

Pabna University of

Submission date: 26/02/2025

* **Submitted by:**Jannatul Azmi  
  Roll : 220623  
  Session :2021-2022   
  2nd year 2nd semester  
  Dept. of Information and Communication Engineering
* **Submitted to:**Md. Anwar Hossain

Professor

Department of Information and Communication Engineering

Pabna University of Science and Technology

Course Code : STAT-2202  
Course Title : Engineering

Statistics Sessional

**LAB REPORT**

Information and Communication Engineering

Department of

Science & Technology

|  |  |
| --- | --- |
| No. | 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.** | |

INDEX

Problem 1 :   
 write a program to sort a linear array using the bubble sort algorithm.

**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 pass through the list is repeated until the list is sorted. The algorithm gets its name because smaller elements gradually "bubble" to the top of the list.**

**Working Principle:**

1. **Compare adjacent elements.**
2. **Swap them if they are in the wrong order.**
3. **Move to the next pair of elements.**
4. **Repeat the process for each pair in the list.**
5. **Repeat the entire process for the remaining elements until no swaps are needed.**

**Time Complexity:**

* **Best Case: O(n) [If the array is already sorted]**
* **Worst Case: O(n^2)**
* **Average Case: O(n^2)**

**Space Complexity: O(1) [In-place Sorting Algorithm]**

**Algorithm:**

1. **Start**
2. **Input the array size n.**
3. **Initialize an empty array arr.**
4. **Input n elements into the array.**
5. **Display the original array.**
6. **Repeat for i from 0 to n-2:** 
   * **Display the current pass number.**
   * **Repeat for j from 0 to n-2-i:** 
     + **Compare arr[j] and arr[j+1].**
     + **If arr[j] > arr[j+1], swap them and display the updated array.**
     + **Else, display that no swap occurred.**
7. **Display the final sorted array.**
8. **End**

**Sample Input and Output:**

Input:

Array size: 5

Element no 1 : 64

Element no 2 : 34

Element no 3 : 25

Element no 4 : 12

Element no 5 : 22

Output:

Your array: [64, 34, 25, 12, 22]

Pass 1:

Comparing 64 and 34 => Swapped to [34, 64, 25, 12, 22]

Comparing 64 and 25 => Swapped to [34, 25, 64, 12, 22]

Comparing 64 and 12 => Swapped to [34, 25, 12, 64, 22]

Comparing 64 and 22 => Swapped to [34, 25, 12, 22, 64]

Pass 2:

Comparing 34 and 25 => Swapped to [25, 34, 12, 22, 64]

Comparing 34 and 12 => Swapped to [25, 12, 34, 22, 64]

Comparing 34 and 22 => Swapped to [25, 12, 22, 34, 64]

Pass 3:

Comparing 25 and 12 => Swapped to [12, 25, 22, 34, 64]

Comparing 25 and 22 => Swapped to [12, 22, 25, 34, 64]

Pass 4:

Comparing 12 and 22 => No Swap

Sorted Array: [12, 22, 25, 34, 64]

**Source code in python:**

n=int(input("Array size:"))

arr=[]

for i in range(n):

    element= int(input(f"Element no {i+1} :"))

    arr.append(element)

print("your array :",arr)

for i in range(n - 1):

    print(f"\nPass {i+1}:")

    for j in range(n - 1 - i):

        print(f"Comparing {arr[j]} and {arr[j+1]}", end=" ")

        if arr[j] > arr[j + 1]:

            arr[j], arr[j + 1] = arr[j + 1], arr[j]

            print(f"=> Swapped to {arr}")

        else:

            print("=> No Swap")

print("\nSorted Array:", arr)

Problem 2:

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

**Theory:**

Linear Search is a straightforward searching algorithm that checks each element of the list sequentially until the desired element is found or the list ends. It is the simplest search algorithm, particularly useful for small or unsorted datasets.

Working Principle:

1. Start from the first element.
2. Compare the target element with the current element.
3. If they match, return the index.
4. If they don't match, move to the next element.
5. Repeat the process until the target is found or the end of the list is reached.

Time Complexity:

* Best Case: O(1) [Element found at the first position]
* Worst Case: O(n) [Element found at the last position or not found at all]
* Average Case: O(n)

Space Complexity: O(1)

**Algorithm:**

1. Start
2. Define an array numbers.
3. Input the target element.
4. Initialize found as False.
5. Repeat for each element in the array:
   * Compare the current element with the target element.
   * If they match, print the index and set found to True, then break the loop.
6. If no match is found, print that the number is not in the list.
7. End

**Sample Input and Output:**

Input: Enter the number to search: 30

Enter the number to search: 30

Output:

Searching steps:

Step 1: Checking 10

Step 2: Checking 20

Step 3: Checking 30

Found at index 2

**Source code in Python:**

numbers = [10, 20, 30, 40, 50, 60, 70, 80]

target = int(input("Enter the number to search: "))

found = False

print("\n Searching steps:")

for i in range(len(numbers)):

    print(f"Step {i+1}: Checking {numbers[i]}")

    if numbers[i] == target:

        print(f" Found at index {i}")

        found = True

        break

if not found:

    print(" Number not found in the list.")

Problem 3:

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

**Theory:**

Merge Sort is a divide-and-conquer algorithm that divides the array into two halves, sorts each half recursively, and then merges the sorted halves to produce the final sorted array.

**Working Principle:**

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

**Time Complexity:**

* Best Case: O(n log n)
* Worst Case: O(n log n)
* Average Case: O(n log n)

**Space Complexity:** O(n) [Temporary array required for merging]

**Algorithm:**

1. Start
2. Divide the array into sub-arrays of size 1.
3. Merge adjacent sub-arrays into larger sorted sub-arrays.
4. Repeat the merging process until the entire array is sorted.
5. Display the sorted array.
6. End

**Sample Input and Output:**

**Input:**

Initial Array: [38, 27, 43, 3, 9, 82, 10]

**Output:**

Step 1: Merging sub-arrays of size 1

Merging: [38] & [27]

After Merge: [27, 38]

Merging: [43] & [3]

After Merge: [3, 43]

Merging: [9] & [82]

After Merge: [9, 82]

Array after Step 1: [27, 38, 3, 43, 9, 82, 10]

Step 2: Merging sub-arrays of size 2

Merging: [27, 38] & [3, 43]

After Merge: [3, 27, 38, 43]

Merging: [9, 82] & [10]

After Merge: [9, 10, 82]

Array after Step 2: [3, 27, 38, 43, 9, 10, 82]

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

**Source code in python:**

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

n = len(arr)

temp = [0] \* n

print("Initial Array:", arr)

step = 1

for size in [2\*\*i for i in range(n) if 2\*\*i < n]:

    print(f"\nStep {step}: Merging sub-arrays of size {size}")

    for left in range(0, n-1, 2\*size):

        mid = min(left + size - 1, n - 1)

        right = min(left + 2 \* size - 1, n - 1)

      print(f"  Merging: {arr[left:mid+1]} & {arr[mid+1:right+1]}")

        i, j, k = left, mid + 1, left

        for \_ in range(right - left + 1):

            if i <= mid and (j > right or arr[i] < arr[j]):

                temp[k] = arr[i]

                i += 1

            else:

                temp[k] = arr[j]

                j += 1

            k += 1

        for x in range(left, right + 1):

            arr[x] = temp[x]

        print(f"  After Merge: {arr[left:right+1]}")

    print(f"Array after Step {step}: {arr}")

    step += 1

print("\nSorted Array:", arr)

Problem 4:

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

**Theory:**

Binary search is a searching algorithm used to find the position of a target value within a **sorted array**. The basic idea behind the algorithm is to repeatedly divide the search interval in half.

1. If the target value is less than the value in the middle of the interval, the algorithm narrows the interval to the lower half.
2. If the target value is greater, the algorithm narrows it to the upper half.
3. This process is repeated until the target value is found or the interval is empty, meaning the target is not in the array.

In this problem, the array is first sorted before performing the binary search.

**Algorithm:**

1. **Input the unsorted array**.
2. **Sort the array**: This ensures that the array is in ascending order to apply binary search.
3. **Take input** for the target element you want to search.
4. **Initialize variables**: start = 0, end = len(arr) - 1, found = False.
5. **Loop until the element is found or the interval is invalid**:
   * Calculate the middle index mid = (start + end) // 2.
   * If the element at arr[mid] is equal to the target, print the index and set found = True.
   * If the target is greater than arr[mid], move the start index to mid + 1.
   * If the target is smaller, move the end index to mid - 1.
6. **If the element is not found** (when found is still False), print a message saying the element is not in the array.

**Sample Input and Output:**

**Sample Input**:

pgsql

CopyEdit

Unsorted Array [22, 13, 26, 15, 10, 41, 65]

Sorted Array: [10, 13, 15, 22, 26, 41, 65]

What to find: 26

**Sample Output**:

pgsql

CopyEdit

Element is found at index: 4

**Source code in python:**

arr =[22,13,26,15,10,41,65]

print("Unsorted Array",arr)

arr.sort()

print("Sorted Array:" ,arr)

target=int(input("What to find:"))

start=0

end= len(arr)-1

found= False

for \_ in range(len(arr)):

    if start> end:

        break

    mid= (start+end)//2

    if arr[mid] == target:

        print("Element is found at index: ", mid )

        found = True

        break

    elif arr[mid] <target:

        start= mid+1

    else:

        end = mid-1

if not found:

    print ("Element is not in the Array.")

Problem 5:

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

**Theory:**

Pattern matching is a technique used to find a substring (the "pattern") within a larger string (the "text"). The algorithm works by scanning the text from left to right, checking every possible position in the text where the pattern could match.

The brute force pattern matching approach checks each possible starting index in the text. For each starting index, it checks if the substring of the text matches the pattern. If a match is found, it returns the index where the pattern starts. If no match is found after scanning all the positions, it outputs that the pattern does not exist in the text.

**Algorithm:**

1. **Input the text and the pattern**: Take the main text and the pattern you want to find within that text.
2. **Initialize variables**: n is the length of the text, m is the length of the pattern, and found is initially set to False.
3. **Loop through the text**:
   * For each possible starting position i in the text (from 0 to n - m):
     + Assume the pattern matches (match = True).
     + Loop through each character of the pattern and compare it with the corresponding character in the text.
     + If any character does not match, set match = False and break out of the inner loop.
     + If all characters match, print the index i where the pattern is found and set found = True.
4. **Check if the pattern was found**: If after checking all positions found is still False, print that the pattern was not found.

**Sample Input and Output:**

**Sample Input**:

vbnet

CopyEdit

Your text: This is a simple text matching example.

What to find: simple

**Sample Output**:

pgsql

CopyEdit

Pattern found in index: 10

If the pattern doesn't exist in the text, for example:

**Sample Input**:

arduino

CopyEdit

Your text: I am learning pattern matching.

What to find: algorithm

**Sample Output**:

mathematica

CopyEdit

Pattern not found in the Text.

**Source code in python:**

text = input("Your text:")

pattern =input("what to find:")

n=len(text)

m=len(pattern)

found= False

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:

        print("Pattern found in index:",i)

        found=True

if not found:

    print("Pattern not found in the Text.")

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 **FIFO (First In First Out)** principle. This means that elements are added (enqueued) at the back of the queue and removed (dequeued) from the front.

The typical operations on a queue are:

* **enqueue**: Adding an element to the queue (at the back).
* **dequeue**: Removing an element from the queue (from the front).
* **peek**: Viewing the element at the front of the queue without removing it.
* **isEmpty**: Checking if the queue is empty.

In this problem, we implement a simple queue using Python lists and perform typical queue operations such as adding, removing, and peeking at elements.

**Algorithm:**

1. **Create an empty queue**: Use an empty list queue = [].
2. **Enqueue elements**: Take four elements from the user using input() and add them to the queue using the append() method.
3. **Print the current queue**: Display the queue after enqueuing the elements.
4. **Dequeue an element**:
   * If the queue is not empty, remove the first element (queue[0]) and update the queue to remove that element (queue = queue[1:]).
   * Print the removed element and the queue after removal.
   * If the queue is empty, print a message saying the queue is empty.
5. **Peek at the first element**: If the queue is not empty, print the first element (queue[0]); otherwise, print that the queue is empty.

**Sample Input and Output:**

**Sample Input**:

yaml

CopyEdit

element 1: 10

element 2: 20

element 3: 30

element 4: 40

**Sample Output**:

yaml

CopyEdit

Queue: [10, 20, 30, 40]

Removed: 10

Queue after removal: [20, 30, 40]

First item: 20

If the queue is empty before removal:

**Sample Input**:

yaml

CopyEdit

element 1: 10

element 2: 20

element 3: 30

element 4: 40

Queue is emptied after removal of all elements.

**Sample Output**:

yaml

CopyEdit

Queue: [10, 20, 30, 40]

Removed: 10

Queue after removal: [20, 30, 40]

Removed: 20

Queue after removal: [30, 40]

Removed: 30

Queue after removal: [40]

Removed: 40

Queue after removal: []

Queue is empty.

First item: Queue is empty.

**Source Code in python:**

queue = []

queue.append(int(input("element 1:")))

queue.append(int(input("element 2:")))

queue.append(int(input("element 3:")))

queue.append(int(input("element 4:")))

print("Queue:", queue)

if len(queue) > 0:

    removed = queue[0]

    queue = queue[1:]

    print("Removed:", removed)

    print("Queue after removal:", queue)

else:

    print("Queue is empty.")

if len(queue) > 0:

    print("First item:", queue[0])

else:

    print("Queue is empty.")

Problem-7:

Write a program to solve **n** queen's problem using backtracking.

**Theory:**

The N Queen’s problem can be solved using a backtracking approach. The idea is to place a queen in each row and column while ensuring that no two queens share the same diagonal or row. We recursively try to place queens in each row one by one, starting from the first row, and move to the next row only if the current queen placement is safe. If no safe spot is found, we backtrack and move the queen to the next column in the previous row.

**Algorithm:**

1. Create a board of size N×N (NxN).
2. Start placing queens in the first row.
3. For each row, try placing the queen in all columns.
4. For each column, check if the queen’s position is safe (no two queens are attacking each other).
5. If the placement is valid, move to the next row.
6. If placing the queen leads to a solution (i.e., all queens are placed), return the solution.
7. If no valid position is found for the queen in the current row, backtrack to the previous row and move the queen to the next column.
8. Repeat the process until all queens are placed successfully.

**Sample Input and Output:**

**Input:**

n= 4

**Output:**

Solution 1:

[ [1, 0, 0, 0],

[0, 0, 1, 0],

[0, 0, 0, 1],

[0, 1, 0, 0] ]

Solution 2:

[ [0, 0, 1, 0],

[1, 0, 0, 0],

[0, 1, 0, 0],

[0, 0, 0, 1] ]

**Source Code in Python:**

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

for i in range(row):

if board[i][col] == 1:

return False

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

if board[i][j] == 1:

return False

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

if board[i][j] == 1:

return False

return True

def solve\_nqueens(board, row, n):

if row == n:

# All queens are placed successfully

return True

for col in range(n):

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

# Place the queen

board[row][col] = 1

if solve\_nqueens(board, row + 1, n):

return True

board[row][col] = 0

return False

def print\_board(board, n):

for row in range(n):

print(board[row])

def n\_queens(n):

board = [[0 for \_ in range(n)] for \_ in range(n)]

if not solve\_nqueens(board, 0, n):

print("No solution exists")

return

print("Solution:")

print\_board(board, n)

n = 4

n\_queens(n)

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 to find a subset of a given set S whose sum is exactly equal to a specified number d. This is a classic example of a combinatorial problem that can be solved using backtracking. The goal is to explore all possible subsets of the set S and check whether their sum equals d.

**2. Algorithm:**

1. Create a function that explores all subsets of S.
2. For each subset, calculate its sum.
3. If the sum of a subset is equal to d, print that subset.
4. Use backtracking to generate the subsets of S.

**Sample Input and Output:**

**Input:**

S = {5, 10, 12, 13, 15, 18}

d = 30

**Output:**

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

Subset with sum 30: [12, 18]

**Source Code in Python:**

def find\_subsets(S, n, d, index, current\_subset, current\_sum):

if current\_sum == d:

print("Subset with sum", d, ":", current\_subset)

return

if index == n:

return

find\_subsets(S, n, d, index + 1, current\_subset + [S[index]], current\_sum + S[index])

find\_subsets(S, n, d, index + 1, current\_subset, current\_sum)

def sum\_of\_subsets(S, d):

n = len(S)

find\_subsets(S, n, d, 0, [], 0)

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

d = 30

sum\_of\_subsets(S, d)

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.

**Theory:**

The 0/1 Knapsack problem is a classic problem in optimization. Given a set of items, each with a weight and a profit, and a knapsack with a maximum capacity, the goal is to determine the maximum profit you can achieve without exceeding the capacity of the knapsack. The 0/1 Knapsack means that each item can either be included or excluded from the knapsack (i.e., no fractions of items).

**Algorithm:**

1. Create a 2D array dp[][] where dp[i][j] represents the maximum profit achievable with the first i items and a knapsack capacity of j.
2. The state transition is:
   * If the weight of the current item is less than or equal to the capacity j, you have two choices:
     + Include the item: dp[i-1][j-W[i-1]] + P[i-1]
     + Exclude the item: dp[i-1][j]
   * The value of dp[i][j] will be the maximum of these two choices.
3. Initialize the base case where dp[0][j] = 0 (0 items means 0 profit) and dp[i][0] = 0 (0 capacity means 0 profit).
4. The final solution will be dp[n][C], which gives the maximum profit for n items and capacity C.

**Sample Input and Output:**

**Input:**

P = (15, 25, 13, 23)

W = (2, 6, 12, 9)

C = 20

n = 4

**Output:**

Maximum profit: 38

**Source Code in Python:**

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

# Create a 2D array for dynamic programming

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

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

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

if W[i-1] <= j:

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

else:

dp[i][j] = dp[i-1][j]

return dp[n][C]

P = [15, 25, 13, 23]

W = [2, 6, 12, 9]

C = 20

n = 4

result = knapsack(P, W, C, n)

print("Maximum profit:", result)

Problem-10:

Write a program to solve the Tower of Hanoi problem for the N disk.

**Theory:**

In the Tower of Hanoi problem, you are given three poles and a number of disks. The objective is to move all disks from the source pole to the target pole using the auxiliary pole. The challenge is to solve this problem with the least number of moves, which follows the recursive solution pattern.

The minimum number of moves required to solve the problem is 2^N - 1, where N is the number of disks.

**Algorithm:**

1. If there is only one disk, move it from the source pole to the target pole.
2. For more than one disk, first move N-1 disks from the source pole to the auxiliary pole using the target pole.
3. Then move the Nth disk from the source pole to the target pole.
4. Finally, move the N-1 disks from the auxiliary pole to the target pole using the source pole.

**Sample Input and Output:**

**Input:**

N = 3

**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

**Source Code in Python:**

def tower\_of\_hanoi(N, source, target, auxiliary):

if N == 1:

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

return

tower\_of\_hanoi(N-1, source, auxiliary, target)

# Move the Nth disk from source to target

print(f"Move disk {N} from {source} to {target}")

tower\_of\_hanoi(N-1, auxiliary, target, source)

N = 3

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