

**LAB REPORT**

**Submitted By** :

Md. Shakib Hossain

Roll : 220637

Session : 2021-22

Dept. of Information and Communication Engineering

**ICE-2202  
Data Structure and Algorithm Sessional**

**Submitted To** :

Md. Anwar Hossain

Professor

Dept. of Information and Communication Engineering

**Submission Date : 01-03-2025**

Department of

**INFORMATION AND COMMUNICATION ENGINEERING**

**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**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. | |

**Problem 1: Sorting a Linear Array Using Bubble Sort Algorithm**

### Theory:

Bubble Sort is a straightforward sorting algorithm that iteratively examines adjacent elements in an array and swaps them if they are out of order. This process continues through multiple passes until no further swaps are required, signaling that the array is sorted. With each iteration, the largest unsorted element moves to its correct position, resembling a "bubble" floating to the top. Though simple to implement, Bubble Sort is inefficient for large datasets due to its **O(n²)** time complexity, making it most effective for small or nearly sorted arrays.

### Algorithm:

1. Begin with the first element in the array.
2. Compare it with the next element.
3. If the first element is larger than the next, swap them.
4. Move to the following element and repeat the comparison and swapping process for the remaining elements.
5. Continue this process for all elements until no swaps occur, ensuring the array is sorted.

**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 the elements

                int temp = arr[j];

                arr[j] = arr[j + 1];

                arr[j + 1] = temp;

            }

        }

    }

}

int main() {

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

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

    cout << "Original array:\n";

    for (int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

    bubbleSort(arr, n);

    cout << "Sorted array:\n";

    for (int i = 0; i < n; i++) {

        cout << arr[i] << " ";

    }

    cout << endl;

    return 0;

}

**Sample Input and Output**:

Original array:

64 34 25 12 22 11 90

Sorted array:

11 12 22 25 34 64 90

**Problem 2: Finding an Element Using Linear Search Algorithm**

### Theory:

Linear Search is a basic searching technique used to locate an element within an array or list. It operates by examining each element sequentially until the desired value is found or the entire array has been traversed. If the element is present, its index is returned; otherwise, the search is deemed unsuccessful.

Due to its **O(n)** time complexity in both average and worst cases, Linear Search is most suitable for small or unsorted datasets. It does not require extra memory, making it an in-place and easy-to-implement approach. However, for larger datasets, more efficient algorithms like Binary Search are preferred.

### Algorithm:

1. Start from the first element of the array.
2. Compare it with the target value.
3. If they match, return the index.
4. If they do not match, proceed to the next element.
5. Continue the process until the element is found or the array ends.

**Source Code:**

#include <iostream>

using namespace std;

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

    for (int i = 0; i < n; i++) {

        if (arr[i] == target) {

            return i;

        }

    }

    return -1;

}

int main() {

    int arr[] = {10, 20, 30, 40, 50};

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

    int target = 30;

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

    if (result == -1) {

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

    } else {

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

    }

    return 0;

}

**Sample Input and Output**:

Array: {10, 20, 30, 40, 50}

Target: 30

Element found at index: 2

**Problem 3: Sorting Using Merge Sort Algorithm**

## ****Theory:****

### Theory:

Merge Sort is a **divide-and-conquer** sorting algorithm that breaks an array into smaller subarrays, sorts them individually, and then merges them back together in order. The process starts by recursively splitting the array into halves until each subarray contains a single element, which is inherently sorted. These sorted subarrays are then combined step by step to form a completely sorted array.

With a **time complexity of O(n log n)** in all cases (best, average, and worst), Merge Sort is highly efficient for large datasets. It is a **stable sorting algorithm,** meaning it maintains the relative order of equal elements. However, it requires **O(n) additional space** for temporary arrays during the merging process. Merge Sort is especially useful for **linked lists and large datasets** where stability is crucial.

### Algorithm:

1. Split the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves to create a fully sorted array.

**Source Code:**

#include <iostream>

using namespace std;

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) {

        if (L[i] <= R[j]) {

            arr[k] = L[i];

            i++;

        } else {

            arr[k] = R[j];

            j++;

        }

        k++;

    }

    while (i < n1) {

        arr[k] = L[i];

        i++;

        k++;

    }

    while (j < n2) {

        arr[k] = R[j];

        j++;

        k++;

    }

}

void mergeSort(int arr[], int l, int r) {

    if (l >= r)

        return;

    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);

    mergeSort(arr, m + 1, r);

    merge(arr, l, m, r);

}

int main() {

    int arr[] = {12, 11, 13, 5, 6, 7};

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

    cout << "Given array is \n";

    for (int i = 0; i < n; i++)

        cout << arr[i] << " ";

    cout << endl;

    mergeSort(arr, 0, n - 1);

    cout << "Sorted array is \n";

    for (int i =

::contentReference[oaicite:0]{index=0}

**Sample Input and Output**:

Original array:

38 27 43 3 9 82 10

Sorted array:

3 9 10 27 38 43 82

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

## ****Theory:****

### Theory:

Binary Search is a highly efficient algorithm for finding a target element in a sorted array. The search begins by comparing the target value with the middle element of the array. If they match, the index is returned. Otherwise, if the target is smaller, the search continues in the left half, and if it's larger, the search proceeds in the right half. This process of reducing the search space by half continues until the element is found or the search range becomes empty.

With a time complexity of O(log n), Binary Search is much faster than Linear Search, especially for large datasets. However, its efficiency depends on the array being sorted beforehand, which may introduce additional overhead in certain applications.

### Algorithm:

1. Ensure the array is sorted.
2. Set low and high pointers to the start and end of the array.
3. Find the middle element.
4. If the middle element matches the target, return its index.
5. If the middle element is greater than the target, search in the left half.
6. If the middle element is smaller, search in the right half.
7. Repeat the process until the target is found or the search space is empty.

**Source Code:**

#include <iostream>

using namespace std;

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

    while (left <= right) {

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

        if (arr[mid] == key)

            return mid;

        if (arr[mid] < key)

            left = mid + 1;

        else

            right = mid - 1;

    }

    return -1;

}

int main() {

    int arr[] = {10, 20, 30, 40, 50};

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

    int key = 30;

    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;

}

### Sample Input and Output:

**Input:** [10, 20, 30, 40, 50], Key = 30

**Output:** Element found at index 2

### Problem 5: Write a program to find a given pattern from text using the pattern matching algorithm.

## ****Theory:****

## Pattern Matching Algorithms are designed to locate occurrences of a specific sequence, known as the pattern, within a larger sequence, known as the text. These algorithms are widely used in applications such as substring searching, DNA sequence analysis, and text editing tools.

Common pattern matching algorithms include:

1. **Naive String Matching** – This simple approach compares the pattern against every possible position in the text. Its time complexity is O(m \* n), where m is the pattern length and n is the text length.
2. **Knuth-Morris-Pratt (KMP) Algorithm** – This method preprocesses the pattern to construct a longest prefix-suffix (LPS) array, which helps in skipping unnecessary comparisons. It operates in O(n + m) time, making it more efficient for longer texts.
3. **Boyer-Moore Algorithm** – This algorithm uses precomputed pattern-based heuristics to skip sections of the text, making it highly efficient, especially for large alphabets. Its best-case time complexity is O(n/m), where m is the pattern length.

### Algorithm:

1. Iterate through the text from index 0 to (text length - pattern length).
2. At each position, compare the pattern with the corresponding substring.
3. If all characters match, record the position.
4. Continue the process until the end of the text is reached.

### C++ Code:

#include <iostream>

#include <string>

using namespace std;

void patternSearch(string text, string pattern) {

    int textLength = text.length();

    int patternLength = pattern.length();

    for (int i = 0; i <= textLength - patternLength; i++) {

        int j;

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

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

                break;

            }

        }

        if (j == patternLength) {

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

        }

    }

}

int main() {

    string text = "ababcabcabababd";

    string pattern = "ababd";

    patternSearch(text, pattern);

    return 0;

}

### Sample Input and Output:

**Input:** Text = "ababcabcabababd", Pattern = "ababd"

**Output:** Pattern found at index 10

### Problem 6: Write a program to implement a queue data structure along with its typical operations.

## ****Theory:****

## A queue is a linear data structure that operates on the ****First-In-First-Out (FIFO)**** principle, meaning elements are inserted at the rear and removed from the front. This ordering makes queues useful in scenarios where maintaining sequence is important, such as task scheduling, print job management, and request handling in web applications.

A queue typically supports several key operations:

* **Enqueue** – Adds an element to the rear of the queue.
* **Dequeue** – Removes and returns the front element.
* **Peek (Front)** – Retrieves the front element without removing it.
* **IsEmpty** – Checks if the queue is empty.
* **Size** – Returns the number of elements in the queue.

Queues can be implemented using arrays or linked lists, with each approach having trade-offs in terms of space efficiency and operation time complexity. Due to their simplicity and efficiency in managing ordered collections, queues are widely used in computer science and programming.

### Algorithm:

1. Initialize an empty queue.
2. Implement enqueue to insert elements at the rear.
3. Implement dequeue to remove elements from the front.
4. Implement a display function to show queue elements.
5. Handle boundary conditions such as underflow and overflow.

### C++ Code:

#include <iostream>

using namespace std;

#define MAX 5

int queue[MAX], front = -1, rear = -1;

int main() {

    int choice, value;

    while (1) {

        cout << "\nQueue Operations:\n";

        cout << "1. Enqueue\n";

        cout << "2. Dequeue\n";

        cout << "3. Display\n";

        cout << "4. Exit\n";

        cout << "Enter your choice: ";

        cin >> choice;

        switch (choice) {

            case 1:

                if (rear == MAX - 1) {

                    cout << "Queue is full! Cannot enqueue.\n";

                } else {

                    if (front == -1) {

                        front = 0;

                    }

                    cout << "Enter the value to enqueue: ";

                    cin >> value;

                    rear++;

                    queue[rear] = value;

                    cout << value << " enqueued to the queue.\n";

                    cout << "Current front: " << front << ", rear: " << rear << endl;

                }

                break;

            case 2:

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

                    cout << "Queue is empty! Cannot dequeue.\n";

                } else {

                    cout << "Dequeued value: " << queue[front] << endl;

                    front++;

                    cout << "Current front: " << front << ", rear: " << rear << endl;

                    if (front > rear) {

                        front = rear = -1;

                        cout << "Queue is now empty.\n";

                    }

                }

                break;

            case 3:

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

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

                } else {

                    cout << "Queue elements are: ";

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

                        cout << queue[i] << " ";

                    }

                    cout << endl;

                }

                break;

            case 4:

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

                return 0;

            default:

                cout << "Invalid choice! Try again.\n";

        }

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

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

        } else {

            cout << "Current queue: ";

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

                cout << queue[i] << " ";

            }

            cout << endl;

        }

    }

    return 0;

}

### Sample Input and Output:

Output :

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the value to enqueue: 3

3 enqueued to the queue.

Current front: 0, rear: 0

Current queue: 3

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 1

Enter the value to enqueue: 4

4 enqueued to the queue.

Current front: 0, rear: 1

Current queue: 3 4

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 3

Queue elements are: 3 4

Current queue: 3 4

Queue Operations:

1. Enqueue

2. Dequeue

3. Display

4. Exit

Enter your choice: 4

Exiting...

### Problem7: Write a program to solve the n-queens problem using backtracking.

## ****Theory:****

## The N-Queens problem is a well-known combinatorial challenge that requires placing **N queens** on an **N×N chessboard** so that no two queens attack each other. This means that no two queens can be positioned in the same row, column, or diagonal.

A **backtracking algorithm** is an effective way to solve this problem. It works by incrementally placing queens and abandoning placements that lead to conflicts. The process begins by placing a queen in the first row and recursively attempting to position queens in the following rows while ensuring no conflicts with already placed queens. If a placement results in a valid solution, it is recorded. If a conflict arises, the algorithm backtracks by removing the last placed queen and trying the next possible position.

This backtracking approach significantly reduces the number of possible placements by eliminating invalid positions early, making it an efficient method for solving the N-Queens problem.

### Algorithm:

1. Place queens **row by row**.
2. Check if a queen can be placed in a particular **column** of the current row without being attacked.
3. If the placement is **safe**, place the queen and proceed to the next row.
4. If all queens are successfully placed, **record the solution**.
5. If a conflict occurs, **backtrack** by removing the last placed queen and trying the next available position.
6. Repeat the process until all possible solutions are found.

### C++ Code:

#include <iostream>

using namespace std;

#define N 8

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;

    }

    cout << endl;

}

bool isSafe(int board[N][N], int row, int col) {

    for (int i = 0; i < row; i++)

        if (board[i][col]) 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 >= 0 && j < N; i--, j++)

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

    return true;

}

bool solveNQueens(int board[N][N], int row) {

    if (row >= N) {

        printSolution(board);

        return true;

    }

    bool res = false;

    for (int i = 0; i < N; i++) {

        if (isSafe(board, row, i)) {

            board[row][i] = 1;

            res = solveNQueens(board, row + 1) || res;

            board[row][i] = 0;

        }

    }

    return res;

}

int main() {

    int board[N][N] = {0};

    if (!solveNQueens(board, 0))

        cout << "No solution exists" << endl;

    return 0;

}

### Sample Input and Output:

**Input:** N = 8

**Output:**

1 0 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 0 1

0 0 1 0 0 0 0 0

0 0 0 0 1 0 0 0

0 1 0 0 0 0 0 0

0 0 0 0 0 0 1 0

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

## ****Theory:****

The Sum of Subset Problem is a combinatorial optimization problem that involves determining whether a subset of a given set of integers adds up to a specific target value. Given a set S = {5, 10, 12, 13, 15, 18} and a target sum d = 30, the goal is to find a combination of elements in S that sum exactly to d.

A common approach to solving this problem is recursive backtracking, where elements are either included in or excluded from the subset as the algorithm explores possible combinations. If the current sum equals the target, a valid subset is found. If the sum exceeds the target, that path is abandoned, and the algorithm backtracks to explore other possibilities.

This problem has significant applications in resource allocation, cryptography, and decision-making processes. For larger datasets, dynamic programming can be used to improve efficiency by avoiding redundant calculations and optimizing the search process.

### Algorithm:

1. Start with an empty subset and iterate through elements of the given set.
2. Include an element in the subset and check if the sum matches the target.
3. If the sum equals the target, record the subset.
4. If the sum exceeds the target, backtrack by removing the last included element.
5. Continue the process until all possible subsets have been explored.

### C++ Code:

#include <iostream>

#include <vector>

using namespace std;

void printSubset(vector<int>& subset) {

    cout << "{ ";

    for (int num : subset) cout << num << " ";

    cout << "}" << endl;

}

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

    if (sum == target) {

        printSubset(subset);

        return;

    }

    if (index >= set.size() || sum > target) return;

    subset.push\_back(set[index]);

    findSubsets(set, subset, index + 1, sum + set[index], target);

    subset.pop\_back();

    findSubsets(set, subset, index + 1, sum, target);

}

int main() {

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

    int target = 30;

    vector<int> subset;

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

    findSubsets(set, subset, 0, 0, target);

    return 0;

}

### Sample Input and Output:

**Input:**

Set: {5, 10, 12, 13, 15, 18}

Target Sum: 30

**Output:**

{ 5 10 15 }

{ 5 12 13 }

{ 12 18 }

### Problem 9 :Write a program to solve the following 0/1 Knapsack problem using dynamic programming. Given:

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

## ****Theory:****

The 0/1 Knapsack Problem is an optimization problem that involves selecting items with given weights and profits to maximize the total profit without exceeding a specified capacity. Given profits P = (15, 25, 13, 23), weights W = (2, 6, 12, 9), and a knapsack capacity C = 20 with n = 4 items, the objective is to determine the maximum profit achievable by either including or excluding each item. Each item can either be taken (1) or left (0), hence the name "0/1 Knapsack."

Dynamic programming is an efficient approach to solving this problem. A 2D table is constructed where the rows represent items and the columns represent possible weights from 0 to C. The table is filled iteratively, considering whether including an item results in a greater profit than excluding it. This method avoids the exponential time complexity of a brute-force approach, leading to a more efficient solution with a time complexity of O(nC). The maximum profit is found in the last cell of the table, which represents the optimal solution for the given capacity.

### Algorithm:

1. Create a 2D DP table of size (n+1) x (C+1).
2. Initialize the first row and column to zero.
3. For each item, decide whether to include it or not based on the maximum profit obtained.
4. Return the maximum profit found in dp[n][C].

### C++ Code:

#include <iostream>

using namespace std;

int knapsack(int W[], int P[], int C, int n) {

    int dp[n+1][C+1];

    for (int i = 0; i <= n; i++) {

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

            if (i == 0 || w == 0)

                dp[i][w] = 0;

            else if (W[i-1] <= w)

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

            else

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

        }

    }

    return dp[n][C];

}

int main() {

    int P[] = {15, 25, 13, 23};

    int W[] = {2, 6, 12, 9};

    int C = 20;

    int n = sizeof(P)/sizeof(P[0]);

    cout << "Maximum profit: " << knapsack(W, P, C, n) << endl;

    return 0;

}

### Sample Input and Output:

### Maximum profit: 40

**Problem 10:Write a program to slove the Tower of Hanoi problem for the N disk**

## ****Theory:****

The Tower of Hanoi is a well-known mathematical puzzle that involves transferring a stack of disks from one rod to another while following specific rules. The puzzle consists of three rods and multiple disks of different sizes that can be moved between the rods. The goal is to transfer the entire stack from the source rod to the destination rod while following these constraints:

* Only one disk can be moved at a time.
* A disk can only be moved if it is the topmost disk on a rod.
* No larger disk can be placed on top of a smaller disk.

The solution to this problem is based on recursion. The process for **n** disks involves:

1. Moving **n-1** disks from the source rod to an auxiliary rod using the destination rod.
2. Moving the **nth** (largest) disk directly to the destination rod.
3. Moving the **n-1** disks from the auxiliary rod to the destination rod using the source rod.

The minimum number of moves required to solve the puzzle is given by the formula **2ⁿ - 1**. The Tower of Hanoi demonstrates the principles of recursion and is often used to illustrate algorithmic problem-solving techniques.

### Algorithm:

1. Move **N-1** disks from the source rod to the auxiliary rod using the destination rod.
2. Move the **Nth** disk directly from the source rod to the destination rod.
3. Move the **N-1** disks from the auxiliary rod to the destination rod using the source rod.

### ****C++ Code:****

#include <iostream>

using namespace std;

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

    if (n == 1) {

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

        return;

    }

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

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

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

}

int main() {

    int N = 3; // Number of disks

    towerOfHanoi(N, 'A', 'B', 'C');

    return 0;

}

### ****Sample Input and Output:****

### 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