

Department of

**INFORMATION AND COMMUNICATION ENGINEERING**

**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**LAB REPORT**

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

**Submitted To** :

Md. Anwar Hossain

Professor

Dept. of Information and Communication Engineering

**Submitted By** :

Md. Sabbir Hossain

Roll : 220603

Session : 2021-22

Dept. of Information and Communication Engineering

**Submission Date : 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. | |

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

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

Bubble Sort is a simple sorting algorithm that repeatedly compares adjacent elements in a linear array and swaps them if they are in the wrong order. This process is repeated for multiple passes until no swaps are needed, indicating that the array is sorted. With each pass, the largest unsorted element moves to its correct position, similar to a "bubble" rising to the surface. While easy to implement, Bubble Sort is inefficient for large datasets due to its **O(n²)** time complexity, making it suitable mainly for small or nearly sorted arrays.

**Algorithm:**

1. Start from the first element of the array.
2. Compare the current element with the next element.
3. If the current element is greater than the next element, swap them.
4. Move to the next element and repeat steps 2-3 for the remaining elements.
5. Repeat the process for all elements until no swaps are needed.

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 simple searching algorithm used to find an element in an array or list. It works by sequentially checking each element of the array until the target element is found or the end of the array is reached. If the element is found, its index is returned; otherwise, the search is unsuccessful.

This algorithm is best suited for **small or unsorted datasets** due to its **O(n)** time complexity in both the worst and average cases. It requires no additional memory, making it an **in-place and straightforward** search method. However, for large datasets, more efficient algorithms like Binary Search are preferred.

**Algorithm:**

1. Start from the first element.
2. Compare the element with the target value.
3. If it matches, return the index.
4. If it does not match, move to the next element.
5. Repeat until the target is found or the list 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:****

Merge Sort is a divide-and-conquer sorting algorithm that divides an array into smaller subarrays, sorts them, and then merges them back together. The process involves recursively splitting the array in half until each subarray contains a single element, which is inherently sorted. Then, these sorted subarrays are merged to produce a fully sorted array.

Merge Sort has a time complexity of O(n log n) in all cases (best, average, and worst), making it efficient for large datasets. It is a stable sorting algorithm, preserving the relative order of equal elements, and operates in O(n) space due to the need for temporary arrays during the merge process. Merge Sort is particularly effective for linked lists and large datasets where stability is important.

**Algorithm:**

1. Divide the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves into a single 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:****

Binary Search is an efficient algorithm for locating a target element within a sorted array. The algorithm begins by comparing the target value to the middle element of the array. If the target matches the middle element, its index is returned. If the target is less than the middle element, the search continues in the lower half of the array; if it is greater, the search continues in the upper half. This process of halving the search interval continues until the target element is found or the interval is empty. The efficiency of Binary Search comes from its **O(log n)** time complexity, making it significantly faster than linear search for large datasets. However, it requires that the array be sorted beforehand, which can impact its overall performance in certain scenarios.

**Algorithm:**

1. Sort the array.
2. Set low and high pointers 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 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 until the element 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 used to find occurrences of a specific sequence (the pattern) within a larger sequence (the text). These algorithms play a crucial role in various applications, such as searching for substrings in strings, DNA sequence analysis, and text editing tools.

The most commonly used pattern matching algorithms include:

1. **Naive String Matching:** This straightforward method checks for the presence of a pattern in the text by comparing each possible position of the pattern against the text. Its time complexity is **O(m \* n)**, where **m** is the length of the pattern and **n** is the length of the text.
2. **Knuth-Morris-Pratt (KMP) Algorithm:** This more efficient algorithm preprocesses the pattern to create a longest prefix-suffix (LPS) array that allows the search to skip unnecessary comparisons. KMP has a time complexity of **O(n + m)**, making it much faster for longer texts and patterns.
3. **Boyer-Moore Algorithm:** This algorithm utilizes information from the pattern itself to skip sections of the text, making it particularly efficient for large alphabets. It has a best-case time complexity of **O(n/m)**, where **m** is the length of the pattern.

.

### Algorithm:

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

### 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 follows the First-In-First-Out (FIFO) principle, where elements are added at the rear and removed from the front. This characteristic makes queues ideal for scenarios where order needs to be preserved, such as in scheduling tasks, managing print jobs, or handling requests in web applications. A typical queue supports several fundamental operations: **Enqueue**, which adds an element to the end of the queue; **Dequeue**, which removes and returns the front element; **Peek** (or Front), which allows access to the front element without removing it; **IsEmpty**, which checks if the queue is empty; and **Size**, which returns the number of elements currently in the queue. Queues can be implemented using arrays or linked lists, with each implementation offering different performance characteristics, particularly in terms of space efficiency and operation time complexity. Overall, queues are essential data structures widely used in computer science and programming for their straightforward and effective management of ordered collections of items.

### Algorithm:

1. Initialize an empty queue.
2. Implement enqueue (insertion at rear).
3. Implement dequeue (removal from front).
4. Implement display function to show elements.
5. Ensure boundary conditions like underflow and overflow are handled.

### 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 classic combinatorial problem that involves placing N queens on an N×N chessboard so that no two queens threaten each other. This means that no two queens can share the same row, column, or diagonal. The backtracking algorithm is an effective approach to solve this problem by incrementally building potential solutions and abandoning those that fail to satisfy the constraints. The algorithm starts by placing a queen in the first column of the first row and recursively attempts to place queens in subsequent rows while ensuring that the placements do not conflict with already placed queens. If a placement leads to a solution, it is recorded; if not, the algorithm backtracks by removing the queen and trying the next possible position. This process continues until all solutions are found or all possibilities are exhausted. The backtracking method is efficient for the N-Queens problem, as it significantly reduces the search space by eliminating invalid placements early in the process.

### Algorithm:

1. Start placing 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 safe, place the queen and move to the next row.
4. If all queens are placed successfully, print the board.
5. If not, backtrack by removing the last placed queen and trying the next possible position.
6. Repeat until all 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 there exists a subset of a given set of integers that sums to a specific target value, denoted as ddd. In this context, 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, the challenge is to find if a combination of the elements in SSS can yield this exact sum. The problem can be approached using recursive backtracking, where the algorithm explores the inclusion or exclusion of each element in the subset. If the current sum equals ddd, a valid subset has been found; if the current sum exceeds ddd, that path is abandoned. This method systematically searches through all possible subsets, making it a fundamental problem in computer science with applications in resource allocation, cryptography, and decision-making processes. The efficiency of the solution can be improved using dynamic programming techniques, particularly for larger sets or target sums, allowing for more effective exploration of possible combinations.

### 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 is equal to the target, print the subset.
4. If the sum exceeds the target, backtrack and remove the last included element.
5. Continue until all subsets are checked.

### 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 a classic optimization problem that involves selecting items with given weights and profits to maximize total profit without exceeding a specified capacity, known as the knapsack capacity. In this case, we are 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), and a knapsack capacity C=20C = 20C=20 with n=4n = 4n=4 items. The challenge is to determine the maximum profit achievable by including or excluding each item, where each item can either be taken (1) or left (0), hence the name "0/1 Knapsack."

Dynamic programming is used to solve this problem efficiently by constructing a 2D table where the rows represent the items and the columns represent possible weights from 0 to CCC. The table is filled iteratively, considering for each item whether it should be included in the knapsack or not, based on whether it offers a greater profit than not including it. This approach avoids the exponential time complexity of a brute-force solution, leading to a more efficient solution with a time complexity of O(nC)O(nC)O(nC). The final solution can be found in the last cell of the table, representing the maximum profit obtainable with 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 classic mathematical puzzle that involves moving a stack of disks from one rod to another, following specific rules. The puzzle consists of three rods and a number of disks of different sizes that can slide onto any rod. The objective is to move the entire stack to another rod, adhering to the following constraints: only one disk can be moved at a time, each move consists of taking the upper disk from one of the stacks and placing it on top of another stack, and no larger disk may be placed on top of a smaller disk.

The solution to the Tower of Hanoi puzzle can be achieved through recursion, where the problem is broken down into smaller subproblems. For nnn disks, the process involves recursively moving n−1n-1n−1 disks to an auxiliary rod, then moving the largest disk directly to the target rod, and finally moving the n−1n-1n−1 disks from the auxiliary rod to the target rod. The minimum number of moves required to solve the puzzle is given by the formula 2n−12^n - 12n−1. This elegant problem not only illustrates the principles of recursion and algorithm design but also serves as a foundation for understanding more complex computational theories.

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

1. Move N-1 disks from Source to Auxiliary using Destination.
2. Move the Nth disk directly from Source to Destination.
3. Move the N-1 disks from Auxiliary to Destination using Source.

### ****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