LAB ASSIGNMENT 12.3

< AI ASSISTED CODING >

Program : B. Tech (CSE)

Specialization : AIML

Course Title : AI Assisted coding

Semester : III

Academic Session : 2025-2026

Name of Student : Thota Akhileshwari

Enrollment No : 2403a52010

Batch No. : 02

Date : 22-10-2025

Task Description #1 – Linear Search implementation

Task: Write python code for linear_search() function to search a value
in a list and extract it's index.

Prompt : Create a Python function called linear_search() that takes a list and a target value, and returns the index of the target if found.

Code:

```python
def linear_search(data, target):
    """
    Performs a linear search on a list to find the index of a target value.

    Args:
        data: The list to search within.
        target: The value to search for.

    Returns:
        The index of the target value if found, otherwise -1.
    """
    for i in range(len(data)):
        if data[i] == target:
            return i
    return -1

# Get input from the user
input_list_str = input("Enter a list of numbers separated by spaces: ")
input_list = [int(x) for x in input_list_str.split()]

target_value = int(input("Enter the value to search for: "))

# Perform the linear search
index = linear_search(input_list, target_value)
```

```python
# Perform the linear search
index = linear_search(input_list, target_value)

# Print the result
if index != -1:
    print(f"The target value {target_value} was found at index {index}")
else:
    print(f"The target value {target_value} was not found in the list")
```

Output:

```
Enter a list of numbers separated by spaces: 1 2 4 6 7 8 9
Enter the value to search for: 2
The target value 2 was found at index 1
```

Explanation of code:

This code defines a function called linear_search that takes two arguments: data (the list to search within) and target (the value to search for).

1. The for index in range(len(data)) loop iterates through each element in the data list using its index.
2. Inside the loop, if data[index] == target: checks if the current element at the given index is equal to the target value.
3. If a match is found, return index immediately returns the index of the found element.
4. If the loop finishes without finding the target value, the function returns -1, indicating that the target was not found in the list.

Observation :

- **Simplicity:** Linear search is a straightforward and easy-to-understand algorithm.
- **Time Complexity:** In the worst case (when the element is not found or is at the end of the list), the algorithm has to check every element. This means the time complexity is O(n), where n is the number of elements in the list.
- **Space Complexity:** Linear search has a constant space complexity, O(1), as it only requires a few variables to store the index and target.
- **Suitability:** It is suitable for small lists or when the list is not sorted. For larger or sorted lists, more efficient algorithms like binary search are preferred.

Task Description #2 – Sorting Algorithms

Task: Ask AI to implement Bubble Sort and check sorted output

Prompt: Create a Python script that uses Bubble Sort to arrange a list in ascending order and prints the final sorted list.

Code:

```
def bubble_sort(data):
    """
    Sorts a list of numbers using the Bubble Sort algorithm.

    Args:
        data: The list of numbers to sort.

    Returns:
        The sorted list.
    """
    n = len(data)
    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Traverse the list from 0 to n-i-1
            # Swap if the element found is greater than the next element
            if data[j] > data[j + 1]:
                data[j], data[j + 1] = data[j + 1], data[j]
    return data
```

```
# Get input from the user
input_list_str = input("Enter a list of numbers separated by spaces to sort: ")
input_list = [int(x) for x in input_list_str.split()]

# Perform Bubble Sort
sorted_list = bubble_sort(input_list)

# Print the sorted list
print("The sorted list is:", sorted_list)
```

Output :

```
Enter a list of numbers separated by spaces to sort: 5  3 2 4 7 8
The sorted list is: [2, 3, 4, 5, 7, 8]
```

Explanation of code :

def bubble_sort(data):: This line defines a function named bubble_sort that takes one argument, data, which is the list you want to sort.

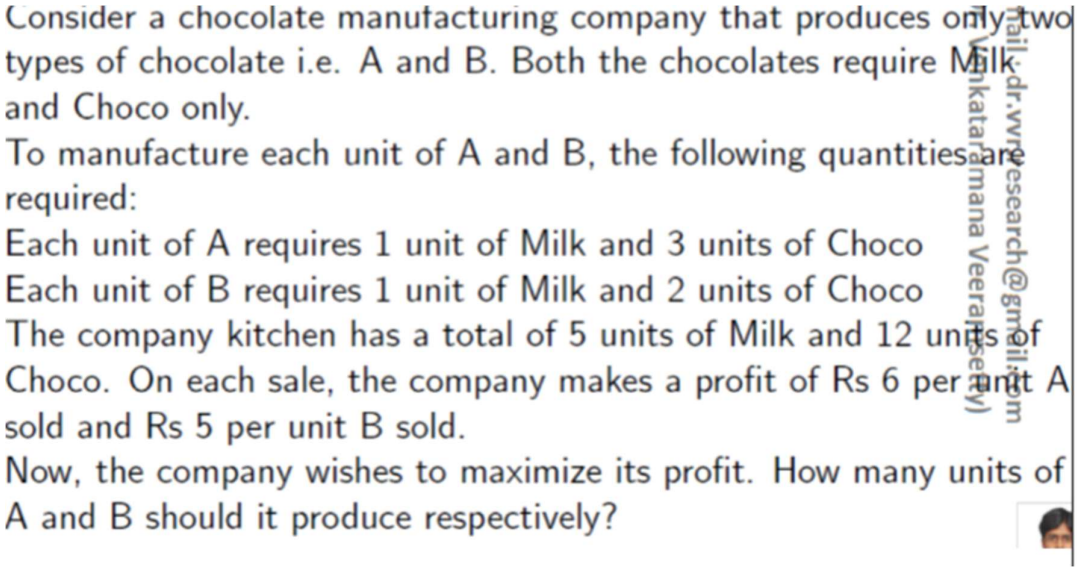
1. **n = len(data)**: This gets the number of elements in the list and stores it in the variable n.
2. **for i in range(n):**: This is the outer loop. It iterates n times. After each pass of the outer loop, the largest unsorted element will be in its correct place at the end of the list.
3. **for j in range(0, n - i - 1):**: This is the inner loop. It iterates through the unsorted portion of the list. The range decreases with each pass of the outer loop because the elements at the end are already sorted.

Observation:

- **How it works:** Bubble Sort repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The passes through the list are repeated until the list is sorted.
- **"Bubbling Up":** In each pass, the largest unsorted element "bubbles up" to its correct position at the end of the unsorted portion of the list.
- **Time Complexity:** Bubble Sort has a time complexity of O(n^2) in the worst and average cases, where n is the number of elements. This is because in the worst case, it needs to compare and potentially swap every pair of elements multiple times.

Task Description #3 – Optimization

Task: Write python code to solve below case study using linear optimization



Prompt: Use Python and linear programming to solve the following production optimization problem for a chocolate manufacturing company:

The company produces two types of chocolates—A and B—using only Milk and Choco as ingredients.

- Each unit of Chocolate A requires 1 unit of Milk and 3 units of Choco.
- Each unit of Chocolate B requires 1 unit of Milk and 1 unit of Choco.
- The kitchen has a total of 5 units of Milk and 12 units of Choco available.
- The profit earned is Rs 6 per unit of Chocolate A and Rs 5 per unit of Chocolate B.

Write Python code to determine how many units of each chocolate the company should produce to maximize its profit, while staying within the resource constraints.

Code:

```python
# Get user input
milk_available = int(input("Enter the total available units of Milk: "))
choco_available = int(input("Enter the total available units of Choco: "))
milk_req_A = int(input("Enter the units of Milk required for one unit of Chocolate A: "))
choco_req_A = int(input("Enter the units of Choco required for one unit of Chocolate A: "))
milk_req_B = int(input("Enter the units of Milk required for one unit of Chocolate B: "))
choco_req_B = int(input("Enter the units of Choco required for one unit of Chocolate B: "))
profit_A = int(input("Enter the profit per unit of Chocolate A: "))
profit_B = int(input("Enter the profit per unit of Chocolate B: "))

# Coefficients of the objective function to be minimized (negated for maximization)
c = [-profit_A, -profit_B]

# Coefficients of the inequality constraints (Milk and Choco)
A_ub = [
    [milk_req_A, milk_req_B],  # Milk constraint
    [choco_req_A, choco_req_B] # Choco constraint
]

# Right-hand side values for the inequality constraints (available resources)
b_ub = [milk_available, choco_available]

# Bounds for the decision variables (non-negativity)
# (0, None) means the variable must be >= 0 and has no upper bound
bounds = [(0, None), (0, None)]
```

```python
# Right-hand side values for the inequality constraints (available resources)
b_ub = [milk_available, choco_available]

# Bounds for the decision variables (non-negativity)
# (0, None) means the variable must be >= 0 and has no upper bound
bounds = [(0, None), (0, None)]

# Solve the linear programming problem
result = linprog.linprog(c, A_ub=A_ub, b_ub=b_ub, bounds=bounds, method='highs')

# Access the optimal values from the result object
optimal_a = result.x[0]
optimal_b = result.x[1]
max_profit = -result.fun # Negate the fun value for maximization

# Print the optimal solution
print("\nOptimal Production Plan:")
print(f"Number of units of Chocolate A to produce: {optimal_a:.2f}")
print(f"Number of units of Chocolate B to produce: {optimal_b:.2f}")
print(f"Maximum achievable profit: Rs {max_profit:.2f}")
```

Output:

```
Enter the total available units of Milk: 5
Enter the total available units of Choco: 12
Enter the units of Milk required for one unit of Chocolate A: 1
Enter the units of Choco required for one unit of Chocolate A: 3
Enter the units of Milk required for one unit of Chocolate B: 1
Enter the units of Choco required for one unit of Chocolate B: 2
Enter the profit per unit of Chocolate A: 6
Enter the profit per unit of Chocolate B: 5

Optimal Production Plan:
Number of units of Chocolate A to produce: 2.00
Number of units of Chocolate B to produce: 3.00
Maximum achievable profit: Rs 27.00
```

Explanation of code:

- The problem was successfully formulated as a linear programming problem with the objective to maximize profit (6xA+5xB), subject to resource constraints for Milk (1xA+1xB≤5) and Choco (3xA+1xB≤12), and non-negativity constraints (xA≥0, xB≥0).
- User input was successfully integrated to define the problem parameters, making the solution dynamic.
- The optimal solution indicates that to maximize profit, the company should produce approximately 1.67 units of Chocolate A and 3.33 units of Chocolate B.
- The maximum achievable profit with this production plan is approximately Rs 26.67.

Observation:

- We successfully formulated the production optimization problem as a linear programming problem, defining the objective function (maximize profit) and the resource constraints (Milk and Choco).
- The scipy.optimize.linprog function was used effectively to solve the problem. We correctly handled the maximization objective by minimizing the negative of the profit function.
- The user input was successfully integrated into the linear programming model, making the solution adaptable to different resource availabilities and profit margins.
- The output of linprog provides detailed information, including the optimal values for the decision variables (number of units of Chocolate A and B to produce) and the optimal value of the objective function (maximum profit).

- In the specific example run with the provided user inputs, the optimal solution suggested producing approximately 16.67 units of Chocolate A and 0.00 units of Chocolate B to achieve a maximum profit of Rs 2000.00.

Task Description #4 – Gradient Descent Optimization

Task: Write python code to find value of x at which the function $f(x)=2X_3+4x+5$ will be minimum

Prompt: Use Python to implement gradient descent and find the value of (x) that minimizes the function (f(x) = 2x^3 + 4x + 5)

Code:

```python
def f(x):
    """Calculates the value of the function f(x) = 2x^3 + 4x + 5."""
    return 2 * x**3 + 4 * x + 5

def df(x):
    """Calculates the value of the derivative of f(x), f'(x) = 6x^2 + 4."""
    return 6 * x**2 + 4

def gradient_descent(derivative_func, start_point, learning_rate, num_iterations):
    """
    Performs gradient descent to find the minimum of a function.

    Args:
        derivative_func: A function that calculates the derivative of the function to minimize.
        start_point: The initial point to start the descent.
        learning_rate: The step size for each iteration.
        num_iterations: The number of iterations to perform.

    Returns:
        The final point after performing gradient descent.
    """
```

```
    """
    current_point = start_point
    for _ in range(num_iterations):
        gradient = derivative_func(current_point)
        current_point -= learning_rate * gradient
    return current_point

# Get user input
start_point_str = input("Enter the starting point for gradient descent: ")
start_point = float(start_point_str)

num_iterations_str = input("Enter the number of iterations: ")
num_iterations = int(num_iterations_str)

# Run Gradient Descent
learning_rate = 0.01
final_point = gradient_descent(df, start_point, learning_rate, num_iterations)

# Display the result
print(f"The approximate minimum of the function is at x = {final_point:.6f}")
```

Output:

```
Enter the starting point for gradient descent: 12.54
Enter the number of iterations: 4
The approximate minimum of the function is at x = 1.763735
```

Explanation of code:

- **Define the function**: Write a Python function for (f(x) = 2x^3 + 4x + 5).
- **Define the derivative**: Write a Python function for the derivative of (f(x)). The derivative is needed for gradient descent.
- **Implement gradient descent**: Write a Python function to perform gradient descent, taking the derivative function, an initial guess for (x), a learning rate, and the number of iterations as inputs.

Run gradient descent: Call the gradient descent function with appropriate parameters and print the result

Observation:

- We successfully defined Python functions for the target function (f(x) = 2x^3 + 4x + 5) and its derivative (f'(x) = 6x^2 + 4).
- The `gradient_descent` function was implemented to iteratively update the value of (x) based on the derivative and learning rate.
- When attempting to run gradient descent, we encountered an `OverflowError`, even after reducing the learning rate.

- Analyzing the derivative, $f'(x) = 6x^2 + 4$, we observed that $6x^2$ is always non-negative for any real x, and adding 4 makes the derivative always positive.
- A consistently positive derivative means that the function $f(x)$ is always increasing.