Illustration of the Problem

Merge sort is a divide-and-conquer algorithm that splits an array into halves, sorts each half, and then merges them.

### Step-by-Step Algorithm

1.Divide the array into two halves.

2.Recursively sort each half.

3.Merge the two sorted halves back together.

### Source Code

#include <iostream>using namespace std;

// Merge two subarrays L and R into arrvoid merge(int arr[], int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int L[n1], R[n2];

// Copy data to temporary arrays

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

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

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

R[j] = arr[mid + 1 + j];

// Merge the temporary arrays back into arr[left..right]

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

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

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

// Copy the remaining elements of L[], if any

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

// Copy the remaining elements of R[], if any

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

// Merge Sort functionvoid mergeSort(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);

}

}

// Print an arrayvoid printArray(int arr[], int size) {

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

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

cout << endl;

}

// Driver codeint main() {

int arr[] = {38, 27, 43, 3, 9, 82, 10};

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

cout << "Unsorted array: ";

printArray(arr, size);

mergeSort(arr, 0, size - 1);

cout << "Sorted array: ";

printArray(arr, size);

return 0;

}

### ****Input:****

Unsorted array: 38 27 43 3 9 82 10

### ****Output:****

Sorted array: 3 9 10 27 38 43 82

## Problem 4: Binary Search

### Problem Statement

Write a program to find an element using the binary search algorithm.

### Illustration of the Problem

Binary search works on sorted arrays by repeatedly dividing the search interval in half.

### Step-by-Step Algorithm

1.Sort the array if it is not already sorted.

2.Set two pointers, low and high, at the start and end of the array.

3.Find the middle element.

4.If the middle element is the target, return its index.

5.If the target is smaller, search in the left half.

6.If the target is larger, search in the right half.

7.Repeat steps 3-6 until the element is found or the range becomes invalid.

### Source Code

#include <iostream>

using namespace std;

// Binary Search Function (Iterative)

int binarySearch(int arr[], int left, int right, int key) {

while (left <= right) {

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

// If key is found

if (arr[mid] == key)

return mid;

// If key is smaller, ignore right half

if (arr[mid] > key)

right = mid - 1;

else // If key is larger, ignore left half

left = mid + 1;

}

return -1; // If key is not found

}

// Driver Code

int main() {

int n, key;

// Taking input for array size

cout << "Enter number of elements: ";

cin >> n;

int arr[n];

// Taking sorted array as input

cout << "Enter sorted elements: ";

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

cin >> arr[i];

}

// Taking input for key to search

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

cin >> key;

int result = binarySearch(arr, 0, n - 1, key);

if (result != -1)

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

else

cout << "Element not found!" << endl;

return 0;

}

Input:

Enter number of elements: 7

Enter sorted elements: 2 3 4 10 18 20 25

Enter the element to search: 10

## Problem 5: Pattern Matching Algorithm

### Problem Statement

Write a program to find a given pattern from text using the pattern matching algorithm.

### Illustration of the Problem

Pattern matching algorithms like the Knuth-Morris-Pratt (KMP) algorithm efficiently locate substrings in text.

### Step-by-Step Algorithm

1.Preprocess the pattern to create a partial match table.

2.Compare the pattern with the text character by character.

3.Use the precomputed table to skip unnecessary comparisons.

4.Return the indices of occurrences.

Code:

#include <iostream>

using namespace std;

// Naïve Pattern Matching Algorithm

void patternSearch(string text, string pattern) {

int n = text.length();

int m = pattern.length();

bool found = false;

// Loop through the text

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

int j;

// Check if the pattern matches the substring of text

for (j = 0; j < m; j++) {

if (text[i + j] != pattern[j])

break;

}

// If pattern is found

if (j == m) {

cout << "Pattern found at index: " << i << endl;

found = true;

}

}

if (!found)

cout << "Pattern not found!" << endl;

}

// Driver Code

int main() {

string text, pattern;

// Taking user input

cout << "Enter the main text: ";

getline(cin, text);

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

getline(cin, pattern);

// Searching pattern in text

patternSearch(text, pattern);

return 0;

}

Input:

#### ****Input:****

Enter the main text: abcabcabc

Enter the pattern to search: abc

#### ****Output:****

Pattern found at index: 0

Pattern found at index: 3

Pattern found at index: 6

## ****Problem 6: Queue Data Structure Implementation****

### ****Problem Statement:****

Implement a queue data structure along with its typical operations.

### ****Solution Explanation:****

A queue follows the **FIFO (First In, First Out)** principle. The main operations are:

**Enqueue:** Insert an element at the rear.

**Dequeue:** Remove an element from the front.

**Front:** Get the front element without removing it.

**IsEmpty:** Check if the queue is empty.

### ****Algorithm:****

1.Initialize an empty queue.

2.Perform operations based on user input.

3.Maintain the front and rear pointers.

Source Code:

#include <iostream>

#define SIZE 5

using namespace std;

class Queue {

int items[SIZE], front, rear;

public:

Queue() { front = -1; rear = -1; }

void enqueue(int element);

int dequeue();

void display();

};

void Queue::enqueue(int element) {

if (rear == SIZE - 1) {

cout << "Queue is Full" << endl;

return;

}

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

items[++rear] = element;

}

int Queue::dequeue() {

if (front == -1 || front > rear) {

cout << "Queue is Empty" << endl;

return -1;

}

return items[front++];

}

void Queue::display() {

if (front == -1 || front > rear) {

cout << "Queue is Empty" << endl;

return;

}

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

cout << items[i] << " ";

cout << endl;

}

int main() {

Queue q;

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

q.display();

cout << "Dequeued: " << q.dequeue() << endl;

q.display();

return 0;

}.

### ****Sample Input & Output:****

Inserted: 10

Inserted: 20

Inserted: 30

Queue elements: 10 20 30

Removed: 10

Queue elements: 20 30

### ****Problem 7: N-Queens Problem Using Backtracking****

#### ****Problem Illustration & Logic:****

The N-Queens problem places N queens on an N×N chessboard such that no two queens attack each other. This is solved using **backtracking**:

1.Place a queen in a row.

2.Move to the next row and check for a safe position.

3.If a safe position is found, place the queen and repeat.

4.If no safe position is found, backtrack and try another placement.

#### ****Algorithm:****

1.Place the first queen.

2.Move to the next row and place the next queen in a valid column.

3.If a conflict arises, backtrack to the previous row.

4.Repeat until all N queens are placed.

Source Code:

#include <iostream>

using namespace std;

#define N 8

bool isSafe(int board[N][N], 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; j >= 0 && i < N; i++, j--)

if (board[i][j]) return false;

return true;

}

bool solveNQUtil(int board[N][N], int col) {

if (col >= N) return true;

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

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

board[i][col] = 1;

if (solveNQUtil(board, col + 1)) return true;

board[i][col] = 0;

}

}

return false;

}

void printSolution(int board[N][N]) {

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

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

cout << board[i][j] << " ";

cout << endl;

}

}

void solveNQ() {

int board[N][N] = {0};

if (!solveNQUtil(board, 0)) cout << "Solution does not exist";

else printSolution(board);

}

int main() {

solveNQ();

return 0;

}

#### ****Sample Output:****

0 0 1 0 0 0 0 0

1 0 0 0 0 0 0 0

0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 1

0 1 0 0 0 0 0 0

0 0 0 1 0 0 0 0

0 0 0 0 0 1 0 0

0 0 0 0 0 0 1 0

## ****Problem 8: Sum of Subset Problem****

**Problem Statement:**  
Given a set S={5,10,12,13,15,18}S = \{5,10,12,13,15,18\}S={5,10,12,13,15,18} and a target sum d=30d = 30d=30, write a program to find subsets that sum up to ddd.

### ****Explanation:****

The Sum of Subset problem is a classic backtracking problem where we try to generate all subsets and check if their sum equals the given target ddd.

### ****Algorithm (Backtracking Approach):****

1. Start with an empty subset.
2. Recursively explore subsets by including or excluding each element.
3. If the sum of a subset equals ddd, print it.
4. If the sum exceeds ddd, stop further exploration in that path.

### ****Python Code:****

python

CopyEdit

def subset\_sum(arr, target, subset=[], index=0):

if sum(subset) == target:

print(subset)

return

if sum(subset) > target or index >= len(arr):

return

# Include current element

subset\_sum(arr, target, subset + [arr[index]], index + 1)

# Exclude current element

subset\_sum(arr, target, subset, index + 1)

# Given set and target sum

S = [5, 10, 12, 13, 15, 18]

d = 30print("Subsets with sum 30:")

subset\_sum(S, d)

### ****Input****

Set: [5, 10, 12, 13, 15, 18]

Target sum: 30

### ****Output:****

Subsets with sum 30:

[5, 12, 13]

[12, 18]

[10, 12, 8]

## ****Problem 9: 0/1 Knapsack Problem (Dynamic Programming Approach)****

**Problem Statement:**  
Given:

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

Find the maximum profit that can be obtained using the **0/1 Knapsack Dynamic Programming Approach**.

### ****Explanation:****

The 0/1 Knapsack problem involves selecting items such that the total weight does not exceed the given capacity, and the profit is maximized. The **Dynamic Programming** approach builds a table to store optimal solutions for subproblems.

### ****Algorithm (Dynamic Programming Approach):****

1. Create a 2D table dp[i][w] where:
   * i represents the first i items.
   * w represents the weight limit.
2. If an item’s weight is less than or equal to w, decide whether to include it (maximizing profit).
3. Use the relation: dp[i][w]=max⁡(dp[i−1][w],P[i−1]+dp[i−1][w−W[i−1]])dp[i][w] = \max(dp[i-1][w], P[i-1] + dp[i-1][w-W[i-1]])dp[i][w]=max(dp[i−1][w],P[i−1]+dp[i−1][w−W[i−1]])
4. Return the value in dp[n][C] as the maximum profit.

### ****Python Code:****

python

CopyEdit

def knapsack(P, W, C, n):

dp = [[0 for \_ in range(C + 1)] for \_ in range(n + 1)]

for i in range(1, n + 1):

for w in range(1, C + 1):

if W[i - 1] <= w:

dp[i][w] = max(dp[i - 1][w], P[i - 1] + dp[i - 1][w - W[i - 1]])

else:

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

return dp[n][C]

# Given values

P = [15, 25, 13, 23]

W = [2, 6, 12, 9]

C = 20

n = 4

print("Maximum Profit:", knapsack(P, W, C, n)

### ****input:****

Profits: [15, 25, 13, 23]

Weights: [2, 6, 12, 9]

Knapsack Capacity: 20

Number of items: 4

### ****Output:****

Maximum Profit: 40

## ****Problem 10: Tower of Hanoi****

**Problem Statement:**  
Write a program to solve the Tower of Hanoi problem for NNN disks.

### ****Explanation:****

The Tower of Hanoi is a classic recursive problem where the goal is to move NNN disks from the **source rod** to the **destination rod**, using an **auxiliary rod**, while following these rules:

1. Only one disk can be moved at a time.
2. A larger disk cannot be placed on top of a smaller disk.
3. All disks start on the source rod and must end on the destination rod.

### ****Algorithm (Recursive Approach):****

1. Move N−1N-1N−1 disks from **source** to **auxiliary**.
2. Move the largest disk from **source** to **destination**.
3. Move N−1N-1N−1 disks from **auxiliary** to **destination**.

### ****Python Code:****

python

CopyEdit

def tower\_of\_hanoi(n, source, auxiliary, destination):

if n == 1:

print(f"Move disk 1 from {source} to {destination}")

return

tower\_of\_hanoi(n-1, source, destination, auxiliary)

print(f"Move disk {n} from {source} to {destination}")

tower\_of\_hanoi(n-1, auxiliary, source, destination)

# Number of disks

N = 3 # Change N for different casesprint(f"Tower of Hanoi solution for {N} disks:")

tower\_of\_hanoi(N, 'A', 'B', 'C')

### ****Input:****

Number of disks: 3

### ****Output:****

Tower of Hanoi solution for 3 disks:

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