 **LAB REPORT**

Department of

**INFORMATION AND COMMUNICATION ENGINEERING**

**PABNA UNIVERSITY OF SCIENCE AND TECHNOLOGY**

**Submitted To** :

Md. Anwar Hossain

Professor

Dept. of Information and Communication Engineering

**Submitted By** :

Al Kavy Joy

Roll : 220627

Session : 2021-22

Dept. of Information and Communication Engineering

**ICE-2202**

**Data Structure and Algorithm Sessional**

**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. | |

**Lab Report 01: Bubble Sort Algorithm**

**1. Theory** Bubble Sort is a simple sorting algorithm that works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. The process is repeated until the list is sorted. This algorithm has a time complexity of O(n^2) in the worst and average cases, making it inefficient for large datasets. However, it is easy to understand and implement.

**2. Algorithm**

1. Start at the beginning of the array.
2. Compare the first two elements:If the first element is greater than the second, swap them.
3. Move to the next pair of adjacent elements and repeat step 2.
4. Continue this process for the entire array.
5. Repeat the entire process for (n-1) passes until the array is sorted.
6. If no swaps occur in a pass, the array is already sorted, and the algorithm terminates early.

**3. Source Code**

# Bubble Sort Implementation in Python

def bubble\_sort(arr):

    n = len(arr)

    for i in range(n - 1):

        swapped = False

        for j in range(n - 1 - i):

            if arr[j] > arr[j + 1]:

                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # Swap elements

                swapped = True

        if not swapped:

            break  # Optimization: Stop if no swaps were made

# Example usage:

arr = [64, 34, 25, 12, 22, 11, 90]

print("Original array:", arr)

bubble\_sort(arr)

print("Sorted array:", arr)

**Input**

Original array: [64, 34, 25, 12, 22, 11, 90]

**5. Output**

Sorted array: [11, 12, 22, 25, 34, 64, 90]

**Lab Report 02: Linear Search Algorithm**

**1. Theory** Linear Search is a simple searching algorithm used to find the position of a target element in an array. It works by iterating through the list sequentially and comparing each element with the target value. If a match is found, the index of the element is returned; otherwise, the search continues until the end of the list. This algorithm has a time complexity of O(n) in the worst case, making it inefficient for large datasets but useful for small or unsorted lists.

**2. Algorithm**

1. Start from the first element of the array.
2. Compare the current element with the target value.
3. If they match, return the index of the element.
4. If they do not match, move to the next element.
5. Repeat steps 2-4 until the end of the array is reached.
6. If the target value is not found, return -1 to indicate absence.

**3. Source Code**

# Linear Search Implementation in Python

def linear\_search(arr, target):

    for i in range(len(arr)):

        if arr[i] == target:

            return i  # Return index of the target element

    return -1  # Return -1 if not found

# Example usage:

arr = [10, 20, 30, 40, 50]

target = 30

result = linear\_search(arr, target)

if result != -1:

    print(f"Element found at index {result}")

else:

    print("Element not found")

**4. Input**

Array: [10, 20, 30, 40, 50]

Target element: 30

**5. Output**

Element found at index 2

**Lab Report 03: Merge Sort Algorithm**

**1. Theory** Merge Sort is a divide-and-conquer sorting algorithm that splits an array into smaller subarrays, sorts them, and then merges them back together. The process involves recursively dividing the array into two halves until each subarray contains only a single element. Then, the merging step combines sorted subarrays to form a fully sorted array. Merge Sort has a time complexity of O(n log n) in all cases, making it more efficient than Bubble Sort for large datasets.

**2. Algorithm**

1. If the array has only one element or is empty, return it as it is already sorted.
2. Divide the array into two halves.
3. Recursively apply Merge Sort to both halves.
4. Merge the sorted halves back together.
5. Return the sorted array.

**3. Source Code**

# Merge Sort Implementation in Python

def merge\_sort(arr):

    if len(arr) <= 1:

        return arr

    mid = len(arr) // 2  # Find the middle of the array

    left\_half = merge\_sort(arr[:mid])

    right\_half = merge\_sort(arr[mid:])

    return merge(left\_half, right\_half)

def merge(left, right):

    sorted\_array = []

    i = j = 0

    while i < len(left) and j < len(right):

        if left[i] < right[j]:

            sorted\_array.append(left[i])

            i += 1

        else:

            sorted\_array.append(right[j])

            j += 1

    sorted\_array.extend(left[i:])

    sorted\_array.extend(right[j:])

    return sorted\_array

# Example usage:

arr = [38, 27, 43, 3, 9, 82, 10]

print("Original array:", arr)

sorted\_arr = merge\_sort(arr)

print("Sorted array:", sorted\_arr)

**4. Input**

Original array: [38, 27, 43, 3, 9, 82, 10]

**5. Output**

Sorted array: [3, 9, 10, 27, 38, 43, 82]

**Lab Report 04: Binary Search Algorithm**

**1. Theory** Binary Search is an efficient searching algorithm that finds the position of a target element in a sorted array. It follows the divide-and-conquer approach by repeatedly dividing the search interval in half. If the target value is smaller than the middle element, the search continues in the left half; otherwise, it continues in the right half. This process repeats until the target value is found or the search interval is empty. The time complexity of Binary Search is O(log n), making it significantly faster than Linear Search for large datasets.

**2. Algorithm**

1. Start with a sorted array.
2. Set two pointers: low at the beginning and high at the end of the array.
3. Compute the middle index mid = (low + high) // 2.
4. Compare the middle element with the target value:
   * If equal, return the index.
   * If smaller, update low = mid + 1.
   * If larger, update high = mid - 1.
5. Repeat the process until low exceeds high.
6. If the target value is not found, return -1.

**3. Source Code**

# Binary Search Implementation in Python

def binary\_search(arr, target):

    low, high = 0, len(arr) - 1

    while low <= high:

        mid = (low + high) // 2

        if arr[mid] == target:

            return mid  # Target found

        elif arr[mid] < target:

            low = mid + 1  # Search in the right half

        else:

            high = mid - 1  # Search in the left half

    return -1  # Target not found

# Example usage:

arr = [10, 20, 30, 40, 50, 60, 70]

target = 40

result = binary\_search(arr, target)

if result != -1:

    print(f"Element found at index {result}")

else:

    print("Element not found")

**4. Input**

Array: [10, 20, 30, 40, 50, 60, 70]

Target element: 40

**5. Output**

Element found at index 3

**Lab Report 05: Pattern Matching Algorithm**

**1. Theory** Pattern Matching is a technique used to find occurrences of a given pattern within a larger text. It is widely used in text processing, search engines, and data validation. The simplest approach, known as the **Brute Force Algorithm**, involves checking the pattern at every possible position in the text. More advanced techniques include the **Knuth-Morris-Pratt (KMP) Algorithm** and the **Boyer-Moore Algorithm**, which optimize the search by reducing redundant comparisons.

**2. Algorithm (Brute Force Pattern Matching)**

1. Start from the beginning of the text and slide the pattern over it one character at a time.
2. Compare each character of the pattern with the corresponding text characters.
3. If all characters match, return the starting index of the match.
4. If a mismatch occurs, shift the pattern by one position and repeat the comparison.
5. Continue until the end of the text is reached.
6. If no match is found, return -1.

**3. Source Code**

# Pattern Matching Implementation in Python (Brute Force Algorithm)

def pattern\_matching(text, pattern):

    n, m = len(text), len(pattern)

    for i in range(n - m + 1):

        match = True

        for j in range(m):

            if text[i + j] != pattern[j]:

                match = False

                break

        if match:

            return i  # Pattern found at index i

    return -1  # Pattern not found

# Example usage:

text = "this is a simple text"

pattern = "simple"

result = pattern\_matching(text, pattern)

if result != -1:

    print(f"Pattern found at index {result}")

else:

    print("Pattern not found")

**4. Input**

Text: "this is a simple text"

Pattern: "simple"

**5. Output**

Pattern found at index 10

**Lab Report 06: Queue Data Structure Implementation**

**1. Theory** A queue is a linear data structure that follows the **First-In-First-Out (FIFO)** principle, meaning the element inserted first is removed first. Queues are widely used in scheduling processes, handling requests in web servers, and managing resources in operating systems.

There are two main types of queues:

* **Simple Queue**: Elements are inserted at the rear and removed from the front.
* **Circular Queue**: The last position is connected to the first to utilize space efficiently.
* **Priority Queue**: Elements are dequeued based on priority instead of order.
* **Double-ended Queue (Deque)**: Elements can be added or removed from both ends.

**2. Algorithm (Simple Queue Operations)**

1. **Enqueue (Insert an element)**:
   * Check if the queue is full.
   * Insert the element at the rear.
   * Increment the rear pointer.
2. **Dequeue (Remove an element)**:
   * Check if the queue is empty.
   * Remove the element from the front.
   * Increment the front pointer.
3. **Peek (Retrieve front element without removing it)**.
4. **Display (Show all elements in the queue)**.

**3. Source Code**

# Queue Implementation in Python

class Queue:

    def \_\_init\_\_(self, size):

        self.queue = []

        self.size = size

    def enqueue(self, item):

        if len(self.queue) < self.size:

            self.queue.append(item)

            print(f"Enqueued: {item}")

        else:

            print("Queue is full")

    def dequeue(self):

        if self.queue:

            removed = self.queue.pop(0)

            print(f"Dequeued: {removed}")

            return removed

        else:

            print("Queue is empty")

            return None

    def peek(self):

        if self.queue:

            return self.queue[0]

        else:

            print("Queue is empty")

            return None

    def display(self):

        print("Queue:", self.queue)

# Example usage:

q = Queue(5)

q.enqueue(10)

q.enqueue(20)

q.enqueue(30)

q.display()

q.dequeue()

q.display()

**4. Input**

Enqueue: 10, 20, 30

Display queue

Dequeue an element

Display queue

**5. Output**

Enqueued: 10

Enqueued: 20

Enqueued: 30

Queue: [10, 20, 30]

Dequeued: 10

Queue: [20, 30]

**Lab Report 07: N-Queens Problem Using Backtracking**

**1. Theory** The N-Queens problem is a classic combinatorial problem 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 in the same row, column, or diagonal. The problem is commonly solved using the backtracking technique, which systematically explores all possible placements and backtracks when a conflict is found.

Backtracking is a depth-first search approach that incrementally builds a solution and abandons a path as soon as it determines that it cannot lead to a valid solution. The time complexity of the algorithm is O(N!), making it inefficient for large values of N but feasible for moderate sizes.

**2. Algorithm**

1. Start from the leftmost column.
2. Place the queen in a row where it is not attacked.
3. If a valid position is found, move to the next column and repeat step 2.
4. If no position is valid, backtrack to the previous column and move the queen to the next available row.
5. Repeat until all queens are placed.
6. If all columns are filled, print the solution.

**3. Source Code**

# N-Queens Problem Using Backtracking

def print\_solution(board):

    for row in board:

        print(" ".join("Q" if cell else "." for cell in row))

    print()

def is\_safe(board, row, col, n):

    for i in range(col):

        if board[row][i]:

            return False

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):

        if board[i][j]:

            return False

    for i, j in zip(range(row, n), range(col, -1, -1)):

        if board[i][j]:

            return False

    return True

def solve\_n\_queens(board, col, n):

    if col >= n:

        print\_solution(board)

        return True

    res = False

    for i in range(n):

        if is\_safe(board, i, col, n):

            board[i][col] = True

            res = solve\_n\_queens(board, col + 1, n) or res

            board[i][col] = False  # Backtrack

    return res

def n\_queens(n):

    board = [[False] \* n for \_ in range(n)]

    if not solve\_n\_queens(board, 0, n):

        print("No solution exists")

# Example usage:

n\_queens(4)

**4. Input**

N = 4

**5. Output**

. Q . .

. . . Q

Q . . .

. . Q .

. . Q .

Q . . .

. . . Q

. Q . .

**Lab Report 08: Solving the Sum of Subset Problem**

**Theory**

The **Sum of Subset Problem** is a well-known combinatorial optimization problem. Given a set of integers and a target sum, the goal is to determine if there is a subset of the set whose sum is equal to the target sum.

This is a **NP-complete problem**, meaning that there is no known polynomial-time solution to this problem. However, a **backtracking** approach or a **dynamic programming** approach can be used to find the solution efficiently for smaller inputs.

The **backtracking approach** explores each possible subset of the set SSS and checks whether its sum equals the target sum ddd.

**Algorithm**

1. **Input**: A set SSS of integers and a target sum ddd.
2. **Output**: A subset of SSS whose sum equals ddd (if one exists), or a message indicating that no such subset exists.

The steps for solving the problem using backtracking are:

* Start with an empty subset.
* For each element in the set, decide to either include or exclude it from the current subset.
* For each decision, check if the sum of the current subset equals ddd. If it does, print the subset and terminate.
* If the sum exceeds ddd, backtrack (undo the last decision).
* If the algorithm completes without finding a valid subset, report that no solution exists.

**Source Code**

# Python program to solve the Sum of Subset Problem using Backtracking

def sum\_of\_subset(S, d, subset=[], index=0):

    # Base case: If sum of the current subset equals d

    if sum(subset) == d:

        print("Subset with sum", d, ":", subset)

        return True

    # If we have considered all elements

    if index == len(S):

        return False

    # Include current element in subset

    subset.append(S[index])

    if sum\_of\_subset(S, d, subset, index + 1):

        return True

    # Exclude current element from subset (backtrack)

    subset.pop()

    if sum\_of\_subset(S, d, subset, index + 1):

        return True

    # No valid subset found

    return False

# Input set and target sum

S = [5, 10, 1212, 13, 15, 18]

d = 30

# Call the function

if not sum\_of\_subset(S, d):

    print("No subset with sum", d, "exists.")

**Input**

The input consists of:

* A set S={5,10,1212,13,15,18}S = \{5, 10, 1212, 13, 15, 18\}S={5,10,1212,13,15,18}
* A target sum d=30d = 30d=30

**Output**

1. The program checks all possible subsets of SSS and looks for a subset whose sum equals ddd.
2. If such a subset exists, it is printed. If not, the program prints a message saying no such subset exists.

For the input S={5,10,1212,13,15,18}S = \{5, 10, 1212, 13, 15, 18\}S={5,10,1212,13,15,18} and d=30d = 30d=30, the output will be:

Subset with sum 30 : [5, 10, 15]

# **Lab Report 09: 0/1 Knapsack Problem Using Dynamic Programming**

## **1. Theory**

The problem is NP-hard, but a **dynamic programming (DP)** approach can solve it in pseudo-polynomial time. The DP approach involves building a table dp[i][w] where:

* i represents the first i items,
* w represents the capacity constraint up to C.

The recurrence relation used is:

* If the weight of the current item is less than or equal to w, then:

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]])

* Otherwise, the item cannot be included:

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

The value dp[n][C] will then represent the maximum profit that can be achieved with the given items and knapsack capacity.

## **2. Algorithm**

**Step 1:** Initialize a DP table dp with dimensions (n+1) x (C+1) with all values set to 0.

**Step 2:** Loop through items i from 1 to n and capacities w from 1 to C:

* If W[i-1] <= w:
  + Update dp[i][w] with the maximum of:
    - Not taking the item: dp[i-1][w]
    - Taking the item: P[i-1] + dp[i-1][w-W[i-1]]
* Otherwise, set dp[i][w] = dp[i-1][w].

**Step 3:** The maximum profit is found at dp[n][C].

**Step 4 (Optional):** Trace back through the table to determine which items were included in the optimal solution.

## **4. Source Code**

def knapsack(profits, weights, capacity):

    n = len(profits)

    # Create a DP table with dimensions (n+1) x (capacity+1)

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

    # Build the DP table

    for i in range(1, n + 1):

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

            if weights[i - 1] <= w:

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

            else:

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

    return dp[n][capacity], dp

def find\_selected\_items(dp, weights, capacity):

    selected\_items = []

    i = len(weights)

    w = capacity

    while i > 0 and w > 0:

        # Check if item i was included by comparing with the previous row

        if dp[i][w] != dp[i - 1][w]:

            selected\_items.append(i - 1)  # Include item index (0-indexed)

            w -= weights[i - 1]

        i -= 1

    selected\_items.reverse()  # To maintain the original order

    return selected\_items

# Input Data

profits = [15, 25, 13, 23]

weights = [2, 6, 12, 9]

capacity = 20

# Compute the solution

max\_profit, dp\_table = knapsack(profits, weights, capacity)

selected\_items = find\_selected\_items(dp\_table, weights, capacity)

# Display the results

print("Maximum Profit:", max\_profit)

print("Selected Items (0-indexed):", selected\_items)

print("Selected Items Details:")

for index in selected\_items:

    print(f"  Item {index + 1}: Profit = {profits[index]}, Weight = {weights[index]}")

## **Input**

* **Profits (P):** [15, 25, 13, 23]
* **Weights (W):** [2, 6, 12, 9]
* **Knapsack Capacity (C):** 20
* **Number of Items (n):** 4

## **Output**

When running the above program, you should see an output similar to:

Maximum Profit: 63

Selected Items (0-indexed): [0, 1, 3]

Selected Items Details:

Item 1: Profit = 15, Weight = 2

Item 2: Profit = 25, Weight = 6

Item 4: Profit = 23, Weight = 9

# Lab Report 10: Tower of Hanoi Problem

## **1. Theory**

The Tower of Hanoi problem is fundamentally recursive. The solution involves:

* **Moving N−1N-1N−1 disks:** First, move the top N−1N-1N−1 disks from the source rod to the auxiliary rod.
* **Moving the largest disk:** Next, move the NNNth (largest) disk directly from the source rod to the destination rod.
* **Moving N−1N-1N−1 disks again:** Finally, move the N−1N-1N−1 disks from the auxiliary rod to the destination rod, placing them on top of the largest disk.

The recurrence relation for the number of moves required is:

T(N)=2×T(N−1)+1withT(1)=1T(N) = 2 \times T(N-1) + 1 \quad \text{with} \quad T(1) = 1T(N)=2×T(N−1)+1withT(1)=1

This gives a total of 2N−12^N - 12N−1 moves.

## **2. Algorithm**

**Step-by-Step Algorithm:**

1. **Base Case:**
   * If there is only 1 disk, simply move it from the source rod to the destination rod.
2. **Recursive Steps:**
   * **Step 1:** Recursively move the top N−1N-1N−1 disks from the source rod to the auxiliary rod.
   * **Step 2:** Move the NNNth (largest) disk from the source rod to the destination rod.
   * **Step 3:** Recursively move the N−1N-1N−1 disks from the auxiliary rod to the destination rod.
3. **Stop Condition:**
   * The recursion stops when there is only 1 disk left.

## 3. Source Code

Below is the Python program implementing the recursive solution for the Tower of Hanoi problem:

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

    """

    Solve the Tower of Hanoi problem for n disks.

    Parameters:

    n (int): Number of disks.

    source (str): The source rod.

    auxiliary (str): The auxiliary rod.

    destination (str): The destination rod.

    """

    if n == 1:

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

        return

    # Move top n-1 disks from source to auxiliary, so they are out of the way

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

    # Move the nth disk from source to destination

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

    # Move the n-1 disks that we left on auxiliary to destination

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

# Main part of the program

# Input: Number of disks

n = int(input("Enter the number of disks: "))

# Rod names: A (source), B (auxiliary), C (destination)

print("\nThe sequence of moves to solve the Tower of Hanoi is:")

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

## Input

* **Number of Disks (NNN)**: The user is prompted to enter the number of disks.  
  Example Input:

Enter the number of disks: 3

## **Output**

For an input of N=3N = 3N=3, the expected output (sequence of moves) would be:

Enter the number of disks: 3

The sequence of moves to solve the Tower of Hanoi is:

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