****

**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 01:**

**Titte : Write a program to sort a linear array using bubble sort algorithm**.

**Theory:** Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the entire array is sorted. The algorithm derives its name from the way smaller elements "bubble" to the top of the list with each pass.

* **Time Complexity:**
  + Worst-case: O(n²)
  + Average-case: O(n²)
  + Best-case: O(n) (when already sorted)
* **Space Complexity:** O (1) (in-place sorting)

Bubble Sort is inefficient for large datasets but is useful for educational purposes and scenarios where simplicity is preferred over efficiency.

**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 until the end of the array is reached.
5. Repeat the entire process for (n-1) passes, where n is the number of elements in the array.
6. Stop when no swaps are needed in a full pass.

**Source Code:**

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

n = len(arr)

for i in range(n):

    for j in range(0, n - i - 1):

        if arr[j] > arr[j + 1]:

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

print("Sorted array:", arr)

**Input**:

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

**Output:**

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

.

**Lab-02:**

**Title: Write a program to find an element using the linear search algorithm.**

**Theory:** Linear Search is a simple searching algorithm that sequentially checks each element in a list until the target element is found or the end of the list is reached. It is straightforward but inefficient for large datasets.

### Characteristics of Linear Search:

* **Time Complexity:**
  + Worst-case: O(n)
  + Average-case: O(n)
  + Best-case: O(1) (if the element is found at the beginning)
* **Space Complexity:** O(1) (in-place search)

**Algorithm:**

1. Start from the first element of the array.
2. Compare the current element with the target element.
3. If the current element matches the target, return its index.
4. If the end of the array is reached without finding the element, return -1 (indicating not found).

**Source Code:**

def linear\_search(arr, target):

    for i in range(len(arr)):

        if arr[i] == target:

            return i

    return -1

array = [10, 20, 30, 35, 40, 50]

target = 30

print("Element found at index:", linear\_search(array, target))

**Input:**

Array = [10, 20, 30, 35, 40, 50]

Target = 30

**Output:**

Element found at index: 2

**Lab 03:**

**Title: Write a program to sort a linear array using the merge sort algorithm.**

**Theory:**

Merge Sort follows the divide-and-conquer paradigm, which involves three main steps:

1. **Divide:** The input array is divided into two halves until each subarray contains only one element.
2. **Conquer:** Each subarray is sorted recursively.
3. **Merge:** The sorted subarrays are merged together to produce the final sorted array.

The merge operation is the key step in Merge Sort, where two sorted arrays are combined to maintain the overall order. Since Merge Sort splits the array in half at each step, the depth of recursion is O (log n), and merging at each level takes O(n) time, resulting in an overall time complexity of O (n log n). The algorithm is stable, meaning that the relative order of equal elements is preserved, and it is particularly useful for sorting linked lists due to its efficient merging process.

**Algorithm:**

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

**Source Code:**

def merge\_sort(arr):

    if len(arr) > 1:

        mid = len(arr) // 2

        left\_half = arr[:mid]

        right\_half = arr[mid:]

        merge\_sort(left\_half)

        merge\_sort(right\_half)

        i = j = k = 0

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

            if left\_half[i] < right\_half[j]:

                arr[k] = left\_half[i]

                i += 1

            else:

                arr[k] = right\_half[j]

                j += 1

            k += 1

        while i < len(left\_half):

            arr[k] = left\_half[i]

            i += 1

            k += 1

        while j < len(right\_half):

            arr[k] = right\_half[j]

            j += 1

            k += 1

    return arr

# Example

array = [12, 11, 13, 5, 6, 7]

print("Sorted Array:", merge\_sort(array))

**Input:**

array = [12, 11, 13, 5, 6, 7]

**Output:**

Sorted Array: [5, 6, 7, 11, 12, 13]

**Lab 04:**

**Title: Write a program to find an element using the binary search algorithm**

**Theory:**

Binary search is an efficient searching technique that follows the divide-and-conquer approach. It works by repeatedly dividing the search interval in half. If the value of the target element is equal to the middle element of the sorted array, the search is complete. If the target is smaller, the search continues in the left half, otherwise, it continues in the right half. This process repeats until the element is found or the search interval is empty.

The time complexity of binary search is **O (log n)**, making it much more efficient than linear search, which has a time complexity of **O(n)**. However, binary search requires the array to be sorted beforehand, which may involve an additional computational cost depending on the sorting algorithm used.

**Algorithm:**

1. Define an array of unsorted integers.
2. Sort the array using Python’s built-in sorting function.
3. Implement the binary search algorithm.
4. Execute the algorithm to search for a target element.
5. Compare its performance with a linear search.

**Source Code:**

def binary\_search(arr, target):

    low = 0

    high = len(arr) - 1

    while low <= high:

        mid = (low + high) // 2

        if arr[mid] == target:

            return mid

        elif arr[mid] < target:

            low = mid + 1

        elif arr[mid] > target:

            high = mid - 1

    return -1

array = [2, 3, 4, 10, 40]

target = 4

print("Element found at index:", binary\_search(array, target))

**Input:**

array = [2, 3, 4, 10, 40, 50]

target = 4

**Output:**

**Element found at index: 2**

**Lab 05:**

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

**Theory:**

Pattern matching is the process of searching for a sequence of characters (pattern) within another sequence (text). One of the fundamental pattern matching algorithms is the Knuth-Morris-Pratt (KMP) algorithm, which preprocesses the pattern to allow efficient searching within the text. The time complexity of naive pattern matching is **O(m\*n)**, while efficient algorithms like KMP run in **O(n + m)** time, where **n** is the length of the text and **m** is the length of the pattern.

**Algorithm:**

1. Accept a text and pattern as input.
2. Implement a pattern matching algorithm (e.g., naive search or KMP algorithm).
3. Search for occurrences of the pattern in the text.
4. Output the indices where the pattern is found.

**Source Code:**

def naive\_pattern\_search(text, pattern):

    text\_length = len(text)

    pattern\_length = len(pattern)

    for i in range(text\_length - pattern\_length + 1):

        match = True

        for j in range(pattern\_length):

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

                match = False

                break

        if match:

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

text = "some people say anger brings disaster"

pattern = "anger"

naive\_pattern\_search(text, pattern)

**Input:**

Text= “some people say anger brings disaster”

Pattern= “anger**”**

**Output:**

Pattern found at index 17

**Lab 06:**

**Title: 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. In this structure, elements are inserted at the rear and removed from the front. A queue is commonly used in scenarios where order matters, like scheduling tasks, handling requests, or managing processes in an operating system.

**Algorithm:**

* + 1. **Create Queue Class**:
* Initialize an empty queue using deque().
  + 1. **Enqueue Operation**:
* Add the element to the end of the queue using append(item).
  + 1. **Dequeue Operation**:
* If the queue is not empty, remove the element from the front using popleft().
* If the queue is empty, display a message that dequeueing is not possible.
  + 1. **Peek Operation**:
* If the queue is not empty, return the front element.
* If the queue is empty, return a message saying the queue is empty.
  + 1. **Is Empty Operation**:
* Check if the length of the queue is zero.
* Return True if empty, otherwise return False.
  + 1. **Size Operation**:
* Return the number of elements in the queue using len().

**Source Code:**

class Queue:

    def \_\_init\_\_(self):

        self.queue = []

    def enqueue(self, item):

        self.queue.append(item)

    def dequeue(self):

        if not self.is\_empty():

            return self.queue.pop(0)

        return "Queue is empty"

    def is\_empty(self):

        return len(self.queue) == 0

    def peek(self):

        if not self.is\_empty():

            return self.queue[0]

        return "Queue is empty"

    def display(self):

        return self.queue

# Example

q = Queue()

q.enqueue(10)

q.enqueue(20)

q.enqueue(30)

print("Queue after enqueue:", q.display())

print("Dequeued element:", q.dequeue())

print("Queue after dequeue:", q.display())

**Input:**

Queue after enqueue: [10, 20, 30]

Dequeued element: 10

**Output:**

Queue after dequeue: [20, 30]

**Lab 07:**

**Title: Write a program to solve the n queen’s problem using backtracking.**

**Theory:**

The **N-Queens problem** is a classic combinatorial problem in which **N queens** must be placed on an **N × N chessboard** so that no two queens attack each other. A queen can attack another queen if they share the **same row, column, or diagonal**.

### **Backtracking Approach**

* The algorithm places queens **one by one in different columns**.
* If a safe position is found, it proceeds to place the next queen.
* If no safe position is available, it **backtracks** by removing the previous queen and trying the next possibility.

**Algorithm:**

1. Start with the **first column** and attempt to place a queen in each row.
2. Check if a **queen can be placed safely** in the current row.
3. If yes, place the queen and move to the **next column**.
4. Repeat steps 1-3 for subsequent columns.
5. If placing a queen in a column is not possible, **backtrack** to the previous column.
6. If all queens are placed successfully, **print the solution**.

**Source Code:**

def solve\_n\_queens(n):

    def is\_safe(board, row, col):

        for i in range(row):

            if board[i] == col or \

               board[i] - i == col - row or \

               board[i] + i == col + row:

                return False

        return True

    def place\_queen(board, row):

        if row == n:

            print(board)

            return

        for col in range(n):

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

                board[row] = col

                place\_queen(board, row + 1)

                board[row] = -1

    board = [-1] \* n

    place\_queen(board, 0)

# Example usage

n = 4  # You can change this value to solve for different sizes

solve\_n\_queens(n)

**Output:**

**[1, 3, 0, 2]**

**[2, 0, 3, 1]**

**Lab 08:**

**Title: Consider a set S= {5,10,12,13,15,18) and d=30.write a program to solve the sum of subsets problem.**

**Theory:**

The **Sum of Subsets Problem** is a classic **combinatorial problem** in which we must find all possible subsets of a given set S whose sum equals a target value d.

This problem is solved efficiently using **backtracking**, which systematically searches for solutions by exploring possible subsets and backtracking whenever a partial solution cannot lead to a valid solution.

**Algorithm:**

1. Sort the Set **S** to improve efficiency.

2. Start with an empty subset and incrementally add elements.

3. Maintain the current sum of elements in the subset.

4. If the current sum equals **d**, print the subset.

5. If the sum exceeds **d**, backtrack (remove the last added element).

6. Repeat until all subsets have been explored.

**Source Code:**

def sum\_of\_subsets(S, target, current\_sum=0, subset=[], index=0):

    if current\_sum == target:

        print(subset)

        return

    if current\_sum > target or index >= len(S):

        return

    sum\_of\_subsets(S, target, current\_sum + S[index], subset + [S[index]], index + 1)

    sum\_of\_subsets(S, target, current\_sum, subset, index + 1)

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

target\_sum = 30

print("Subsets with sum", target\_sum, ":")

sum\_of\_subsets(S, target\_sum)

**Output:**

Subsets with sum 30 :

[5, 10, 15]

[5, 12, 13]

[12, 18]

**Lab 09:**

**Title: 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 optimization problem where we aim to **maximize the total profit** by selecting items within a given **capacity limit**. The problem is called **0/1** because each item can either be included (1) or not included (0) in the knapsack.

**Mathematical Formulation**

Given:

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

Define K[i][j] as the **maximum profit** obtained using the first iii items with a knapsack capacity of j.

The recurrence relation is:

K[i][j] if W[i]>j otherwise

**Algorithm**

1. Create a **DP table** K[n+1][C+1]K[n+1][C+1]K[n+1][C+1] initialized with zeros.
2. Iterate over **each item** and **each capacity**:
   * If the item's weight is **greater** than the current capacity, exclude it.
   * Otherwise, compute the **maximum profit** by including or excluding the item.
3. The final answer is stored in K[n][C]K[n][C]K[n][C].

**Source Code**:

def knapsack(profits, weights, capacity, n):

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

    for i in range(1, n + 1):

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

            if weights[i - 1] > w:

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

            else:

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

    selected\_items = []

    w = capacity

    for i in range(n, 0, -1):

        if dp[i][w] != dp[i - 1][w]:

            selected\_items.append(i - 1)

            w -= weights[i - 1]

    return dp[n][capacity], selected\_items

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

profits = list(map(int, input("Enter the profits separated by spaces: ").split()))

weights = list(map(int, input("Enter the weights separated by spaces: ").split()))

capacity = int(input("Enter the knapsack capacity: "))

max\_profit, selected\_items = knapsack(profits, weights, capacity, n)

print(f"\nMaximum Profit: {max\_profit}")

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

print("Selected Profits:", [profits[i] for i in selected\_items])

print("Selected Weights:", [weights[i] for i in selected\_items])

**Input:**

Enter the number of items: 4

Enter the profits separated by spaces: 15 25 13 23

Enter the weights separated by spaces: 2 6 12 9

Enter the knapsack capacity: 20

**Output:**

Maximum Profit: 53

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

Selected Profits: [23, 25, 15]

Selected Weights: [9, 6, 2]

**Lab 10:**

**Title: write a program to solve the Tower of Hanoi problem for the N disk.**

**Theory:**

The Tower of Hanoi is a mathematical puzzle that consists of three rods and **N** disks of different sizes. The objective is to move all the disks from the source rod to the destination rod while following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk.
3. Only the top disk of any rod can be moved.

The problem was invented by the French mathematician **Édouard Lucas** in 1883.

**Mathematical Formula**

The minimum number of moves required to solve the Tower of Hanoi problem is:

T(N)=−1

where **N** is the number of disks.

**Algorithm**

1. **Base Case**: If **N == 1**, move the disk from the source rod to the destination rod.
2. **Recursive Step**:
   * Move **N-1** disks from the source rod to an auxiliary rod.
   * Move the **Nth** disk from the source rod to the destination rod.
   * Move **N-1** disks from the auxiliary rod to the destination rod.

This process follows a recursive **divide and conquer** approach.

**Source Code:**

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

    if n == 1:

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

        return

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

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

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

# Input: Number of disks

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

print(f"Solution for {n} disks:")

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

**Input:**

Enter the number of disks: 3

**Output:**

Solution for 3 disks:

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