

**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**

Department of

**INFORMATION AND COMMUNICATION ENGINEERING**

**LAB REPORT**

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

**Submitted By** :

MD Sazzad Hossain Shakkhor

Roll : 220625

Session : 2021-22

Dept. of Information and Communication Engineering

**Submitted To** :

Md. Anwar Hossain

Professor

Dept. of Information and Communication Engineering

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

**Course Code : ICE-2202**

**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 basic comparison-based 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 entire array is sorted. After each complete pass, the largest unsorted element "bubbles up" to its correct position, reducing the number of elements to be checked in subsequent passes.  
Although easy to understand and implement, Bubble Sort is inefficient for large datasets due to its **O(n²) time complexity**, making it impractical for extensive sorting tasks. It performs best when the array is already nearly sorted.

**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 one of the simplest searching techniques, used to locate an element in a list or array. The algorithm checks each element sequentially, one by one, until it finds the target or reaches the end of the list.  
Although easy to implement, Linear Search has an **O(n) time complexity**, making it inefficient for large datasets. It works well for small or unsorted datasets but is outperformed by more advanced search methods like Binary Search when dealing with sorted arrays.

**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 an efficient sorting algorithm that follows the **divide-and-conquer** approach. The algorithm recursively divides the array into two halves until each subarray contains a single element. Then, these subarrays are merged back in sorted order to produce a fully sorted array.  
Merge Sort consistently runs in **O(n log n) time complexity**, making it significantly faster than Bubble Sort for large datasets. However, it requires **extra space** (O(n)) to store temporary arrays during the merging process. This makes it less efficient in terms of space compared to in-place sorting algorithms like Quick Sort.

**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 a fast and efficient searching algorithm used to locate an element in a sorted array. The algorithm works by repeatedly dividing the search space in half. It starts by comparing the middle element of the array with the target. If the middle element matches the target, its index is returned. If the target is smaller, the search continues in the left half; otherwise, it proceeds in the right half. This process continues until the target is found or the search space becomes empty. The key advantage of Binary Search is its O(log n) time complexity, making it significantly more efficient than Linear Search for large datasets. However, its primary limitation is that the array must be sorted before performing the search.

**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.

Pattern Matching is the process of finding occurrences of a specific sequence of characters (the pattern) within a larger text. This is widely used in applications such as text searching, plagiarism detection, and DNA sequence analysis. Several algorithms are used for pattern matching:

* Naive String Matching: It checks every possible position in the text by comparing the pattern character by character. Its time complexity is O(m \* n), where m is the pattern length and n is the text length.
* Knuth-Morris-Pratt (KMP) Algorithm: It preprocesses the pattern to create a Longest Prefix Suffix (LPS) array, allowing the search to skip unnecessary comparisons. Its time complexity is O(n + m).
* Boyer-Moore Algorithm: This method skips sections of the text by utilizing information from the pattern itself, making it highly efficient, especially for large texts.

### 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. Elements are inserted at the **rear** and removed from the **front**. Queues are widely used in scheduling, buffering, and managing shared resources. The primary operations of a queue include:

* **Enqueue:** Adding an element to the rear.
* **Dequeue:** Removing an element from the front.
* **Peek (Front):** Viewing the front element without removing it.
* **IsEmpty:** Checking if the queue is empty.
* **Size:** Getting the number of elements in the queue.

Queues can be implemented using **arrays** or **linked lists**. The **array-based implementation** has a fixed size, whereas the **linked-list implementation** allows dynamic resizing.

### 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 constraint satisfaction problem where the goal is to place N queens on an N×N chessboard so that no two queens attack each other. This means that no two queens should be in the same row, column, or diagonal.

A backtracking algorithm is an efficient way to solve this problem by placing queens row by row and ensuring that each placement follows the constraints. If a queen cannot be placed in a row without conflict, the algorithm backtracks and tries a different column in the previous row. This method significantly reduces the search space, making it an effective approach.

### 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 involves determining whether a subset of a given set of numbers sums to a specific target value. This problem can be solved using backtracking, which explores all possible subsets by including or excluding elements recursively. If a valid subset is found, it is printed; if the sum exceeds the target, the algorithm backtracks. This approach is useful in decision-making applications such as resource allocation.

### 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 an optimization problem where we aim to maximize profit while ensuring the total weight does not exceed a given capacity. Given a set of items, each with a specific weight and profit, we must decide whether to include each item in the knapsack. This problem can be effectively solved using dynamic programming by constructing a table that records the maximum profit for varying capacities. The solution iteratively determines the best combination by considering whether including an item yields a higher profit than excluding it. The final answer is obtained from the last cell of the table.

### 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 set of disks from one rod to another while following specific rules. The puzzle consists of three rods and multiple disks of varying sizes, which are initially stacked in decreasing order on one rod. The objective is to move the entire stack to a different rod while adhering to the following constraints:

1. Only one disk can be moved at a time.
2. A disk can only be placed on an empty rod or on top of a larger disk.
3. A larger disk cannot be placed on a smaller disk.

The puzzle is typically solved using recursion, where the problem is divided into smaller subproblems. For N disks, the recursive approach follows these steps:

1. Move the top (N-1) disks from the Source rod to an Auxiliary rod, using the Destination rod as a temporary holder.
2. Move the Nth (largest) disk directly to the Destination rod.
3. Transfer the N-1 disks from the Auxiliary rod to the Destination rod, using the Source rod as a helper.

The minimum number of moves required to solve the Tower of Hanoi puzzle for N disks is given by the formula:

Total moves=2N−1\text{Total moves} = 2^N - 1

This problem serves as a classic example of recursion and algorithm design and is widely used in computer science to illustrate the principles of divide and conquer strategies. It also has applications in areas such as data structure manipulation, algorithm optimization, and computational problem-solving.

### ****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