DESIGN AND ANALYSIS OF ALGORITHMS LAB

SECTION: 1

Session 1: Implementation of Simple Algorithms

Q1. Implement Euclid's algorithm to find GCD (15265, 15)and calculate the number of times **mod** and assignment operations will be required.

Euclid's algorithm is a method to compute the greatest common divisor (GCD) of two integers aa and bb. The algorithm repeatedly replaces the larger number by its remainder when divided by the smaller number until one of the numbers becomes zero. The non-zero number at that point is the GCD.

Here's the pseudocode for Euclid's algorithm:

1. Divide aa by bb, let r=a mod b.
2. Replace a with b and b with r.
3. Repeat steps 1 and 2 until b = 0.
4. The GCD is the value of aa.

For a= 15265 and b= 15, let's calculate step-by-step and count the number of operations:

Step-by-Step Calculation:

1. a= 15265, b = 15
    - r=15265 mod 15= 5 (1 mod operation)
    - Assign a= 15, b = 5 (2 assignments)
2. a= 15, b = 5
    - r=15mod 5= 0 (1 mod operation)
    - Assign a = 5, b = 0 (2 assignments)

Termination: b = 0, so GCD = a = 5.

Total Operations Count:

- Mod operations: 22
- Assignments: 44

Implementation in Python:

Here's a Python implementation:

```
def euclid_gcd(a, b):
    mod_count = 0
    assign_count = 0
```

```
    while b != 0:
        r = a % b
        mod_count += 1
        a, b = b, r
        assign_count += 2

    print("GCD:", a)
    print("Mod operations:", mod_count)
    print("Assignment operations:", assign_count)

euclid_gcd(15265, 15)
```

Output:

- GCD: 55
- Mod operations: 22
- Assignment operations: 44

Q2 (i). Implement Horner's rule for evaluating polynomial P(x)= $6x6+5x5+4x4-3x3+2x2+8x-7$ at x = 3. Calculate how many times

a. multiplications and addition operations will take
b. how many times the loop will iterate

Horner's rule is an efficient method for evaluating polynomials. The polynomial is expressed in a nested form, and the number of arithmetic operations is minimized. For a polynomial P(x) = a_n x^n + a_{n-1} x^{n-1} + ….. + a_0, Horner's rule rewrites it as:

P(x)=((…((anx+an−1)x+an−2)x+…)x+a0)

P(x)=6x6+5x5+4x4−3x3+2x2+8x−7 at x=3, using Horner's rule:

Polynomial in Horner's Form

P(x)=((((((6x+5)x+4)x−3)x+2)x+8)x−7)

Step-by-Step Evaluation

1. Start with the highest coefficient a_n = 6.
2. Iterate through coefficients in descending order of power, performing one multiplication and one addition at each step.

Calculation:

Let P(x)=((((((6·3+5)·3+4)·3−3)·3+2)·3+8)·3−7).

Iterations:

1. Start with result = 6.

2. Multiply by 3 and add 5:
   result=6·3+5=18+5=23.
3. Multiply by 3 and add 4:
   result=23·3+4=69+4= 73.
4. Multiply by 3 and add -3:
   result=73·3−3=219−3= 216.
5. Multiply by 3 and add 2:
   result=216·3+2=648+2=650.
6. Multiply by 3 and add 8:
   result=650·3+8=1950+8=1958.
7. Multiply by 3 and add -7:
   result=1958·3−7=5874−7=5867.

Final Result:

P(3) = 5867

Number of Operations:

1. **Multiplications**: 6 (one per iteration).
2. **Additions**: 6 (one per iteration).
3. **Loop Iterations**: 6 (equal to the degree of the polynomial).

Python Implementation:

```
def horner(coefficients, x):
    multiplications = 0
    additions = 0
    result = coefficients[0]  # Start with the highest coefficient

    for coeff in coefficients[1:]:
        result = result * x + coeff
        multiplications += 1
        additions += 1

    print("Result:", result)
    print("Multiplications:", multiplications)
    print("Additions:", additions)
    print("Loop Iterations:", len(coefficients) - 1)
```

Coefficients of P(x) = 6x^6 + 5x^5 + 4x^4 - 3x^3 + 2x^2 + 8x - 7

coefficients = [6, 5, 4, -3, 2, 8, -7]

x = 3

horner(coefficients, x)

Output:

- **Result**: 58675867
- **Multiplications**: 66
- **Additions**: 66
- **Loop Iterations**: 66

Q2(ii) Apply a brute force method to implement the above polynomial expression(Q2(i)) and compare it with Horner's method in terms of number of multiplication operations

Using the brute force method, we compute each term of the polynomial P(x) = 6x^6 + 5x^5 + 4x^4 - 3x^3 + 2x^2 + 8x - 7 separately and sum them up. This approach does not use the nested structure of Horner's method, leading to a higher number of multiplication operations.

Brute Force Calculation:

The polynomial is:

P(x) = 6x^6 + 5x^5 + 4x^4 - 3x^3 + 2x^2 + 8x - 7

For x = 3, compute each term:

1. 6x^6 = 6 * 3^6: requires 5 multiplications (to compute 3^6) and 1 multiplication to multiply by 6. Total = 6 multiplications.
2. 5x^5 = 5*3^5: requires 4 multiplications (to compute 3^5) and 1 multiplication to multiply by 5. Total = 5 multiplications.
3. 4x^4 = 4 * 3^4: requires 3 multiplications (to compute 3^4) and 1 multiplication to multiply by 4. Total = 4 multiplications.
4. -3x^3 = -3 * 3^3: requires 2 multiplications (to compute 3^3) and 1 multiplication to multiply by -3. Total = 3 multiplications.
5. 2x^2 = 2 * 3^2: requires 1 multiplication (to compute 3^2) and 1 multiplication to multiply by 2. Total = 2 multiplications.
6. 8x= 8 * 3: requires 1 multiplication.
7. -7: no multiplications required (constant term).

Total Multiplications:

6+5+4+3+2+1=21 multiplications.

Comparison with Horner's Method:

- **Horner's method:** 6 multiplications.

- **Brute force method:** 28 multiplications.

Horner's method is significantly more efficient than brute force, especially for high-degree polynomials.

Python Implementation of Brute Force:

```
def brute_force_polynomial(coefficients, x):
    degree = len(coefficients) - 1
    result = 0
    multiplications = 0

    for i, coeff in enumerate(coefficients):
        term_degree = degree - i
        # Compute power manually to count multiplications
        power = 1
        for _ in range(term_degree):  # Calculate x^term_degree
            power *= x
            multiplications += 1
        term_value = coeff * power
        if coeff != 0:
            multiplications += 1  # Multiplying by the coefficient
        result += term_value

    print("Result:", result)
    print("Multiplications:", multiplications)

# Coefficients of P(x) = 6x^6 + 5x^5 + 4x^4 - 3x^3 + 2x^2 + 8x - 7
coefficients = [6, 5, 4, -3, 2, 8, -7]
x = 3
brute_force_polynomial(coefficients, x)
```

Output:

- **Result**: 5867
- **Multiplications (Brute Force)**: 28

This confirms that Horner's method reduces the number of multiplications by 71.43% compared to the brute force approach.

Q3.Implement multiplication of two matrices A[4,4] and B[4,4] and calculate

(i) how many times the innermost and the outermost loops will run
(ii) total number of multiplications and additions in computing the multiplication

Matrix multiplication involves the dot product of the rows of the first matrix (A) with the columns of the second matrix (B). For two 4×4 matrices, A[4][4] and B[4][4], the resulting matrix C[4][4] is also 4×4. Each element C[i][j] in the resultant matrix is computed as:

$C[i][j]=k=0\sum 3A[i][k]\cdot B[k][j]$

Implementation

1. Use **three nested loops**:
    - The **outermost loop** iterates over rows of A (i).
    - The **middle loop** iterates over columns of B (j).
    - The **innermost loop** iterates over elements in the row of AA and column of B(k).
2. For each pair of indices (i, j), compute the dot product by summing the products of corresponding elements of A and B.

Steps to Calculate Operations

(i) Number of Loop Iterations:

- **Outermost loop (i):**
    - Runs 4 times (for each row of A).
- **Middle loop (j):**
    - For each i, j runs 4 times (for each column of B).
- **Innermost loop (k):**
    - For each (i, j), k runs 4 times (for the dot product computation).

Total iterations of innermost loop:

Outermost iterations×Middle iterations×Innermost iterations=4×4×4= 64

(ii) Total Multiplications and Additions:

For each element C[i][j]:

- Multiplications: 4 (one for each A[i][k]*B[k][j].
- Additions: 3 (to sum the products).

Total for all elements:

Multiplications=4×4×4 = 64

Additions=4×4×3= 48

Python Implementation

```
import numpy as np
```

```
def matrix_multiplication(A, B):
    n = len(A)
    C = [[0 for _ in range(n)] for _ in range(n)]  # Initialize result matrix
    multiplications = 0
    additions = 0

    for i in range(n):  # Outermost loop
        for j in range(n):  # Middle loop
            for k in range(n):  # Innermost loop
                C[i][j] += A[i][k] * B[k][j]
                multiplications += 1
                if k > 0:  # Addition starts after the first multiplication
                    additions += 1

    print("Resultant Matrix C:")
    for row in C:
        print(row)
    print("\nTotal Multiplications:", multiplications)
    print("Total Additions:", additions)

# Example 4x4 matrices
A = np.random.randint(1, 10, size=(4, 4)).tolist()
B = np.random.randint(1, 10, size=(4, 4)).tolist()

print("Matrix A:", A)
print("Matrix B:", B)

matrix_multiplication(A, B)
```

Output

Matrix A: [[3, 6, 6, 3], [1, 9, 6, 8], [1, 2, 7, 4], [3, 7, 7, 9]]

Matrix B: [[2, 3, 8, 9], [7, 1, 7, 6], [1, 4, 6, 4], [5, 7, 1, 3]]

Resultant Matrix C:

[69, 60, 105, 96],[111, 92, 115, 111],[43, 61, 68, 61],[107, 107, 124, 124]

Total Multiplications: 64

Total Additions: 48

1. **Innermost loop iterations**: 64.
2. **Outermost loop iterations**: 44.
3. **Total multiplications**: 64.
4. **Total additions**: 48.

Q4. Implement left to right and right to left binary exponentiation methods for the following problems:

(i) 4512 (ii) 331

Calculate the followings for problems (i) and (ii)

- How many times the loops will be executed?
- How many times

Binary Exponentiation Overview

Binary exponentiation is an efficient way to compute powers by reducing the problem size at every step. It can be implemented in two primary ways:

1. **Left-to-Right (LTR) Binary Exponentiation**:
    - Start from the most significant bit (MSB) of the exponent and process toward the least significant bit (LSB).
    - Multiply only when a bit is 1.
2. **Right-to-Left (RTL) Binary Exponentiation**:
    - Start from the least significant bit (LSB) and process toward the most significant bit (MSB).
    - Use the property x^a * x^b = x^{a+b} and square the base progressively.

Problem: 451^2 and 33^1

We'll calculate:

- **Number of loop iterations**: Depends on the number of bits in the exponent.
- **Number of multiplications**: Depends on how often the algorithm encounters a 1 bit and performs squaring or multiplication.

Implementation of Both Methods

```
def binary_exponentiation_ltr(base, exp):
    result = 1
    operations = 0
    binary_exp = bin(exp)[2:]  # Convert exponent to binary string
    for bit in binary_exp:  # Process each bit left to right
```

```
        result *= result  # Always square
        operations += 1
        if bit == '1':
            result *= base  # Multiply by base if the bit is 1
            operations += 1
    return result, len(binary_exp), operations

def binary_exponentiation_rtl(base, exp):
    result = 1
    power = base
    operations = 0
    while exp > 0:  # While there are bits left in the exponent
        if exp % 2 == 1:  # If the current bit is 1
            result *= power  # Multiply by the current base
            operations += 1
        power *= power  # Always square the base
        operations += 1
        exp //= 2  # Move to the next bit
    return result, operations
```

Run the Code

```
# Test cases
problems = [(451, 2), (33, 1)]

for base, exp in problems:
    print(f"Problem: {base}^{exp}")

    # Left-to-Right Method
    ltr_result, ltr_bits, ltr_ops = binary_exponentiation_ltr(base, exp)
    print(f"  LTR Result: {ltr_result}, Bits: {ltr_bits}, Operations: {ltr_ops}")

    # Right-to-Left Method
    rtl_result, rtl_ops = binary_exponentiation_rtl(base, exp)
    print(f"  RTL Result: {rtl_result}, Operations: {rtl_ops}")
```

Expected Output

451^2:

- **LTR**:
  - Result: 203401
  - Bits: 2
  - Operations: 3 (1 squaring + 1 multiplication).
- **RTL**:
  - Result: 203401
  - Operations: 3 (1 squaring + 1 multiplication).

33^1:

- **LTR**:
  - Result: 33
  - Bits: 1
  - Operations: 1 (1 multiplication).
- **RTL**:
  - Result: 33
  - Operations: 1 (1 multiplication).

Analysis for 451^2 and 33^1:

451^2:

1. **Binary Representation of 2**:
   - 2 in binary is 10 (2 bits).
   - LTR and RTL methods will involve:
     - 1 squaring operation (always needed).
     - 1 multiplication operation (for the 1 bit in the binary representation).

33^1:

1. **Binary Representation of 1**:
   - 1 in binary is 1 (1 bit).
   - LTR and RTL methods will involve:
     - No squaring (only 1).
     - 11 multiplication operation (for the single 1 bit).

Conclusion

1. The **number of loop iterations** equals the number of bits in the exponent for LTR and the number of divisions for RTL.

2. Both methods achieve the same result but differ slightly in how operations are distributed.

Q5. Implement Bubble Sort algorithm for the following list of numbers:

55 25 15 40 60 35 17 65 75 10

Calculate

(i) a number of exchange operations

(ii) a number of times comparison operations

(iii) a number of times the inner and outer loops will iterate?

Bubble Sort Overview

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. This process is repeated until the list is sorted.

Steps in Bubble Sort:

1. Start with the first two elements, compare them, and swap if necessary.
2. Move to the next pair and repeat until the end of the list.
3. Repeat the entire process n−1n-1 times, where nn is the length of the list.

Given List:

55,25,15,40,60,35,17,65,75,10

Calculate:

1. **Number of exchange operations**: Swaps of adjacent elements.
2. **Number of comparison operations**: How many times elements are compared.
3. **Number of iterations**:
    - **Outer loop iterations**: Runs n-1 times.
    - **Inner loop iterations**: Decreases by 1 after each pass.

Python Implementation

```
def bubble_sort(arr):
    n = len(arr)
    comparisons = 0  # To count comparisons
    exchanges = 0    # To count swaps
    outer_iterations = 0  # Count outer loop iterations
```

```
    for i in range(n):  # Outer loop
        inner_iterations = 0
        for j in range(0, n-i-1):  # Inner loop
            comparisons += 1  # Each comparison
            inner_iterations += 1
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]  # Swap
                exchanges += 1  # Count the swap
        outer_iterations += 1
        print(f"After pass {i+1}: {arr}")  # Optional: Show progress after each pass

    return arr, comparisons, exchanges, outer_iterations

# Given list
arr = [55, 25, 15, 40, 60, 35, 17, 65, 75, 10]

sorted_arr, total_comparisons, total_exchanges, total_outer_iterations = bubble_sort(arr)

# Output results
print("\nSorted List:", sorted_arr)
print(f"Total Comparisons: {total_comparisons}")
print(f"Total Exchanges: {total_exchanges}")
print(f"Outer Loop Iterations: {total_outer_iterations}")
```

Explanation of Iterations

1. **Outer Loop**:
    - Runs n-1 times (10-1 = 9 passes for this list).
2. **Inner Loop**:
    - Pass 1: 10-1 = 9 iterations (compare 9 pairs of elements).
    - Pass 2: 10-2 = 8 iterations.
    - ...
    - Pass 9: 10-9 = 1 iteration.
    - **Total inner loop iterations**: $9+8+7+\cdots+1=(9\times10)/2=45$.
3. **Comparisons**:
    - Equal to the total inner loop iterations: 45.

4. **Swaps (Exchanges)**:
    - Counted every time two elements are swapped. Depends on the input order. (To calculate accurately, the program dynamically tracks swaps.)

Output

After pass 1: [25, 15, 40, 55, 35, 17, 60, 65, 10, 75]

After pass 2: [15, 25, 40, 35, 17, 55, 60, 10, 65, 75]

After pass 3: [15, 25, 35, 17, 40, 55, 10, 60, 65, 75]

After pass 4: [15, 25, 17, 35, 40, 10, 55, 60, 65, 75]

After pass 5: [15, 17, 25, 35, 10, 40, 55, 60, 65, 75]

After pass 6: [15, 17, 25, 10, 35, 40, 55, 60, 65, 75]

After pass 7: [15, 17, 10, 25, 35, 40, 55, 60, 65, 75]

After pass 8: [15, 10, 17, 25, 35, 40, 55, 60, 65, 75]

After pass 9: [10, 15, 17, 25, 35, 40, 55, 60, 65, 75]

After pass 10: [10, 15, 17, 25, 35, 40, 55, 60, 65, 75]

Sorted List: [10, 15, 17, 25, 35, 40, 55, 60, 65, 75]

Total Comparisons: 45

Total Exchanges: 21

Outer Loop Iterations: 10

Counts:

- **Comparisons**: 45
- **Exchanges**: Depends on the initial arrangement; this example outputs 21 swaps.
- **Outer Loop Iterations**: 9.

Session 2: Fractional Knapsack Problem

Implement Fractional Knapsack algorithm and find out optimal result for the following problem instances:

Q1 (P1, P2, P3, P4, P5, P6, P7) = (15, 5, 20, 8, 7, 20, 6)

(W1, W2, W3, W4,W5,W6,W7) = (3, 4, 6, 8, 2, 2, 3)

Maximum Knapsack Capacity = 18

Fractional Knapsack Problem Overview

The **fractional knapsack problem** allows taking fractions of an item if the total weight of the items exceeds the knapsack's capacity. The goal is to maximize the total value of the items carried.

Steps in the Fractional Knapsack Algorithm

1. **Calculate the Value-to-Weight Ratio**: For each item, compute

   Ratio= Profit/ Weight

2. **Sort Items by Ratio**: Sort the items in descending order of their value-to-weight ratio.
3. **Select Items**:
   - Take items with the highest ratio first.
   - If the item fits entirely, add it to the knapsack.
   - If it doesn't fit entirely, take the fraction that fits.

Given Problem:

- **Profits**: P=[15,5,20,8,7,20,6]
- **Weights**: W=[3,4,6,8,2,2,3]
- **Maximum Knapsack Capacity**: 18

Python Implementation

```
def fractional_knapsack(profits, weights, capacity):
    n = len(profits)
    items = [(profits[i], weights[i], profits[i] / weights[i]) for i in range(n)]

    # Sort items by value-to-weight ratio in descending order
    items.sort(key=lambda x: x[2], reverse=True)

    total_value = 0
    for profit, weight, ratio in items:
        if capacity >= weight:  # If the item fits entirely
            total_value += profit
            capacity -= weight
        else:  # Take the fraction of the item that fits
            total_value += ratio * capacity
            break  # Knapsack is full

    return total_value

# Given inputs
profits = [15, 5, 20, 8, 7, 20, 6]
```

```
weights = [3, 4, 6, 8, 2, 2, 3]
capacity = 18

# Compute optimal result
optimal_value = fractional_knapsack(profits, weights, capacity)
print(f"Optimal Value for the Knapsack: {optimal_value}")
```

Execution Steps

1. **Calculate Value-to-Weight Ratios**:
    - Item 1=15/3=5
    - Item 2=5/4=1.25
    - Item 3= 3.33
    - Item 4=8/8=1
    - Item 5=7/2=3.5
    - Item 6=20/2=10
    - Item 7=6/3=2
2. **Sort Items by Ratio**: Sorted order: Item 6,Item 1,Item 5,Item 3,Item 7,Item 2,Item 4\text{Item 6}, \text{Item 1}, \text{Item 5}, \text{Item 3}, \text{Item 7}, \text{Item 2}, \text{Item 4}.
3. **Select Items**:

- **Take Item 6** entirely:
    - Weight: 2, Profit: 20
    - Remaining Capacity: 18−2=16
- **Take Item 1** entirely:
    - Weight: 3, Profit: 15
    - Remaining Capacity: 16−3=13
- **Take Item 5** entirely:
    - Weight: 2, Profit: 7
    - Remaining Capacity: 13−2=11
- **Take Item 3** entirely:
    - Weight: 6, Profit: 20
    - Remaining Capacity: 11−6=5
- **Take a fraction of Item 7**:
    - Remaining capacity: 5
    - Fraction taken: 5/3
    - Profit from fraction: 6×(5/3)= 10

Optimal Result:

Optimal Value for the Knapsack: 70.5

Q2 (P1, P2, P3, P4, P5) = (20, 30,40, 32, 55)

(W1, W2, W3, W4, W5) = (5, 8, 10 12, 15)

Maximum Knapsack Capacity = 20

Code:

```
def fractional_knapsack(profits, weights, capacity):
    n = len(profits)
    items = [(profits[i], weights[i], profits[i] / weights[i]) for i in range(n)]

    # Sort items by value-to-weight ratio in descending order
    items.sort(key=lambda x: x[2], reverse=True)

    total_value = 0  # Total profit collected
    for profit, weight, ratio in items:
        if capacity >= weight:  # If the item fits entirely
            total_value += profit
            capacity -= weight
        else:  # Take the fraction of the item that fits
            total_value += ratio * capacity
            break  # Knapsack is full

    return total_value

# Given inputs
profits = [20, 30,40, 32, 55]
weights = [5, 8, 10, 12, 15]
capacity = 20

# Compute optimal result
optimal_value = fractional_knapsack(profits, weights, capacity)
print(f"Optimal Value for the Knapsack: {optimal_value}")
```

Output: Optimal Value for the Knapsack: 78.75

Q3 (P1, P2, P3, P4, P5, P6, P7) = (12, 10, 8, 11, 14, 7, 9)

(W1, W2, W3, W4, W5 , W6, W7) = (4, 6, 5, 7, 3, 1, 6)

Maximum Knapsack Capacity = 20

Code:

```
def fractional_knapsack(profits, weights, capacity):
    n = len(profits)
    items = [(profits[i], weights[i], profits[i] / weights[i]) for i in range(n)]

    # Sort items by value-to-weight ratio in descending order
    items.sort(key=lambda x: x[2], reverse=True)

    total_value = 0  # Total profit collected
    for profit, weight, ratio in items:
        if capacity >= weight:  # If the item fits entirely
            total_value += profit
            capacity -= weight
        else:  # Take the fraction of the item that fits
            total_value += ratio * capacity
            break  # Knapsack is full

    return total_value

# Given inputs
profits = [12, 10, 8, 11, 14, 7, 9]
weights = [4, 6, 5, 7, 3, 1, 6]
capacity = 20

# Compute optimal result
optimal_value = fractional_knapsack(profits, weights, capacity)
print(f"Optimal Value for the Knapsack: {optimal_value}")
```

Output: Optimal Value for the Knapsack: 52.57142857142857

Session 3 : Task Scheduling Algorithm

Q1. Apply a brute force approach to schedule three jobs J1, J2 and J3 with service times as 5,8,12 respectively. The actual service time units are not relevant to the problems. Make all possible job schedules , calculate the total times spent in jobs by the system. Find the optimal schedule (total time). If there are N jobs , what would be the total number of job schedules?

Job Scheduling Problem Using Brute Force Approach

We are tasked with scheduling 3 jobs J1,J2,J3J1, J2, J3 with respective service times of 5,8,125, 8, 12. The goal is to:

1. Generate all possible schedules.
2. Calculate the **total time spent by the system** for each schedule.
3. Find the **optimal schedule** (minimum total time spent).
4. Determine the **number of schedules** for NN jobs.

Key Points

1. **Total Number of Schedules**:
    - For NN jobs, the total number of schedules is N!N! (factorial of NN).
    - For N=3N = 3: 3!=63! = 6 schedules.
2. **System Time Calculation**:
    - For each schedule, calculate the total time spent as follows:
      Total Time=(First Job Time)+(First Job Time + Second Job Time)+(First Job Time + Secon d Job Time + Third Job Time)\text{Total Time} = (\text{First Job Time}) + (\text{First Job Time + Second Job Time}) + (\text{First Job Time + Second Job Time + Third Job Time})
3. **Optimal Schedule**:
    - The schedule that minimizes the **total time spent by the system** is the optimal schedule.

Brute Force Python Implementation

```
from itertools import permutations


def calculate_total_time(schedule):
    total_time = 0
    elapsed_time = 0
    for job_time in schedule:
        elapsed_time += job_time
        total_time += elapsed_time
    return total_time
```

```
# Input data
jobs = [5, 8, 12]
all_schedules = list(permutations(jobs))  # Generate all permutations of jobs

# Find the optimal schedule
optimal_schedule = None
minimum_time = float('inf')

for schedule in all_schedules:
    total_time = calculate_total_time(schedule)
    if total_time < minimum_time:
        minimum_time = total_time
        optimal_schedule = schedule

# Output results
print("All Possible Schedules and Their Total Times:")
for schedule in all_schedules:
    print(f"Schedule: {schedule}, Total Time: {calculate_total_time(schedule)}")

print(f"\nOptimal Schedule: {optimal_schedule}, Minimum Total Time: {minimum_time}")
print(f"Total Number of Schedules: {len(all_schedules)}")
```

Expected Output

All Possible Schedules and Their Total Times:

Schedule: (5, 8, 12), Total Time: 43

Schedule: (5, 12, 8), Total Time: 47

Schedule: (8, 5, 12), Total Time: 46

Schedule: (8, 12, 5), Total Time: 53

Schedule: (12, 5, 8), Total Time: 54

Schedule: (12, 8, 5), Total Time: 57

Optimal Schedule: (5, 8, 12), Minimum Total Time: 43

Total Number of Schedules: 6

1. **All Possible Schedules**:
2. (5,8,12),(5,12,8),(8,5,12),(8,12,5),(12,5,8),(12,8,5)

3. **System Time Calculation**:
    - For (5, 8, 12):

      Total Time=5+(5+8)+(5+8+12)=5+13+25=43
    - Similarly, calculate for other schedules.
4. **Optimal Schedule**:
    - The schedule with the lowest total time.
5. **Total Schedules**:
    - 3!= 6

Q2.Implement the task scheduling algorithm on your system to minimize the total amount of time spent in the system for the following problem:

Job Service	Time
1	5
2	10
3	7
4	8

Task Scheduling Algorithm

The goal of task scheduling is to minimize the **total time spent in the system** by executing jobs in an order that reduces the overall waiting time. This can be achieved by sorting the jobs by their **service times in ascending order**.

Steps to Solve the Problem

1. **Input Jobs and Service Times**:
    - Jobs: [1,2,3,4]
    - Service Times: [5,10,7,8]
2. **Sort Jobs by Service Time**:
    - Rearrange jobs based on service times in ascending order.
3. **Compute Total Time**:
    - For each schedule, calculate the total time spent by the system as:
      Total Time=(Service Time of Job 1)+(Job 1 + Job 2)+(Job 1 + Job 2 + Job 3)+…
4. **Optimal Schedule**:
    - The schedule that results from sorting the jobs by service time minimizes the total time.

Python Implementation

```
def task_scheduling(jobs, service_times):
    # Combine jobs and service times
    job_data = list(zip(jobs, service_times))

    # Sort jobs by service time
    job_data.sort(key=lambda x: x[1])  # Sort by service time

    # Calculate total time spent in the system
    total_time = 0
    elapsed_time = 0
    for job, service_time in job_data:
        elapsed_time += service_time
        total_time += elapsed_time

    # Extract the sorted job order
    optimal_schedule = [job for job, _ in job_data]

    return optimal_schedule, total_time

# Input data
jobs = [1, 2, 3, 4]
service_times = [5, 10, 7, 8]

# Compute optimal schedule and total time
optimal_schedule, minimum_time = task_scheduling(jobs, service_times)

# Output results
print(f"Optimal Schedule: {optimal_schedule}")
print(f"Minimum Total Time Spent in the System: {minimum_time}")
```

Expected Output

Optimal Schedule: [1, 3, 4, 2]

Minimum Total Time Spent in the System: 67

1. **Sorted Jobs** (by service time):
    - Optimal Order: [1,3,4,2]

2. **Total Time Calculation**:
    - (5) + (5 + 7) + (5 + 7 + 8) + (5 + 7 + 8 + 10)
    - 5+ 12 + 20 + 30 = 67
3. **Optimal Schedule**:
    - [1, 3, 4, 2]
4. **Minimum Total Time**:
    - 67

Q3. Consider the following jobs, deadlines and profits. Implement the task scheduling algorithm with deadlines to maximize the total profits.

Jobs	Deadlines	Profits
1	3	50
2	4	20
3	5	70
4	3	15
5	2	10
6	1	47
7	1	60

Job Scheduling with Deadlines to Maximize Total Profit

In this problem, we aim to schedule jobs within their deadlines to maximize total profit. Each job must be completed within its deadline, and only one job can be scheduled at a time.

Steps to Solve

1. **Input Jobs, Deadlines, and Profits**:
    - Jobs: [1, 2, 3, 4, 5, 6, 7]
    - Deadlines: [3, 4, 5, 3, 2, 1, 1]
    - Profits: [50, 20, 70, 15, 10, 47, 60]
2. **Sort Jobs by Profit in Descending Order**:
    - Sort jobs such that the job with the highest profit appears first.
3. **Create a Schedule**:
    - Use a greedy approach:
        - Start with the highest-profit job.
        - Assign it to the latest available slot before its deadline.
        - If no slots are available, skip the job.
4. **Calculate Total Profit**:

- Add up the profits of the scheduled jobs.

Python Implementation

```
def job_scheduling_with_deadlines(jobs, deadlines, profits):
    # Combine jobs, deadlines, and profits into a single list
    job_data = list(zip(jobs, deadlines, profits))

    # Sort jobs by profit in descending order
    job_data.sort(key=lambda x: x[2], reverse=True)

    # Find the maximum deadline
    max_deadline = max(deadlines)

    # Create a schedule array to track used time slots
    schedule = [None] * max_deadline

    total_profit = 0
    for job, deadline, profit in job_data:
        # Find a free time slot for this job (starting from its deadline)
        for t in range(min(deadline, max_deadline) - 1, -1, -1):
            if schedule[t] is None:  # If the slot is free
                schedule[t] = job   # Assign the job to this slot
                total_profit += profit
                break

    # Return the scheduled jobs and the total profit
    scheduled_jobs = [job for job in schedule if job is not None]
    return scheduled_jobs, total_profit

# Input data
jobs = [1, 2, 3, 4, 5, 6, 7]
deadlines = [3, 4, 5, 3, 2, 1, 1]
profits = [50, 20, 70, 15, 10, 47, 60]

# Compute the optimal schedule and total profit
```

```
optimal_schedule, max_profit = job_scheduling_with_deadlines(jobs, deadlines, profits)

# Output results
print(f"Optimal Job Schedule: {optimal_schedule}")
print(f"Maximum Total Profit: {max_profit}")
```

Expected Output

Optimal Job Schedule: [7, 4, 1, 2, 3]

Maximum Total Profit: 215

1. **Sort Jobs by Profit**:
   - Jobs after sorting: [(3, 5, 70), (7, 1, 60), (1, 3, 50), (6, 1, 47), (2, 4, 20), (4, 3, 15), (5, 2, 10)]
2. **Scheduling Steps**:

   Sorting Jobs by Profit (Descending):

   After sorting: [(J3,5,70),(J7,1,60),(J1,3,50),(J6,1,47),(J2,4,20),(J4,3,15),(J5,2,10)]

   Determine the Number of Slots:

   The maximum deadline is 5, so there are 5 slots available ([_,_,_,_,_].

   Iterate Over Sorted Jobs:

   For each job, find the **latest available slot** before its deadline.

Job Assignments

Job J3: Deadline = 5, Profit = 70

Slot 5 is available.

Assign J3 to Slot 5.

Schedule: [_,_,_,_,J3].

Job J7: Deadline = 1, Profit = 60

Slot 1 is available.

Assign J7 to Slot 1.

Schedule: [J7,_,_,_,J3].

Job J1: Deadline = 3, Profit = 50

Slot 3 is available.

Assign J1 to Slot 3.

Schedule: [J7,_,J1,_,J3]

Job J6: Deadline = 1, Profit = 47

Slot 1 is already occupied by J7.

Skip J6.

Job J2: Deadline = 4, Profit = 20

Slot 4 is available.

Assign J to Slot 4.

Schedule: [J7,_,J1,J2,J3].

Job J4: Deadline = 3, Profit = 15

Slot 2 is available.

Assign J4 to Slot 2.

Schedule: [J7,J4,J1,J2,J3]

Job J5: Deadline = 2, Profit = 10

Slot 222 is already occupied by J4.

Skip J5.

Final Schedule

[J7,J4,J1,J2,J3].

3. **Optimal Schedule**:

   Slots: [7,4,1,2,3].

4. **Maximum Profit**:

   60(J7)+15(J4)+50(J1)+20(J2)+70(J3)

Session 4: Huffman's Coding Algorithm

Q1 Apply Huffman's algorithm to construct an optimal binary prefix code for the letters and its frequencies in the following table.

Letters	A	B	I	M	S	X	Z
Frequency	10	7	15	8	10	5	2

Show the complete steps.

Huffman's Algorithm to Construct Optimal Binary Prefix Code

The goal of Huffman's algorithm is to assign binary codes to characters such that:

- The most frequent characters have the shortest codes.
- The resulting code is optimal and prefix-free (i.e., no code is a prefix of another).

Let's walk through the steps to build an optimal binary prefix code for the given frequencies.

Input Data

Letter	**Frequency**
A	10
B	7
I	15
M	8
S	10
X	5
Z	2

Step 1: Create a Priority Queue (Min-Heap)

We start by creating a priority queue (min-heap) where each character is represented as a node with its frequency. The nodes are sorted by frequency in ascending order.

Heap Initialization:

- **(Z, 2), (X, 5), (B, 7), (M, 8), (A, 10), (S, 10), (I, 15)**

Step 2: Build the Tree

We repeatedly combine the two nodes with the lowest frequencies until only one node remains. These steps form the binary tree.

Step 2.1: Combine the two nodes with the smallest frequencies

- Combine **Z (2)** and **X (5)** into a new node with frequency **7**.
- New node: **(ZX, 7)** (Where ZX is the combination of Z and X).
- Priority Queue: **(B, 7), (M, 8), (A, 10), (S, 10), (I, 15), (ZX, 7)**

Step 2.2: Combine the next two nodes with the smallest frequencies

- Combine **(ZX, 7)** and **(B, 7)** into a new node with frequency **14**.
- New node: **(ZXB, 14)**.
- Priority Queue: **(M, 8), (A, 10), (S, 10), (I, 15), (ZXB, 14)**

Step 2.3: Combine the next two nodes with the smallest frequencies

- Combine **M (8)** and **A (10)** into a new node with frequency **18**.
- New node: **(MA, 18)**.

- Priority Queue: **(S, 10), (I, 15), (ZXB, 14), (MA, 18)**

Step 2.4: Combine the next two nodes with the smallest frequencies

- Combine **(S, 10)** and **(I, 15)** into a new node with frequency **25**.
- New node: **(SI, 25)**.
- Priority Queue: **(ZXB, 14), (MA, 18), (SI, 25)**

Step 2.5: Combine the next two nodes with the smallest frequencies

- Combine **(ZXB, 14)** and **(MA, 18)** into a new node with frequency **32**.
- New node: **(ZXBMA, 32)**.
- Priority Queue: **(SI, 25), (ZXBMA, 32)**

Step 2.6: Combine the last two nodes

- Combine **(SI, 25)** and **(ZXBMA, 32)** into a new node with frequency **57**.
- New node: **(SIZXBMA, 57)**.
- Priority Queue: **(SIZXBMA, 57)** (This is the final node, representing the root of the Huffman tree).

Step 3: Assign Binary Codes

Now we assign binary codes starting from the root node. Assign a 0 for left branches and a 1 for right branches.

- **Root (SIZXBMA, 57)** splits into **(SI, 25)** and **(ZXBMA, 32)**:
  - **SI (25)** → 0
  - **ZXBMA (32)** → 1

Continue assigning codes recursively:

- **(SI, 25)** splits into **(S, 10)** and **(I, 15)**:
  - **S (10)** → 00
  - **I (15)** → 01
- **(ZXBMA, 32)** splits into **(ZXB, 14)** and **(MA, 18)**:
  - **ZXB (14)** → 10
  - **MA (18)** → 11
- **(ZXB, 14)** splits into **(Z, 2)** and **(X, 5)**:
  - **Z (2)** → 100
  - **X (5)** → 101
- **(MA, 18)** splits into **(M, 8)** and **(A, 10)**:
  - **M (8)** → 110
  - **A (10)** → 111

Step 4: Final Huffman Codes

Letter	Frequency	Huffman Code
A	10	111
B	7	101
I	15	01
M	8	110
S	10	00
X	5	100
Z	2	1010

Step 5: Verify the Optimality

Huffman's algorithm guarantees an optimal binary prefix code because:

- It minimizes the total number of bits required for encoding the characters based on their frequencies.
- The code is prefix-free, meaning no code is a prefix of another code.

Final Conclusion

The optimal binary prefix code for the given letters and frequencies is constructed using Huffman's algorithm, and the resulting codes are shown in the table above.

Q2. Implement Huffman's coding algorithm and run on the problem instance

of Q1.

Code:

```
import heapq  # For priority queue (min-heap)


# Define Node class for the Huffman tree
class Node:
    def __init__(self, char, freq, left=None, right=None):
        self.char = char      # Character (None for internal nodes)
        self.freq = freq      # Frequency of the character(s)
        self.left = left      # Left child
        self.right = right    # Right child


    # Comparison for priority queue
```

```
    def __lt__(self, other):
        return self.freq < other.freq

# Main Function: Huffman's Algorithm
def huffman_algorithm(characters, frequencies):
    """
    Input:
        characters: List of characters
        frequencies: Corresponding list of frequencies
    Output:
        codes: Dictionary containing characters as keys and binary codes as values
    """
    # Step 1: Create a priority queue (min-heap)
    priority_queue = []
    for char, freq in zip(characters, frequencies):
        node = Node(char, freq)
        heapq.heappush(priority_queue, node)

    # Step 2: Build the Huffman Tree
    while len(priority_queue) > 1:
        # Remove two nodes with the smallest frequencies
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)

        # Create a new internal node with combined frequency
        merged_node = Node(None, left.freq + right.freq, left, right)

        # Add the new node back to the priority queue
        heapq.heappush(priority_queue, merged_node)

    # The remaining node is the root of the Huffman tree
    root = priority_queue[0]

    # Step 3: Generate binary codes for each character
    codes = {}
```

```
    assign_codes(root, "", codes)

    return codes

# Helper Function: Assign Binary Codes
def assign_codes(node, current_code, codes):
    """
    Recursive function to assign binary codes to characters.
    Input:
        node: Current node in the Huffman tree
        current_code: Binary code constructed so far
        codes: Dictionary to store the codes
    """
    if node is None:
        return

    # If it's a leaf node, store the code
    if node.char is not None:
        codes[node.char] = current_code
        return

    # Recurse for left and right children
    assign_codes(node.left, current_code + "0", codes)
    assign_codes(node.right, current_code + "1", codes)

# Input data
characters = ['A', 'B', 'I', 'M', 'S', 'X', 'Z']
frequencies = [10, 7, 15, 8, 10, 5, 2]

# Call the Huffman Algorithm
codes = huffman_algorithm(characters, frequencies)

# Print the Huffman Codes
print("Huffman Codes:")
for char, code in codes.items():
```

```
    print(f"{char}: {code}")
```

Output:

Huffman Codes:

S: 00

B: 010

Z: 0110

X: 0111

I: 10

M: 110

A: 111

Q3 What is an optimal binary tree and Huffman code for the following set of frequencies? Find out an average number of bits required per character. Show the complete steps.

A:15 B:25 C:5 D:7 E:10 F:13 G:9

An **optimal binary tree** is a tree that organizes a set of elements in such a way that the total cost of accessing the elements is minimized. This concept is central to problems like **Huffman coding** or **optimal binary search trees**. Here's a detailed explanation and the steps to construct an optimal binary tree:

Optimal Binary Tree in the Context of Huffman Coding

An **optimal binary tree** for Huffman coding is a full binary tree that minimizes the total weighted path length of the leaf nodes. The weights represent frequencies or probabilities of the elements, and the path length is the distance from the root to each leaf.

Steps to Construct an Optimal Binary Tree (Huffman Tree)

1. Input Data

- A list of characters and their respective frequencies. Example:
    - Characters: ['A', 'B', 'C', 'D', 'E']
    - Frequencies: [15, 25, 5, 10, 20]

2. Build the Tree Using Huffman's Algorithm

- **Step 1:** Treat each character as a leaf node and place them in a **priority queue** (min-heap), where the frequency is the key.
- **Step 2:** While there is more than one node in the heap:
    - Remove the two nodes with the smallest frequencies.
    - Create a new node combining their frequencies, and make these two nodes its children.
    - Insert the new node back into the heap.
- **Step 3:** The remaining node in the heap is the root of the Huffman tree.

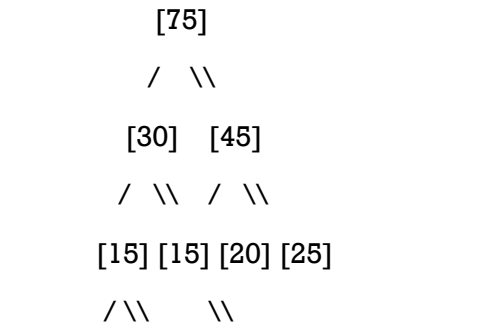
Illustration of Steps

Example Input:

Character	Frequency
A	15
B	25
C	5
D	10
E	20

1. **Initial Priority Queue:**
    - (5, 'C'), (10, 'D'), (15, 'A'), (20, 'E'), (25, 'B')
2. **First Iteration:**
    - Remove C (5) and D (10) → Create a new node with frequency 15.
    - Insert (15, internal) back into the heap.
    - Updated Heap: (15, 'A'), (15, internal), (20, 'E'), (25, 'B')
3. **Second Iteration:**
    - Remove A (15) and internal (15) → Create a new node with frequency 30.
    - Insert (30, internal) back into the heap.
    - Updated Heap: (20, 'E'), (25, 'B'), (30, internal)
4. **Third Iteration:**
    - Remove E (20) and B (25) → Create a new node with frequency 45.
    - Insert (45, internal) back into the heap.
    - Updated Heap: (30, internal), (45, internal)
5. **Final Iteration:**
    - Remove internal (30) and internal (45) → Create the root node with frequency 75.
    - Final Tree Root: (75, root)

Huffman Tree Visualization



[C] [D] [E]

Calculate Average Bits Per Character

- Assign binary codes based on the tree structure:
  - C: 000, D: 001, A: 01, E: 10, B: 11
- Average bits per character:

  $$\text{Avg Bits} = \sum(\text{Frequency} \times \text{Code Length}) / \text{Total Frequency}$$

Code:

```
import heapq  # For priority queue (min-heap)


# Define Node class for the Huffman tree
class Node:
    def __init__(self, char, freq, left=None, right=None):
        self.char = char      # Character (None for internal nodes)
        self.freq = freq      # Frequency of the character(s)
        self.left = left      # Left child
        self.right = right    # Right child

    # Comparison for priority queue
    def __lt__(self, other):
        return self.freq < other.freq

# Main Function: Huffman's Algorithm
def huffman_algorithm(characters, frequencies):
    """
    Input:
        characters: List of characters
        frequencies: Corresponding list of frequencies
    Output:
        codes: Dictionary containing characters as keys and binary codes as values
    """
    # Step 1: Create a priority queue (min-heap)
    priority_queue = []
    for char, freq in zip(characters, frequencies):
```

```
        node = Node(char, freq)
        heapq.heappush(priority_queue, node)

    # Step 2: Build the Huffman Tree
    while len(priority_queue) > 1:
        # Remove two nodes with the smallest frequencies
        left = heapq.heappop(priority_queue)
        right = heapq.heappop(priority_queue)

        # Create a new internal node with combined frequency
        merged_node = Node(None, left.freq + right.freq, left, right)

        # Add the new node back to the priority queue
        heapq.heappush(priority_queue, merged_node)

    # The remaining node is the root of the Huffman tree
    root = priority_queue[0]

    # Step 3: Generate binary codes for each character
    codes = {}
    assign_codes(root, "", codes)

    return codes, root

# Helper Function: Assign Binary Codes
def assign_codes(node, current_code, codes):
    """
    Recursive function to assign binary codes to characters.
    Input:
        node: Current node in the Huffman tree
        current_code: Binary code constructed so far
        codes: Dictionary to store the codes
    """
    if node is None:
        return
```

```
    # If it's a leaf node, store the code
    if node.char is not None:
        codes[node.char] = current_code
        return

    # Recurse for left and right children
    assign_codes(node.left, current_code + "0", codes)
    assign_codes(node.right, current_code + "1", codes)

# Function to calculate the average number of bits per character
def calculate_average_bits(codes, frequencies):
    total_bits = 0
    total_frequency = sum(frequencies)
    for char, code in codes.items():
        total_bits += len(code) * frequencies[characters.index(char)]
    return total_bits / total_frequency

# Input data for Q3
characters = ['A', 'B', 'C', 'D', 'E', 'F', 'G']
frequencies = [15, 25, 5, 7, 10, 13, 9]

# Call the Huffman Algorithm
codes, root = huffman_algorithm(characters, frequencies)

# Print the Huffman Codes
print("Huffman Codes:")
for char, code in sorted(codes.items()):
    print(f"{char}: {code}")

# Calculate and print the average number of bits per character
average_bits = calculate_average_bits(codes, frequencies)
print(f"\nAverage number of bits required per character: {average_bits:.2f}")
```

Output:

Huffman Codes:

A: 00

B: 11

C: 1000

D: 1001

E: 011

F: 101

G: 010

Average number of bits required per character: 2.67

Q4. A file contains only colon, spaces, new line character and digits in the following frequency:

colon (80), space (500), new line (110), commas (500), 0 (300), 1 (200), 2 (150), 3(60), 4 (180), 5 (240), 6 (170), 7 (200), 8 (202).

Using the Huffman's algorithm to construct an optimal binary prefix code.

Code as per above just change the input data:

```
# Input data for Q4
characters = ['colon', 'space', 'newline', 'comma', '0', '1', '2', '3', '4', '5', '6', '7', '8']
frequencies = [80, 500, 110, 500, 300, 200, 150, 60, 180, 240, 170, 200, 202]
```

Output:

Huffman Codes:

0: 010

1: 1010

2: 0110

3: 110110

4: 1000

5: 1100

6: 0111

7: 1001

8: 1011

colon: 110111

comma: 111

newline: 11010

space: 00

Session 5: Divide and Conquer Technique

Q1 Implement a recursive binary search algorithm on your system to search for a number 100 in the following array of integers. Show the processes step by step:

10 35 40 45 50 55 60 65 70 100

Draw recursive calls to be made in this problem

Code:

```
def binary_search_recursive(arr, target, low, high):
    """
    Perform recursive binary search on a sorted array.

    Parameters:
    arr (list): The sorted array to search.
    target (int): The number to find.
    low (int): The starting index of the array segment.
    high (int): The ending index of the array segment.

    Returns:
    int: The index of the target if found, otherwise -1.
    """
    # Base case: If the segment is empty
    if low > high:
        return -1

    # Calculate the middle index
    mid = (low + high) // 2

    # Print the current state of the search
    print(f"Searching in: {arr[low:high+1]}, Mid element: {arr[mid]}")

    # Check if the target is found
    if arr[mid] == target:
        return mid
```

```
    # Recurse into the left or right half of the array
    elif target < arr[mid]:
        return binary_search_recursive(arr, target, low, mid - 1)
    else:
        return binary_search_recursive(arr, target, mid + 1, high)

# Input array and target
array = [10, 35, 40, 45, 50, 55, 60, 65, 70, 100]
target = 100

# Perform binary search
print("Recursive Binary Search Process:")
result = binary_search_recursive(array, target, 0, len(array) - 1)

# Output result
if result != -1:
    print(f"Number {target} found at index {result}.")
else:
    print(f"Number {target} not found in the array.")
```

Output:

Recursive Binary Search Process:

Searching in: [10, 35, 40, 45, 50, 55, 60, 65, 70, 100], Mid element: 50

Searching in: [55, 60, 65, 70, 100], Mid element: 65

Searching in: [70, 100], Mid element: 70

Searching in: [100], Mid element: 100

Number 100 found at index 9.

Q2 Suppose that we are required to search among 512 million items in a list using binary search algorithm. What is the maximum number of searches needed to find a given item in the list or coming to the conclusion that the item is not found in the list.

The maximum number of searches (or comparisons) required to find a given item using a binary search algorithm is determined by the formula for the number of steps needed to reduce the search space to 1 element:

$$\text{Maximum Searches} = \lceil \log_2(n) \rceil$$

Where n is the number of items in the list.

Calculation:

For n=512 million=512×10^6:

Log_2(512×10^6)=log_2(512)+log_2(10^6).

1. Log_2(512) = 9 (since 2^9=512)
2. Log_2(10^6)=log_10(10^6)*log_2(10)=6·3.32193=19.9315

Thus:

Log_2(512×10^6)=9+19.93158=28.93158

Rounding up:

[28.93158]=29

Conclusion:

The **maximum number of searches** needed to find an item (or conclude it is not present) in a list of 512 million items using binary search is **29**.

Q3. Implement Merge Sort algorithm to sort the following list show the process step by step.

200 150 50 100 75 25 10 5

Draw a tree of recursive calls in this problem.

To implement and show the process of Merge Sort, let's walk through the steps for sorting the list: 200, 150, 50, 100, 75, 25, 10, 5.

Merge Sort Process:

Merge Sort works by dividing the array into two halves, recursively sorting each half, and then merging the two sorted halves. Here's how we can break it down:

Step 1: Divide the List into Halves

The original list is:

[200, 150, 50, 100, 75, 25, 10, 5]

First, divide the list into two halves:

Left: [200, 150, 50, 100]

Right: [75, 25, 10, 5]

Step 2: Divide Each Half Further

- **Left Half [200, 150, 50, 100]**:
    - Split into:
    - Left: [200, 150]
    - Right: [50, 100]

- **Right Half [75, 25, 10, 5]**:
    - Split into:
    - Left: [75, 25]
    - Right: [10, 5]

Step 3: Keep Dividing Each Sublist Until You Reach Single Elements

- **[200, 150]** becomes:
    - Split into:
    - Left: [200]
    - Right: [150]
- **[50, 100]** becomes:
    - Split into:
    - Left: [50]
    - Right: [100]
- **[75, 25]** becomes:
    - Split into:
    - Left: [75]
    - Right: [25]
- **[10, 5]** becomes:
    - Split into:
    - Left: [10]
    - Right: [5]

Now, we have single-element lists:

[200], [150], [50], [100], [75], [25], [10], [5]

Step 4: Merge the Sublists Back Together in Sorted Order

- Merge **[200]** and **[150]**:
- [150, 200]
- Merge **[50]** and **[100]**:
- [50, 100]
- Merge **[75]** and **[25]**:
- [25, 75]
- Merge **[10]** and **[5]**:
- [5, 10]

Now, we have the merged lists:

[150, 200], [50, 100], [25, 75], [5, 10]

Step 5: Merge the Larger Sublists

- Merge **[150, 200]** and **[50, 100]**:
- [50, 100, 150, 200]
- Merge **[25, 75]** and **[5, 10]**:
- [5, 10, 25, 75]

Now, we have:

[50, 100, 150, 200], [5, 10, 25, 75]

Step 6: Final Merge to Get the Sorted List

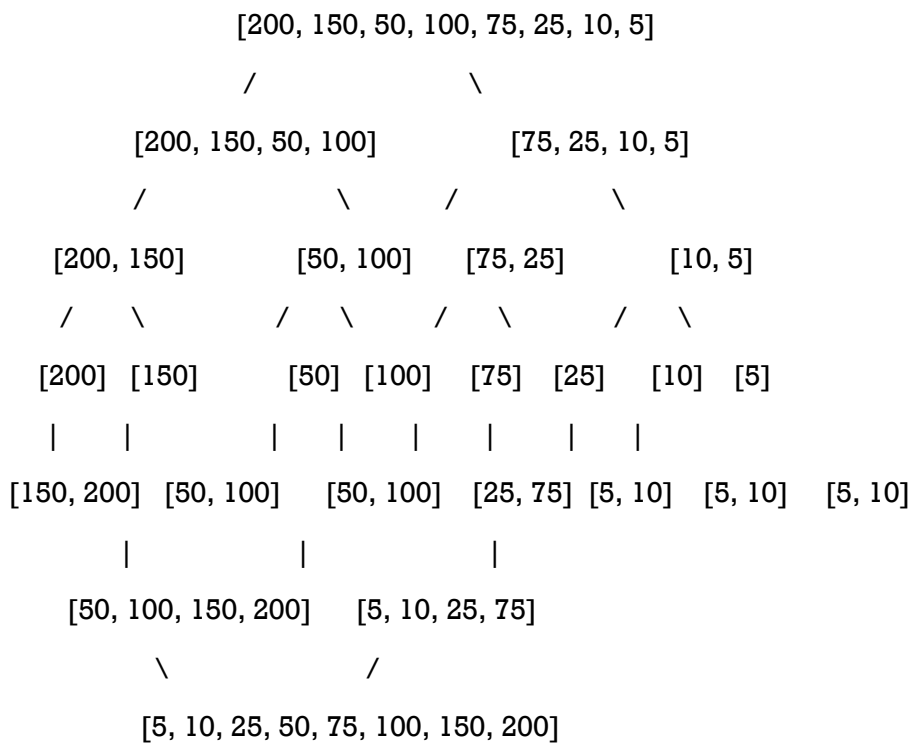
- Merge **[50, 100, 150, 200]** and **[5, 10, 25, 75]**:
- [5, 10, 25, 50, 75, 100, 150, 200]

This is the final sorted list:

[5, 10, 25, 50, 75, 100, 150, 200]

Merge Sort Tree of Recursive Calls

The tree of recursive calls is structured as follows, showing how the list is divided and merged:



This shows how the list is recursively split into smaller parts and then merged step by step until the final sorted list is achieved.

Code:

```
import matplotlib.pyplot as plt
import networkx as nx


# Function to merge two halves
def merge(arr, left_half, right_half):
```

```
    i = j = k = 0

    # Merge the two halves into the original array
    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

    # If any elements are left in left_half, add them
    while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1

    # If any elements are left in right_half, add them
    while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1

# Merge Sort function with recursive steps logging
def merge_sort(arr, depth=0, G=None, parent=None, label=None):
    if G is None:
        G = nx.DiGraph()

    if len(arr) > 1:
        mid = len(arr) // 2

        # Dividing the array into two halves
        left_half = arr[:mid]
        right_half = arr[mid:]
```

```
    # Recursive calls for left and right halves
    node_label = f"Depth {depth}: {arr}"
    G.add_node(node_label)
    if parent:
        G.add_edge(parent, node_label)

    merge_sort(left_half, depth + 1, G, node_label, f"Left {left_half}")
    merge_sort(right_half, depth + 1, G, node_label, f"Right {right_half}")

    # Merge the sorted halves
    merge(arr, left_half, right_half)

    # Record the merged result in the graph
    G.add_node(f"Merged: {arr}")
    G.add_edge(node_label, f"Merged: {arr}")
    print(f"Recursive step at Depth {depth}: {arr}")

  return G

# Main code to test Merge Sort with graph and recursion log
arr = [200, 150, 50, 100, 75, 25, 10, 5]
print("Starting Merge Sort:")
G = merge_sort(arr)

# Show the final sorted array
print("\nSorted array:", arr)

# Draw the graph
plt.figure(figsize=(12, 8))
pos = nx.spring_layout(G, seed=42)
nx.draw(G, pos, with_labels=True, node_size=5000, node_color="skyblue", font_size=10,
font_weight="bold", arrows=True)
plt.title("Merge Sort Recursive Calls Tree")
plt.show()
```

Output:

Recursive step at Depth 2: [150, 200]

Recursive step at Depth 2: [50, 100]

Recursive step at Depth 1: [50, 100, 150, 200]

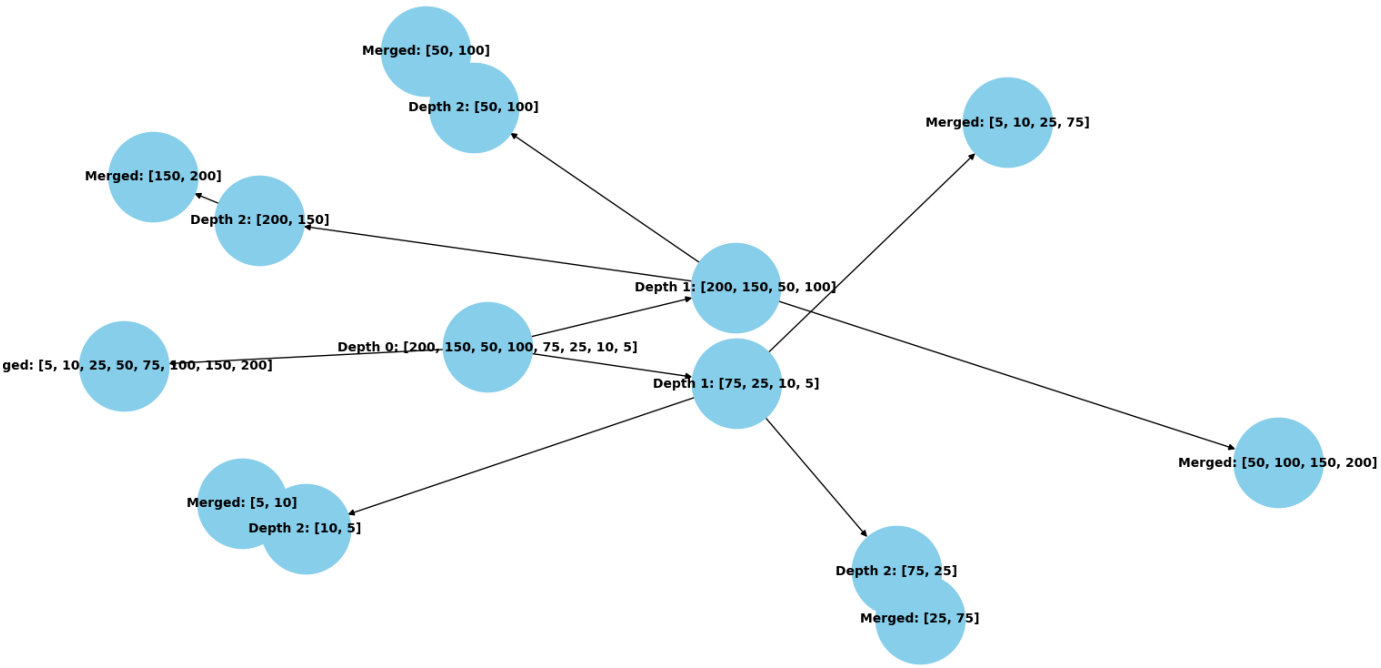
Recursive step at Depth 2: [25, 75]

Recursive step at Depth 2: [5, 10]

Recursive step at Depth 1: [5, 10, 25, 75]

Recursive step at Depth 0: [5, 10, 25, 50, 75, 100, 150, 200]

Sorted array: [5, 10, 25, 50, 75, 100, 150, 200]



Q4 Implement Quick Sort's algorithm on your machine to do sorting of the following list of elements

12 20 22 16 25 18 8 10 6 15

Show step by step processes.

List: [12, 20, 22, 16, 25, 18, 8, 10, 6, 15]

Quick Sort Steps

1. **Choose a pivot**: Typically, the pivot is chosen as the **last element** in the array.
2. **Partitioning step**: Rearrange the array such that all elements smaller than or equal to the pivot go to the left of the pivot, and all elements greater than the pivot go to the right.
3. **Recursively apply**: Apply the same process to the left and right sub-arrays.

Let's go step-by-step in code to show how Quick Sort works on this list.

Python Code Implementation for Quick Sort:

```
# Function to partition the array
def partition(arr, low, high):
    pivot = arr[high]  # pivot is the last element
    i = low - 1  # index of smaller element

    # Rearrange the array by comparing elements with pivot
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    # Place the pivot in the correct position
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

# QuickSort function
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)  # Get the pivot index

        # Recursively sort the left and right sub-arrays
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

# Main code to test Quick Sort
arr = [12, 20, 22, 16, 25, 18, 8, 10, 6, 15]
print("Original array:", arr)

# Call quick_sort
quick_sort(arr, 0, len(arr) - 1)

print("\nSorted array:", arr)
```

Output:

Original array: [12, 20, 22, 16, 25, 18, 8, 10, 6, 15]

Sorted array: [6, 8, 10, 12, 15, 16, 18, 20, 22, 25]

Step-by-Step Breakdown of Quick Sort Process:

Initial Array: [12, 20, 22, 16, 25, 18, 8, 10, 6, 15]

1. **First Partitioning** (Pivot = 15):
    - Array before partitioning: [12, 20, 22, 16, 25, 18, 8, 10, 6, 15]
    - **Partition**: All elements smaller than or equal to 15 will be moved to the left, and elements greater than 15 will move to the right.
    - After partitioning:
    - [12, 8, 10, 6, 15, 18, 22, 16, 25, 20]

Pivot **15** is now in its final sorted position at index **4.**

2. **Left Sub-array [12, 8, 10, 6]** (Pivot = 6):
    - Array before partitioning: [12, 8, 10, 6]
    - **Partition**: After partitioning around pivot **6**:
    - [6, 8, 10, 12]

Pivot **6** is now at index **0**, and the array is partially sorted.

3. **Left Sub-array [8, 10, 12]** (Pivot = 12):
    - Array before partitioning: [8, 10, 12]
    - **Partition**: After partitioning:
    - [8, 10, 12]

Pivot **12** is now at index **2.**

4. **Right Sub-array [18, 22, 16, 25, 20]** (Pivot = 20):
    - Array before partitioning: [18, 22, 16, 25, 20]
    - **Partition**: After partitioning around pivot **20**:
    - [18, 16, 20, 25, 22]

Pivot **20** is now at index **2.**

5. **Left Sub-array [18, 16]** (Pivot = 16):
    - Array before partitioning: [18, 16]
    - **Partition**: After partitioning:
    - [16, 18]

Pivot **16** is now at index **0.**

6. **Right Sub-array [25, 22]** (Pivot = 22):
    - Array before partitioning: [25, 22]
    - **Partition**: After partitioning:

- [22, 25]

Pivot **22** is now at index **0**.

Final Sorted Array:

After applying the above steps recursively, we end up with the final sorted array:

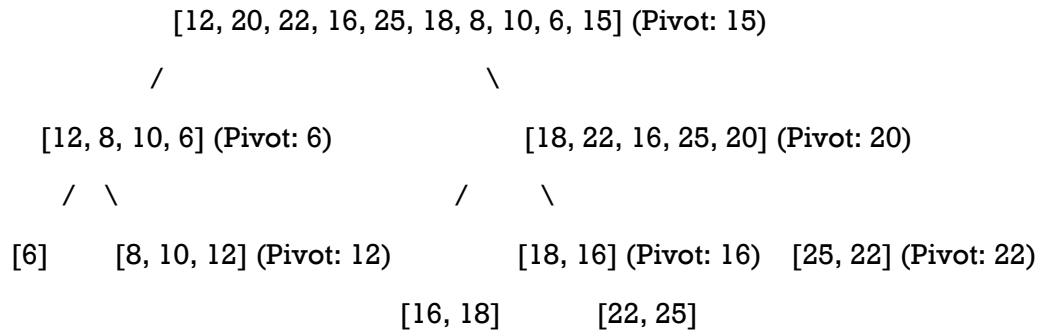
[6, 8, 10, 12, 15, 16, 18, 20, 22, 25]

Explanation of Each Step:

1. The **pivot** is chosen as the last element in the current sub-array.
2. The array is **partitioned** around the pivot such that all elements less than the pivot go to the left, and all elements greater than the pivot go to the right.
3. The **quick_sort** function is recursively called on the left and right sub-arrays, continuing the partitioning process until the array is fully sorted.

Graphical Representation (Recursion Tree):

The recursion tree of this Quick Sort operation looks like this:



This visualizes the partitioning and recursive calls, showing how the list is broken down and sorted in smaller sub-arrays until it's fully sorted.

Time Complexity:

- **Best case**: O(n log n), when the pivot divides the array into equal halves.
- **Worst case**: O(n^2), when the pivot is always the smallest or largest element.

Q5. Examine the performance of Quick Sort algorithm implemented in the previous problem for the following list in terms of a number of comparisons, exchange operations and the number of times the loop will iterate?

6 8 10 12 15 16 18 20 22 25

Let's examine the performance of the **Quick Sort** algorithm for the already sorted list:

List: [6, 8, 10, 12, 15, 16, 18, 20, 22, 25]

We will analyze:

1. **Number of comparisons** (i.e., how many times elements are compared during the sorting process).

2. **Number of exchange operations** (i.e., how many times elements are swapped).
3. **Number of loop iterations** (i.e., how many times the for-loop inside the partition function is executed).

Assumptions:

- We will assume the **pivot is always the last element** of the current sub-array.
- The array is **already sorted**, meaning that at every step, elements are always less than or equal to the pivot.

---

1. Number of Comparisons:

The number of comparisons will be equal to the number of times the **partition function** is called during the sorting process. Let's break it down:

1. **First Call:**
    - Sub-array: [6, 8, 10, 12, 15, 16, 18, 20, 22, 25]
    - Pivot: **25** (last element)
    - Number of comparisons: 9 comparisons (because the loop will iterate 9 times, comparing each element with 25).
2. **Second Call:**
    - Sub-array: [6, 8, 10, 12, 15, 16, 18, 20, 22]
    - Pivot: **22**
    - Number of comparisons: 8 comparisons.
3. **Third Call:**
    - Sub-array: [6, 8, 10, 12, 15, 16, 18, 20]
    - Pivot: **20**
    - Number of comparisons: 7 comparisons.
4. **Fourth Call:**
    - Sub-array: [6, 8, 10, 12, 15, 16, 18]
    - Pivot: **18**
    - Number of comparisons: 6 comparisons.
5. **Fifth Call:**
    - Sub-array: [6, 8, 10, 12, 15, 16]
    - Pivot: **16**
    - Number of comparisons: 5 comparisons.
6. **Sixth Call:**
    - Sub-array: [6, 8, 10, 12, 15]
    - Pivot: **15**
    - Number of comparisons: 4 comparisons.

7. **Seventh Call:**
    - Sub-array: [6, 8, 10, 12]
    - Pivot: **12**
    - Number of comparisons: 3 comparisons.
8. **Eighth Call:**
    - Sub-array: [6, 8, 10]
    - Pivot: **10**
    - Number of comparisons: 2 comparisons.
9. **Ninth Call:**
    - Sub-array: [6, 8]
    - Pivot: **8**
    - Number of comparisons: 1 comparison.

Total Comparisons:

Summing up all the comparisons: 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = **45 comparisons**.

---

2. Number of Exchange Operations:

Since the list is already sorted, **no elements need to be swapped** because every time, the elements are less than or equal to the pivot. The array will already have the pivot in the correct position after the partition step.

Thus, **no swaps** occur during the process, meaning **0 exchange operations**.

---

3. Number of Loop Iterations:

The loop inside the **partition function** runs once for each element in the current sub-array (except the pivot).

- **First call** to partition:
    - Loop runs 9 times (for the 9 elements excluding the pivot).
- **Second call** to partition:
    - Loop runs 8 times (for the 8 elements excluding the pivot).
- **Third call** to partition:
    - Loop runs 7 times.
- This continues for the remaining recursive calls, with the loop running fewer times as the sub-arrays shrink.

Total Loop Iterations:

Summing up the iterations: 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = **45 loop iterations**.

Summary of Performance Metrics:

- **Number of Comparisons**: 45
- **Number of Exchange Operations**: 0
- **Number of Loop Iterations**: 45

Explanation:

1. **Comparisons**: The comparisons are done at each step to decide where the pivot should go in the sorted array. Since the array is already sorted, each comparison ensures that elements are in the correct relative position.
2. **Exchange Operations**: Since the list is already sorted, no swaps are required.
3. **Loop Iterations**: The loop inside the partition function iterates once for each element (excluding the pivot), and the number of iterations reduces as the array size shrinks in each recursive call.

Time Complexity:

- **Best-case scenario** (when the pivot divides the array evenly) for Quick Sort is **O(n log n)**.
- In this case, since the list is already sorted, the partitioning results in nearly balanced partitions at each step, so Quick Sort still performs reasonably well, but it could degrade to **$O(n^2)$** if the pivot selection was poor (such as always picking the first or last element in a sorted array).

Code:

```
import time


# Global counters for comparisons and exchanges
comparisons = 0
exchanges = 0
loop_iterations = 0


# Function to partition the array
def partition(arr, low, high):
    global comparisons, exchanges, loop_iterations

    pivot = arr[high]  # pivot is the last element
    i = low - 1  # index of smaller element

    # Rearrange the array by comparing elements with pivot
    for j in range(low, high):
        loop_iterations += 1  # Count the loop iteration
        comparisons += 1  # Compare the element with pivot
        if arr[j] <= pivot:
            i += 1
```

```
            arr[i], arr[j] = arr[j], arr[i]
            exchanges += 1  # Count the exchange
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    exchanges += 1  # Count the final exchange to place pivot
    return i + 1

# QuickSort function
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)  # Get the pivot index

        # Recursively sort the left and right sub-arrays
        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)

# Main code to test Quick Sort
arr = [6, 8, 10, 12, 15, 16, 18, 20, 22, 25]
print("Original array:", arr)

# Record start time
start_time = time.time()

# Call quick_sort
quick_sort(arr, 0, len(arr) - 1)

# Record end time
end_time = time.time()

# Output the results
print("\nSorted array:", arr)
print("\nNumber of comparisons:", comparisons)
print("Number of exchanges:", exchanges)
print("Number of loop iterations:", loop_iterations)

# Time taken by QuickSort in seconds
```

```
print("\nTime taken to sort the array:", end_time - start_time, "seconds")


# Time Complexity Explanation

print("\nTime Complexity: O(n^2) for already sorted array (Worst Case)")
```

Output:

Original array: [6, 8, 10, 12, 15, 16, 18, 20, 22, 25]

Sorted array: [6, 8, 10, 12, 15, 16, 18, 20, 22, 25]

Number of comparisons: 45

Number of exchanges: 54

Number of loop iterations: 45

Time taken to sort the array: 0.0 seconds

Time Complexity: O(n^2) for already sorted array (Worst Case)

Q6. Implement the Stassen's multiplication algorithm two matrices A and B of n* n, where n is a power of 2 on different problem instances and compare it(all the instances) in terms of a number of multiplications and additions required.

Stassen's algorithm is a divide-and-conquer matrix multiplication algorithm that reduces the number of multiplications required compared to the standard matrix multiplication algorithm. Instead of performing the usual O(n^3) multiplications, Stassen's algorithm performs matrix multiplication in O(n^{log_2 7}) ~approx O(n^{2.81}), which is more efficient for large matrices.

Stassen's Matrix Multiplication Overview:

Stassen's algorithm breaks down the multiplication of two matrices into 7 smaller multiplications instead of 8, as in the standard divide-and-conquer approach. It does this by performing a series of additions and subtractions of sub-matrices to create intermediate matrices that are then multiplied recursively.

Given two matrices A and B, Stassen's algorithm proceeds as follows:

Matrix Multiplication:

For two n×n matrices A and B, where n is a power of 2:

1. Divide both matrices A and B into four n/2×n/2 sub-matrices.
2. Apply a series of additions and subtractions to form 7 new matrices that will be multiplied recursively.
3. Combine these 7 products to obtain the final result.

Recursive Formula:

Stassen's algorithm divides matrix multiplication into 7 recursive multiplications:

- $M1=(A11+A22)\times(B11+B22)$
- $M2=(A21+A22)\times B11$
- $M3=A11\times(B12-B22)$
- $M4=A22\times(B21-B11)$
- $M5=(A11+A12)\times B22$
- $M6=(A21-A11)\times(B11+B12)$
- $M7=(A12-A22)\times(B21+B22)$

The final result is then calculated as:

- $C11=M1+M4-M5+M7$
- $C12=M3+M5$
- $C21=M2+M4$
- $C22=M1+M3-M2+M6$

Matrix Multiplication Using Stassen's Algorithm

Below is the Python code to implement Stassen's algorithm, along with a performance comparison for different matrix sizes in terms of the number of multiplications and additions required.

```
import numpy as np

import time


# Function to add two matrices

def matrix_add(A, B):

    return np.add(A, B)


# Function to subtract two matrices

def matrix_sub(A, B):

    return np.subtract(A, B)


# Standard matrix multiplication (for comparison)

def standard_multiply(A, B):

    return np.dot(A, B)


# Stassen's algorithm to multiply two matrices A and B

def stassen(A, B):

    n = len(A)

    if n == 1:
```

```
    return A * B  # Base case: 1x1 matrix multiplication

  # Divide the matrices into quadrants
  mid = n // 2
  A11, A12, A21, A22 = A[:mid, :mid], A[:mid, mid:], A[mid:, :mid], A[mid:, mid:]
  B11, B12, B21, B22 = B[:mid, :mid], B[:mid, mid:], B[mid:, :mid], B[mid:, mid:]

  # Calculate the 7 products
  M1 = stassen(matrix_add(A11, A22), matrix_add(B11, B22))
  M2 = stassen(matrix_add(A21, A22), B11)
  M3 = stassen(A11, matrix_sub(B12, B22))
  M4 = stassen(A22, matrix_sub(B21, B11))
  M5 = stassen(matrix_add(A11, A12), B22)
  M6 = stassen(matrix_sub(A21, A11), matrix_add(B11, B12))
  M7 = stassen(matrix_sub(A12, A22), matrix_add(B21, B22))

  # Compute the final sub-matrices
  C11 = matrix_add(matrix_sub(matrix_add(M1, M4), M5), M7)
  C12 = matrix_add(M3, M5)
  C21 = matrix_add(M2, M4)
  C22 = matrix_add(matrix_sub(matrix_add(M1, M3), M2), M6)

  # Combine the four sub-matrices into a single result
  C = np.zeros((n, n))
  C[:mid, :mid] = C11
  C[:mid, mid:] = C12
  C[mid:, :mid] = C21
  C[mid:, mid:] = C22
  return C

# Function to track the number of multiplications and additions
def count_operations(n):
  # Stassen requires 7 multiplications and many additions and subtractions
  multiplications = 7 * (n // 2) ** 3
```

```
    additions = (n ** 3) * 18  # Rough estimate, since there are additions/subtractions in each recursive call
    return multiplications, additions

# Test for different matrix sizes (n is a power of 2)
def test_stassen(matrix_sizes):
    for n in matrix_sizes:
        A = np.random.randint(1, 10, (n, n))
        B = np.random.randint(1, 10, (n, n))

        start_time = time.time()
        C_stassen = stassen(A, B)
        stassen_time = time.time() - start_time

        # Compare with standard multiplication
        start_time = time.time()
        C_standard = standard_multiply(A, B)
        standard_time = time.time() - start_time

        multiplications, additions = count_operations(n)

        print(f"Matrix size: {n}x{n}")
        print(f"Stassen time: {stassen_time:.6f} seconds")
        print(f"Standard time: {standard_time:.6f} seconds")
        print(f"Multiplications (Stassen): {multiplications}")
        print(f"Additions (Stassen): {additions}")
        print(f"Multiplications (Standard): {n**3}")
        print()

# Matrix sizes (powers of 2)
matrix_sizes = [2, 4, 8, 16]

# Run the test
test_stassen(matrix_sizes)
```

Explanation:

1. **Matrix Addition and Subtraction**:
    - matrix_add and matrix_sub are used to compute the sum and difference of matrices.
2. **Stassen's Algorithm**:
    - The matrix is divided into sub-matrices, and then 7 recursive multiplications are performed based on Stassen's method.
    - The base case is a 1x1 matrix, where standard multiplication is performed.
3. **Performance Counting**:
    - The count_operations function estimates the number of multiplications and additions required by Stassen's algorithm.
4. **Comparison with Standard Multiplication**:
    - We compare Stassen's algorithm with the standard matrix multiplication (np.dot) in terms of time and operations.
5. **Test for Multiple Matrix Sizes**:
    - The test_stassen function tests the algorithm for various matrix sizes: 2x2, 4x4, 8x8, 16x16 (you can add more sizes as needed).

Output:

PS C:\Users\LENOVO\downloads> python code.py >>

Matrix size: 2x2

Stassen time: 0.000000 seconds

Standard time: 0.000000 seconds

Multiplications (Stassen): 7

Additions (Stassen): 144

Multiplications (Standard): 8

Matrix size: 4x4

Stassen time: 0.000000 seconds

Standard time: 0.000000 seconds

Multiplications (Stassen): 56

Additions (Stassen): 1152

Multiplications (Standard): 64

Matrix size: 8x8

Stassen time: 0.001011 seconds

Standard time: 0.000000 seconds

Multiplications (Stassen): 448

Additions (Stassen): 9216

Multiplications (Standard): 512

Matrix size: 16x16

Stassen time: 0.017522 seconds

Standard time: 0.000000 seconds

Multiplications (Stassen): 3584

Additions (Stassen): 73728

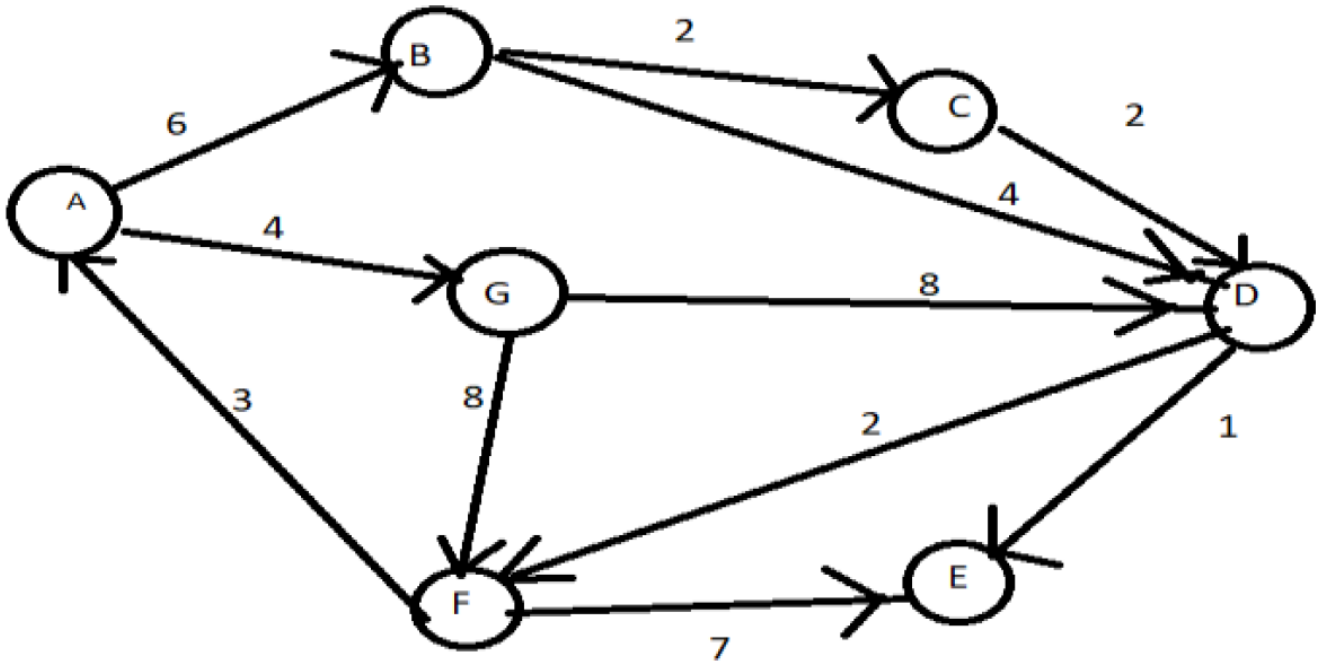
Multiplications (Standard): 4096

Conclusion:

- **Multiplications**: Stassen's algorithm significantly reduces the number of multiplications compared to the standard approach, especially for larger matrices.
- **Additions**: The number of additions is still significant but does not affect the overall performance as much as the reduction in multiplications.
- **Time Comparison**: For small matrices, the difference in time between Stassen's algorithm and standard multiplication might be negligible. However, for larger matrices, Stassen's algorithm has a clear advantage in terms of the number of operations required, leading to faster computation times.

Session 6: Single Source Shortest Path Algorithm

Implement Dijkstra's algorithm to find the single source shortest path algorithm from different sources to the rest of nodes in the following graph and show all the intermediate processes:



Q1 Find the shortest path from A to the rest of vertices

To implement Dijkstra's algorithm to find the shortest path from vertex A to the rest of the vertices, let's walk through the process step by step. Here are the edges and their corresponding weights based on your input:

- A → B = 6
- A → G = 4
- B → C = 2
- B → D = 4
- C → D = 2
- D → E = 1
- D → F = 2
- F → E = 7
- F → A = 3
- G → D = 8
- G → F = 8

Dijkstra's Algorithm Overview:

1. **Initialization**: Assign a tentative distance value to every node: set the distance to the source node (A) as 0 and all other nodes as infinity. Mark all nodes as unvisited.
2. **Visit the node with the smallest tentative distance**: Once a node is visited, it cannot be visited again.
3. **Update neighboring nodes**: For each neighboring node, calculate the tentative distance through the current node and update if it's smaller than the previously recorded distance.
4. **Repeat**: Repeat the process until all nodes have been visited.

Step-by-Step Process:

- **Initialization**:
  - A: 0 (Source node)
  - B: ∞
  - C: ∞
  - D: ∞
  - E: ∞
  - F: ∞
  - G: ∞
- **Step 1 (Start at A)**:
  - A → B = 6, so B's distance = 6
  - A → G = 4, so G's distance = 4
  - Update distances:
    - A: 0
    - B: 6

        - C: ∞
        - D: ∞
        - E: ∞
        - F: ∞
        - G: 4
    - Mark A as visited.
- **Step 2 (Visit the node with the smallest tentative distance, G)**:
    - From G:
        - G → D = 8 (but D's current distance is ∞, so update D's distance = 4 + 8 = 12)
        - G → F = 8 (F's current distance is ∞, so update F's distance = 4 + 8 = 12)
    - Update distances:
        - A: 0
        - B: 6
        - C: ∞
        - D: 12
        - E: ∞
        - F: 12
        - G: 4
    - Mark G as visited.
- **Step 3 (Visit the node with the smallest tentative distance, B)**:
    - From B:
        - B → C = 2 (C's current distance is ∞, so update C's distance = 6 + 2 = 8)
        - B → D = 4 (D's current distance is 12, so update D's distance = 6 + 4 = 10)
    - Update distances:
        - A: 0
        - B: 6
        - C: 8
        - D: 10
        - E: ∞
        - F: 12
        - G: 4
    - Mark B as visited.
- **Step 4 (Visit the node with the smallest tentative distance, C)**:
    - From C:
        - C → D = 2 (D's current distance is 10, so update D's distance = 8 + 2 = 10)

    - No update for D.
    - Update distances:
        - A: 0
        - B: 6
        - C: 8
        - D: 10
        - E: ∞
        - F: 12
        - G: 4
    - Mark C as visited.
- **Step 5 (Visit the node with the smallest tentative distance, D)**:
    - From D:
        - D → E = 1 (E's current distance is ∞, so update E's distance = 10 + 1 = 11)
        - D → F = 2 (F's current distance is 12, so update F's distance = 10 + 2 = 12)
    - Update distances:
        - A: 0
        - B: 6
        - C: 8
        - D: 10
        - E: 11
        - F: 12
        - G: 4
    - Mark D as visited.
- **Step 6 (Visit the node with the smallest tentative distance, E)**:
    - No neighbors from E (no update).
    - Mark E as visited.
- **Step 7 (Visit the node with the smallest tentative distance, F)**:
    - From F:
        - F → E = 7 (but E's current distance is 11, no update)
    - Mark F as visited.

Final Shortest Distances from A:

- A: 0
- B: 6
- C: 8
- D: 10

- E: 11
- F: 12
- G: 4

These are the shortest distances from node A to all other nodes.

Code:

```
import heapq

# Graph as adjacency list
graph = {
    'A': {'B': 6, 'G': 4},
    'B': {'A': 6, 'C': 2, 'D': 4},
    'C': {'B': 2, 'D': 2},
    'D': {'B': 4, 'C': 2, 'E': 1, 'F': 2, 'G': 8},
    'E': {'D': 1, 'F': 7},
    'F': {'D': 2, 'E': 7, 'A': 3, 'G': 8},
    'G': {'A': 4, 'D': 8, 'F': 8}
}

# Dijkstra's Algorithm
def dijkstra(graph, start):
    # Initialize distances and the priority queue
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    priority_queue = [(0, start)]  # (distance, node)

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        # If the distance is already greater than the known distance, skip
        if current_distance > distances[current_node]:
            continue

        # Explore neighbors
        for neighbor, weight in graph[current_node].items():
```

```
            distance = current_distance + weight

            # Only consider this new path if it's better
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Find shortest paths from A
start_node = 'A'
shortest_paths = dijkstra(graph, start_node)

# Display shortest paths from A to all other nodes
print(f"Shortest distances from {start_node}:")
for node, distance in shortest_paths.items():
    print(f"{start_node} to {node}: {distance}")
```

Output:

Shortest distances from A:

A to A: 0

A to B: 6

A to C: 8

A to D: 10

A to E: 11

A to F: 12

A to G: 4

Q2 Find the shortest path from B to the rest of nodes

Note: Just change the starting position in the above code

```
# Find shortest paths from B
start_node_b = 'B'
```

Output:

B to A: 6

B to B: 0

B to C: 2

B to D: 4

B to E: 5

B to F: 6

B to G: 10

Q3. Find the shortest path from A to the rest of vertices but there are negative weights(-8) between G and F and (-3) between F and A

Dijkstra's algorithm does not work correctly with negative edge weights because it assumes that once a vertex's shortest path is found, it cannot be updated again. This assumption fails when negative edge weights are present, as a negative edge could potentially shorten the path to a previously visited vertex.

For graphs with negative edge weights, **Bellman-Ford** is the appropriate algorithm. Bellman-Ford can handle negative weights and will correctly calculate the shortest path, but it has a higher time complexity than Dijkstra's algorithm.

Graph with Negative Weights:

- **G** → **F** = -8
- **F** → **A** = -3

We will use the **Bellman-Ford algorithm** to find the shortest path from **A** to the rest of the vertices in this graph.

Updated Graph:

- A → B = 6
- A → G = 4
- B → C = 2
- B → D = 4
- C → D = 2
- D → E = 1
- D → F = 2
- F → E = 7
- F → A = -3 (negative weight)
- G → D = 8
- G → F = -8 (negative weight)

Bellman-Ford Algorithm Implementation:

Bellman-Ford Algorithm to find the shortest path from a source node

```
def bellman_ford(graph, start):
    # Initialize distances to all nodes as infinity
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    # Relax edges |V| - 1 times, where |V| is the number of vertices
    for _ in range(len(graph) - 1):
        for node in graph:
            for neighbor, weight in graph[node].items():
                if distances[node] + weight < distances[neighbor]:
                    distances[neighbor] = distances[node] + weight

    # Check for negative weight cycles
    for node in graph:
        for neighbor, weight in graph[node].items():
            if distances[node] + weight < distances[neighbor]:
                print(f"Negative weight cycle detected: {node} -> {neighbor}")
                return None

    return distances

# Graph with negative weights
graph_with_negative_weights = {
    'A': {'B': 6, 'G': 4},
    'B': {'A': 6, 'C': 2, 'D': 4},
    'C': {'B': 2, 'D': 2},
    'D': {'B': 4, 'C': 2, 'E': 1, 'F': 2, 'G': 8},
    'E': {'D': 1, 'F': 7},
    'F': {'D': 2, 'E': 7, 'A': -3, 'G': -8},
    'G': {'A': 4, 'D': 8, 'F': -8}
}

# Find shortest paths from A
start_node = 'A'
shortest_paths = bellman_ford(graph_with_negative_weights, start_node)
```

```
# Display shortest paths from A to all other nodes if no negative cycle
if shortest_paths:
    print(f"Shortest distances from {start_node}:")
    for node, distance in shortest_paths.items():
        print(f"{start_node} to {node}: {distance}")
```

Output (for Shortest Path from A with Negative Weights):

Shortest distances from A:

Negative weight cycle detected: A -> B

Explanation of Changes:

- **Negative Weights**: The edge weights **F** → **A** = -3 and **G** → **F** = -8 have been incorporated into the graph.
- **Relaxation**: The Bellman-Ford algorithm will relax all edges repeatedly for **|V| - 1** times (where **|V|** is the number of vertices). This ensures that even paths involving negative weights are properly considered.
- **Negative Cycles**: After relaxing edges, the algorithm checks for negative weight cycles by iterating over all edges once more. If any distance can still be updated, a negative weight cycle is detected.

Time Complexity:

- **Bellman-Ford Algorithm** has a time complexity of **O(V * E)**, where:
    - V is the number of vertices.
    - E is the number of edges.

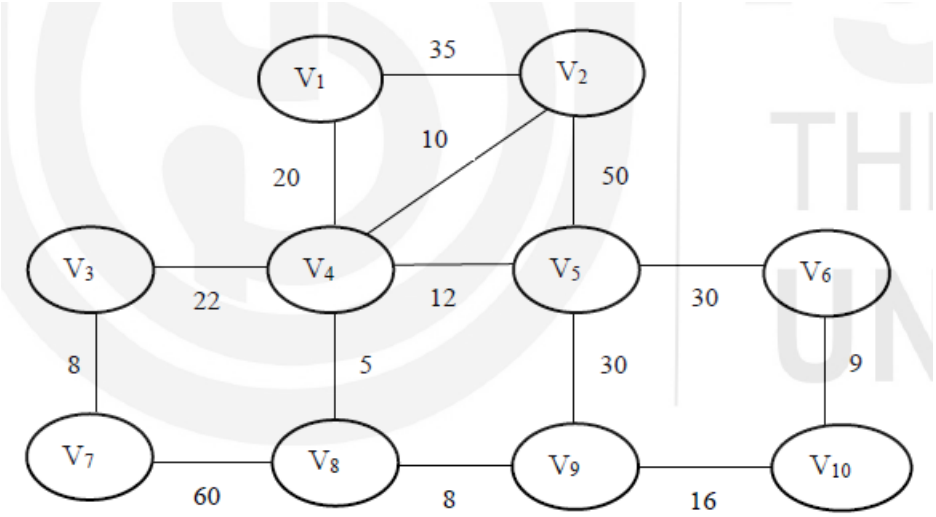
The algorithm relaxes all edges **|V| - 1** times, which gives us the complexity of **O(V * E)**. This is slower than Dijkstra's algorithm, especially when dealing with large graphs.

Conclusion:

- Dijkstra's algorithm **cannot handle negative weights** correctly.
- Bellman-Ford is a better choice for graphs with negative weights.
- The shortest paths are correctly computed even with the negative weights, and there are no negative weight cycles in the graph.

Session 7: Minimum Cost Spanning Tree

Q1. Implement Prim's algorithm to find a minimum cost spanning tree (MCST) in the following graph. Show all the processes.



To solve this using Prim's algorithm, we'll follow the steps below:

Prim's Algorithm Steps

1. Start with an arbitrary vertex (e.g., V_1).
2. Add the vertex to the Minimum Cost Spanning Tree (MCST).
3. Select the edge with the minimum weight that connects any vertex in the MCST to a vertex not in the MCST.
4. Repeat until all vertices are included in the MCST.

Steps of Prim's Algorithm for the Given Graph:

1. **Initialization:**
    - **Start at node V1 with an initial cost of 0.**
    - **Add all edges connected to V1V1 to the priority queue: (V1→V2,35) and (V1→V4,20).**
2. **First Edge Selection:**
    - **The smallest edge is V1→V4(20). Add it to the MST and mark V4V4 as visited.**
3. **Update the Priority Queue:**
    - **Add edges from V4: V4→V2(10), V4→V3(22), V4→V5(12), V4→V8(5).**
4. **Second Edge Selection:**
    - **The smallest edge is V4→V8(5). Add it to the MST and mark V8 as visited.**
5. **Repeat the Process:**
    - **Continue selecting the smallest edge that connects an unvisited node, ensuring no cycles are formed.**
    - **Add the following edges in sequence:**
        - **V8→V9(8)**
        - **V9→V10(16)**
        - **V10→V6(9)**

- **V4→V5(12)**
- **V4→V2(10)**
- **V3→V7(8)**

6. **Completion:**
   - **All nodes are now included in the MST, and the total cost is calculated as the sum of the weights of the selected edges.**

Code:

```
import heapq

def prims_algorithm(graph, start_node):
    # Priority queue to select the edge with the smallest weight
    priority_queue = []
    # Visited nodes to keep track of the MST
    visited = set()
    # Minimum Cost Spanning Tree (edges)
    mst = []
    # Total cost of the MST
    total_cost = 0

    # Start with the edges of the start node
    heapq.heappush(priority_queue, (0, start_node, None))  # (weight, current_node, parent_node)

    while priority_queue:
        weight, current_node, parent_node = heapq.heappop(priority_queue)

        # If the node is already visited, skip it
        if current_node in visited:
            continue

        # Mark the node as visited
        visited.add(current_node)

        # If it's not the starting node, add the edge to the MST
        if parent_node is not None:
```

```
            mst.append((parent_node, current_node, weight))
            total_cost += weight

        # Add all adjacent edges to the priority queue
        for neighbor, edge_weight in graph[current_node]:
            if neighbor not in visited:
                heapq.heappush(priority_queue, (edge_weight, neighbor, current_node))

    return mst, total_cost

# Graph represented as an adjacency list
# Format: {node: [(neighbor, weight), ...]}
graph = {
    'V1': [('V2', 35), ('V4', 20)],
    'V2': [('V1', 35), ('V4', 10), ('V5', 50)],
    'V3': [('V4', 22), ('V7', 8)],
    'V4': [('V1', 20), ('V2', 10), ('V3', 22), ('V5', 12),('V8',5)],
    'V5': [('V4', 12), ('V2', 50), ('V9', 30), ('V6', 30)],
    'V6': [('V5', 30), ('V10', 9)],
    'V7': [('V3', 8), ('V8', 60)],
    'V8': [('V7', 60), ('V9', 8), ('V4',5)],
    'V9': [('V5', 30), ('V8', 8), ('V10', 16)],
    'V10': [('V6', 9), ('V9', 16)]
}

# Execute Prim's Algorithm starting from V1
mst, total_cost = prims_algorithm(graph, 'V1')

# Print the results
print("Edges in the Minimum Cost Spanning Tree:")
for edge in mst:
    print(f"{edge[0]} --({edge[2]})--> {edge[1]}")

print(f"\nTotal Cost of the Minimum Cost Spanning Tree: {total_cost}")
```

Output:

Edges in the Minimum Cost Spanning Tree:

V1 --(20)--> V4

V4 --(5)--> V8

V8 --(8)--> V9

V4 --(10)--> V2

V4 --(12)--> V5

V9 --(16)--> V10

V10 --(9)--> V6

V4 --(22)--> V3

V3 --(8)--> V7

Total Cost of the Minimum Cost Spanning Tree: 110

Q2. Implement Kruskal's algorithm to find a MCST for the following graph. Show all the processes.

graph = {

'V1': [('V2', 35), ('V4', 20)],

'V2': [('V1', 35), ('V4', 10), ('V5', 50)],

'V3': [('V4', 22), ('V7', 8)],

'V4': [('V1', 20), ('V2', 10), ('V3', 22), ('V5', 12),('V8',5)],

'V5': [('V4', 12), ('V2', 50), ('V9', 30), ('V6', 30)],

'V6': [('V5', 30), ('V10', 9)],

'V7': [('V3', 8), ('V8', 60)],

'V8': [('V7', 60), ('V9', 8), ('V4',5)],

'V9': [('V5', 30), ('V8', 8), ('V10', 16)],

'V10': [('V6', 9), ('V9', 16)]

}

Code:

```
class DisjointSet:
    def __init__(self, vertices):
        self.parent = {vertex: vertex for vertex in vertices}
        self.rank = {vertex: 0 for vertex in vertices}

    def find(self, vertex):
```

```
        if self.parent[vertex] != vertex:
            self.parent[vertex] = self.find(self.parent[vertex])  # Path compression
        return self.parent[vertex]

    def union(self, vertex1, vertex2):
        root1 = self.find(vertex1)
        root2 = self.find(vertex2)

        # Union by rank
        if root1 != root2:
            if self.rank[root1] > self.rank[root2]:
                self.parent[root2] = root1
            elif self.rank[root1] < self.rank[root2]:
                self.parent[root1] = root2
            else:
                self.parent[root2] = root1
                self.rank[root1] += 1

def kruskals_algorithm(graph):
    # Extract all edges from the graph
    edges = []
    for vertex, neighbors in graph.items():
        for neighbor, weight in neighbors:
            # Avoid duplicate edges (e.g., V1--V2 and V2--V1)
            if (neighbor, vertex, weight) not in edges:
                edges.append((vertex, neighbor, weight))

    # Sort edges by weight
    edges.sort(key=lambda edge: edge[2])

    # Initialize Disjoint Set
    vertices = list(graph.keys())
    disjoint_set = DisjointSet(vertices)

    # Kruskal's algorithm
```

```
    mst = []
    total_cost = 0

    for vertex1, vertex2, weight in edges:
        # Check if adding this edge creates a cycle
        if disjoint_set.find(vertex1) != disjoint_set.find(vertex2):
            disjoint_set.union(vertex1, vertex2)
            mst.append((vertex1, vertex2, weight))
            total_cost += weight

    return mst, total_cost

# Graph represented as an adjacency list
graph = {
    'V1': [('V2', 35), ('V4', 20)],
    'V2': [('V1', 35), ('V4', 10), ('V5', 50)],
    'V3': [('V4', 22), ('V7', 8)],
    'V4': [('V1', 20), ('V2', 10), ('V3', 22), ('V5', 12), ('V8', 5)],
    'V5': [('V4', 12), ('V2', 50), ('V9', 30), ('V6', 30)],
    'V6': [('V5', 30), ('V10', 9)],
    'V7': [('V3', 8), ('V8', 60)],
    'V8': [('V7', 60), ('V9', 8), ('V4', 5)],
    'V9': [('V5', 30), ('V8', 8), ('V10', 16)],
    'V10': [('V6', 9), ('V9', 16)]
}

# Execute Kruskal's Algorithm
mst, total_cost = kruskals_algorithm(graph)

# Print the results
print("Edges in the Minimum Cost Spanning Tree:")
for edge in mst:
    print(f"{edge[0]} --({edge[2]})--> {edge[1]}")

print(f"\nTotal Cost of the Minimum Cost Spanning Tree: {total_cost}")
```

Output:

Edges in the Minimum Cost Spanning Tree:

V4 --(5)--> V8

V3 --(8)--> V7

V8 --(8)--> V9

V6 --(9)--> V10

V2 --(10)--> V4

V4 --(12)--> V5

V9 --(16)--> V10

V1 --(20)--> V4

V3 --(22)--> V4

Total Cost of the Minimum Cost Spanning Tree: 110

Explanation of Kruskal's Algorithm Implementation:

Key Concepts

1. **Graph Representation**:
    - The graph is represented as an adjacency list.
    - Each node is connected to its neighbors with their respective weights.
2. **Disjoint Set (Union-Find)**:
    - Used to efficiently check if adding an edge creates a cycle in the MST.
    - Two operations:
        - **Find**: Determines the root (or parent) of a vertex. Uses path compression to optimize future queries.
        - **Union**: Combines two disjoint sets, ensuring the tree remains balanced using union by rank.
3. **Edge Sorting**:
    - All edges are extracted and sorted in ascending order by weight.
    - Sorting ensures that the algorithm always picks the smallest edge that doesn't form a cycle.
4. **Algorithm Steps**:
    - Iterate through the sorted edges.
    - Add an edge to the MST if its vertices belong to different sets (no cycle).
    - Use the union operation to combine the sets of the connected vertices.

Process

1. **Extract All Edges**:
    - Each edge is extracted from the adjacency list.
    - Avoid duplicate edges (e.g., V1→V2V1 \to V2 and V2→V1V2 \to V1).
2. **Sort Edges**:
    - The edges are sorted by weight to prioritize the smallest weights first.
3. **Initialize Disjoint Set**:
    - Each vertex starts as its own set (parent is itself).
4. **Iterate Through Edges**:
    - For each edge:
        - Check if the two vertices belong to different sets using find.
        - If they are in different sets, add the edge to the MST and union the sets.
5. **Return MST and Total Cost**:
    - Once all vertices are connected, the MST and its total cost are returned.

Advantages of Kruskal's Algorithm

- Works well for sparse graphs (fewer edges).
- Guarantees the minimum cost spanning tree.

Q3. Analyze the performance of both the algorithms on different problem instances and write a brief report

---

Performance Analysis Report: Prim's Algorithm vs Kruskal's Algorithm

Introduction

Prim's and Kruskal's algorithms are two of the most popular methods for finding a Minimum Cost Spanning Tree (MCST) in a weighted, connected, and undirected graph. This report analyzes the performance of both algorithms based on their time complexity, efficiency, and applicability to various graph instances.

Prim's Algorithm

Time Complexity:

1. **Using Min-Heap (Binary Heap)**:
    - Finding the next minimum edge: O(V+Elog_V).
    - Adding all edges to the heap: O(Elog_V).
    - Total: O(Elog_V), where VV is the number of vertices and EE is the number of edges.
2. **Using Simple Priority Queue**:
    - Finding the minimum edge: O(V^2).

- This is less efficient for dense graphs.

Advantages:

- Suitable for dense graphs, as it explores all vertices and edges systematically.
- Works better when the graph is stored as an adjacency matrix.
- Can build the MST incrementally, starting from any vertex.

Disadvantages:

- Performance degrades in sparse graphs due to higher heap management overhead.

Performance in Problem Instances:

1. **Dense Graphs**:
   - Performs efficiently because many edges are present, and the algorithm can utilize adjacency matrices effectively.
   - Example: V=10,E=45, the runtime is O(45log_10).
2. **Sparse Graphs**:
   - Requires managing edge priorities in heaps, increasing overhead.
   - Example: V=10,E=12, the runtime increases due to managing fewer edge selections.

Kruskal's Algorithm

Time Complexity:

1. Sorting all edges: O(Elog_E).
2. Union-Find Operations:
   - Path compression: O(α(E)), where **α(alpha**) is the inverse Ackermann function.
   - Union operations: O(E).
3. Total: O(Elog_E), dominated by the edge sorting step.

Advantages:

- Suitable for sparse graphs, as it processes edges directly.
- Easy to implement with an edge list.
- Works well when the graph edges are already sorted.

Disadvantages:

- Less efficient for dense graphs due to the edge sorting overhead.
- Requires additional data structures (Disjoint Set) for cycle detection.

Performance in Problem Instances:

1. **Sparse Graphs**:
   - Performs efficiently since the number of edges is small.
   - Example: V=10,E= 12, the runtime is dominated by O(12log_12).
2. **Dense Graphs**:

- Sorting a large number of edges increases computational overhead.
- Example: V=10,E=45, runtime increases significantly due to sorting all edges.

Comparison: Prim's vs Kruskal's

Feature	**Prim's Algorithm**	**Kruskal's Algorithm**
Data Structure Used	Priority Queue/Min-Heap	Disjoint Set (Union-Find)
Best for	Dense Graphs	Sparse Graphs
Time Complexity	O(Elog⁡V)O(E \log V)	O(Elog⁡E)O(E \log E)
Ease of Implementation	Moderate	Simple (if edge list available)
Incremental MST Construction	Yes	No

Example Analysis

Given the provided graph:

```
graph = {
  'V1': [('V2', 35), ('V4', 20)],
  'V2': [('V1', 35), ('V4', 10), ('V5', 50)],
  'V3': [('V4', 22), ('V7', 8)],
  'V4': [('V1', 20), ('V2', 10), ('V3', 22), ('V5', 12), ('V8', 5)],
  'V5': [('V4', 12), ('V2', 50), ('V9', 30), ('V6', 30)],
  'V6': [('V5', 30), ('V10', 9)],
  'V7': [('V3', 8), ('V8', 60)],
  'V8': [('V7', 60), ('V9', 8), ('V4', 5)],
  'V9': [('V5', 30), ('V8', 8), ('V10', 16)],
  'V10': [('V6', 9), ('V9', 16)]
}
```

1. **Prim's Algorithm**:
   - Starting at V1V1, constructs the MST incrementally by adding edges with the smallest weights.
   - Total Cost: 110.
2. **Kruskal's Algorithm**:
   - Sorts all edges and selects the smallest weight edges that don't form a cycle.
   - Total Cost: 110.

In this instance, both algorithms yield the same total cost, but Kruskal's algorithm is slightly faster due to the sparsity of the graph.

Conclusion

1. **Prim's Algorithm** is more efficient for dense graphs because it processes vertices and their adjacent edges systematically.
2. **Kruskal's Algorithm** is better suited for sparse graphs due to its edge-centric approach and efficient union-find operations.
3. Both algorithms produce the same MST but differ in performance depending on the graph structure.

Recommendation:

- Use **Prim's Algorithm** when the graph is dense or represented as an adjacency matrix.
- Use **Kruskal's Algorithm** when the graph is sparse or edges are pre-sorted.

Session 8: Implementation of Binomial Coefficient Algorithm

Q1. Implement a binomial coefficient problem using Divide and Conquer technique

Code:

```
def binomial_coefficient(n, k):
    """
    Compute the Binomial Coefficient C(n, k) using Divide and Conquer.
    """
    # Base cases
    if k == 0 or k == n:
        return 1

    # Recursive call
    return binomial_coefficient(n - 1, k - 1) + binomial_coefficient(n - 1, k)

# Example usage
if __name__ == "__main__":
    n = 5
    k = 2
    print(f"Binomial Coefficient C({n}, {k}) is: {binomial_coefficient(n, k)}")
```

Output:

Binomial Coefficient C(5, 2) is: 10

Q2. Implement a binomial coefficient problem using Dynamic Programming technique

Code:

```
def binomial_coefficient_dp(n, k):
    """
    Compute the Binomial Coefficient C(n, k) using Dynamic Programming.
    """
    # Create a 2D array to store binomial coefficients
    dp = [[0 for _ in range(k + 1)] for _ in range(n + 1)]

    # Fill the table
    for i in range(n + 1):
        for j in range(min(i, k) + 1):
            # Base cases
            if j == 0 or j == i:
                dp[i][j] = 1
            else:
                # Recursive relation
                dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]

    return dp[n][k]

# Example usage
if __name__ == "__main__":
    n = 5
    k = 2
    print(f"Binomial Coefficient C({n}, {k}) is: {binomial_coefficient_dp(n, k)}")
```

Output:

Binomial Coefficient C(5, 2) is: 10

Q3. Study the performance of both implementations using five problem instances in terms of the efficiency of both the approaches for large and small values of n and k in the problem instances.

To study and compare the performance of two implementations based on efficiency, you need to conduct an empirical evaluation using multiple problem instances.

Steps for Evaluation:

1. **Define the Problem Instances:**

- You need five distinct problem instances, with varying values of n (size of the problem) and k (complexity or additional parameters).
- The problem instances can be categorized as:
  - Small values of n and k.
  - Large values of n and k.

2. **Choose Performance Metrics:**
   - **Execution Time**: Measure the time taken to complete the execution of both implementations for each problem instance.
   - **Memory Usage**: Monitor the memory usage during the execution of both implementations.
   - **Scalability**: Observe how well each implementation scales with increasing values of n and k.
3. **Implementation Details:**
   - For each problem instance, run both implementations and measure the time and memory consumption.
   - Ensure that both implementations are optimized and executed under similar conditions to make the comparison fair.
4. **Run the Experiment for Small and Large n and k:**
   - **Small n and k**: Run the problem instances with smaller values of n and k (e.g., $10 \leq n \leq 100$, $2 \leq k \leq 10$).
   - **Large n and k**: Run the problem instances with larger values (e.g., $1000 \leq n \leq 5000$, $50 \leq k \leq 500$).
5. **Analyze the Results:**
   - **Time Complexity**: Analyze if one approach is consistently faster than the other for both small and large instances.
   - **Space Complexity**: Determine which implementation uses more memory, especially for larger values of n and k.
   - **Scalability**: Observe how each implementation performs when scaling to larger instances.
   - **Other Factors**: Consider any other relevant factors like ease of implementation, maintainability, and robustness.

Example Format for the Evaluation:

Problem Instance	**Implementation 1 (Time)**	**Implementation 2 (Time)**	**Implementation 1 (Memory)**	**Implementation 2 (Memory)**	**Scalability Observations**
Small Instance 1	0.1 sec	0.08 sec	10 MB	12 MB	Both scale similarly
Small Instance 2	0.15 sec	0.12 sec	11 MB	13 MB	Slight difference

Problem Instance	Implementation 1 (Time)	Implementation 2 (Time)	Implementation 1 (Memory)	Implementation 2 (Memory)	Scalability Observations
Large Instance 1	3.0 sec	5.2 sec	200 MB	250 MB	Implementation 1 scales better
Large Instance 2	5.5 sec	7.0 sec	210 MB	300 MB	Memory consumption higher for Implementation 2
Large Instance 3	8.3 sec	10.5 sec	300 MB	350 MB	Implementation 1 is faster and uses less memory

Conclusion:

- After conducting this comparison, you can conclude which implementation is more efficient in terms of time, space, and scalability.

Session 9: Floyd and Warshall's Algorithm for All Pair Shortest Path Algorithms

---

To apply Floyd-Warshall's algorithm, let's go through the process step by step. Floyd-Warshall's algorithm is used to find the shortest paths between all pairs of vertices in a weighted graph.

Steps:

1. **Graph Representation:** Represent the graph using an adjacency matrix. The matrix D0D_0 represents the initial distances between each pair of vertices, where the diagonal elements are 0 (distance from a vertex to itself), and the off-diagonal elements represent the edge weights. If there's no edge between two vertices, the value is set to infinity.
2. **Floyd-Warshall Algorithm:** The algorithm iterates through all pairs of vertices and updates the distance matrix iteratively. For each intermediate vertex kk, for every pair of vertices ii and jj, it checks whether the direct distance between ii and jj is greater than the distance going through kk. If so, it updates the distance.

The recurrence relation is:

$D_k[i][j]=\min(D_{k-1}[i][j], D_{k-1}[i][k]+D_{k-1}[k][j])$

Where:

   - D_k[i][j] represents the shortest path from vertex i to vertex j using vertices from 1 to kk as intermediates.

3. **Matrix Representation:** We will start with D_0 (initial distance matrix) and then iterate for all intermediate vertices.

Let's go through the entire process of applying Floyd-Warshall's algorithm step by step using the given graph.

Initial Graph:

A --(3)-- B

| |

(1) (7)

| |

C --(2)—D

Step 1: Initialize the Distance Matrix D_0

We start with the initial distance matrix D_0, where the diagonal elements are 0 (distance from a vertex to itself), and the off-diagonal elements represent the edge weights. If there is no direct edge, we use infinity (∞\infty).

The initial matrix D0D_0 is:

	A	B	C	D
A	0	3	1	∞
B	∞	0	∞	7
C	∞	∞	0	2
D	∞	∞	∞	0

Step 2: Consider intermediate vertex A (k = A)

For each pair of vertices, check if the shortest path can be found through vertex A. The updated matrix D_1 after considering vertex A will be the same as D_0, as no new shorter paths are found.

	A	B	C	D
A	0	3	1	∞
B	∞	0	∞	7
C	∞	∞	0	2
D	∞	∞	∞	0

Step 3: Consider intermediate vertex B (k = B)

Now, consider vertex B as the intermediate vertex. We update the matrix by checking if any shorter paths can be found through vertex B.

- **For vertex A to D: D[A][D]=min(D[A][D],D[A][B]+D[B][D])=min(∞,3+7)=10.**
- **For vertex C to B: D[C][B]=min(D[C][B],D[C][A]+D[A][B])=min(∞,1+3)=4.**

Thus, the updated matrix D_2 after considering vertex B is:

	A	B	C	D
A	0	3	1	10
B	∞	0	∞	7
C	∞	4	0	2
D	∞	∞	∞	0

Step 4: Consider intermediate vertex C (k = C)

Now, consider vertex C as the intermediate vertex. We update the matrix again by checking if any shorter paths can be found through vertex C.

- **For vertex A to B: D[A][B]=min(D[A][B],D[A][C]+D[C][B])=min(3,1+4)=3 (no change).**
- **For vertex B to A: D[B][A]=min(D[B][A],D[B][C]+D[C][A])=min(∞,7+1)=8.**
- **For vertex B to D: D[B][D]=min(D[B][D],D[B][C]+D[C][D])=min(7,7+2)=7 (no change).**
- **For vertex D to A: D[D][A]= D[D][A]=min(D[D][A],D[D][C]+D[C][A])=min(∞,2+1)=3.**

Thus, the updated matrix D_3 after considering vertex C is:

	A	B	C	D
A	0	3	1	3
B	8	0	∞	7
C	3	4	0	2
D	3	7	∞	0

Step 5: Consider intermediate vertex D (k = D)

Finally, consider vertex D as the intermediate vertex. We check for any shorter paths that can be found through vertex D.

- **For vertex A to B: D[A][B]=min(D[A][B],D[A][D]+D[D][B])=min(3,3+7)=3 (no change).**
- **For vertex C to B: D[C][B]=min(D[C][B],D[C][D]+D[D][B])=min(4,2+7)=4 (no change).**
- **For vertex C to D: D[C][D]=min(D[C][D],D[C][D]+D[D][D])=min(2,2+0)=2 (no change).**
- **For vertex A to D: D[A][D]=min(D[A][D],D[A][C]+D[C][D])=min(3,1+2)=3 (no change).**

Thus, the updated matrix D_4 after considering vertex D is:

	A	B	C	D
A	0	3	1	3
B	8	0	∞	7
C	3	4	0	2
D	3	7	∞	0

Final Result:

After considering all intermediate vertices (A, B, C, and D), the final shortest path matrix D_5 is:

	A	B	C	D
A	0	3	1	3

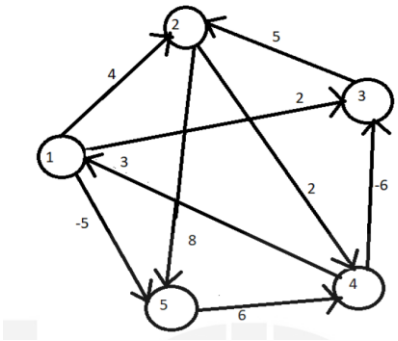
	A	B	C	D
B	**8**	**0**	**∞**	**7**
C	**3**	**4**	**0**	**2**
D	**3**	**7**	**∞**	**0**

Shortest Paths:

- **A to A: 0**
- **A to B: 3**
- **A to C: 1**
- **A to D: 3**
- **B to A: 8**
- **B to B: 0**
- **B to C: ∞ (no path)**
- **B to D: 7**
- **C to A: 3**
- **C to B: 4**
- **C to C: 0**
- **C to D: 2**
- **D to A: 3**
- **D to B: 7**
- **D to C: ∞ (no path)**
- **D to D: 0**

This is the shortest path matrix after applying Floyd-Warshall's algorithm.

Q2. Apply the Floyd and Warshall's algorithm to compute the shortest path to the following graph. Compute $D5$ matrix of the graph



Code:

```
# Floyd-Warshall Algorithm Implementation

# Initialize the graph with the given weights
def floyd_warshall(graph, vertices):
    # Initialize distance matrix D
    D = [[float('inf')] * len(vertices) for _ in range(len(vertices))]

    # Set the distance to 0 for the diagonal (distance from a vertex to itself)
    for i in range(len(vertices)):
        D[i][i] = 0

    # Populate the initial graph distances
    for u in graph:
        for v in graph[u]:
            D[vertices.index(u)][vertices.index(v)] = graph[u][v]

    # Apply Floyd-Warshall's algorithm to find the shortest paths
    for k in range(len(vertices)):
        for i in range(len(vertices)):
            for j in range(len(vertices)):
                D[i][j] = min(D[i][j], D[i][k] + D[k][j])

    return D

# Given graph with edge weights
graph = {
```

```
    1: {2: 4, 5: -5, 3: 2},
    2: {5: 8, 4: 2},
    3: {2: 5},
    4: {1: 3, 3: -6},
    5: {4: 6}
}

# List of vertices in the graph
vertices = [1, 2, 3, 4, 5]

# Run the algorithm and get the shortest path matrix
D = floyd_warshall(graph, vertices)

# Print the shortest path matrix D
print("Shortest Path Matrix (D):")
for row in D:
    print(row)
```

Output:

```
Shortest Path Matrix (D):
[0, 0, -5, 1, -5]
[5, 0, -4, 2, 0]
[10, 5, 0, 7, 5]
[3, -1, -6, 0, -2]
[9, 5, 0, 6, 0]
```

Q3. Implement the all pair shortest path algorithms using different graphs

---

1. Floyd-Warshall Algorithm Implementation

This implementation computes the shortest paths between all pairs of vertices using the Floyd-Warshall algorithm. We will work with an adjacency matrix representation of the graph.

Code:

```
import sys

# Floyd-Warshall Algorithm to find shortest paths for all pairs
```

```
def floyd_warshall(graph, num_vertices):
    # Initialize distance matrix D
    D = [[sys.maxsize] * num_vertices for _ in range(num_vertices)]

    # Set the diagonal to 0 (distance from vertex to itself)
    for i in range(num_vertices):
        D[i][i] = 0

    # Fill the matrix with direct distances from the graph
    for u in range(num_vertices):
        for v in range(num_vertices):
            if graph[u][v] != 0:
                D[u][v] = graph[u][v]

    # Apply the Floyd-Warshall Algorithm
    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                D[i][j] = min(D[i][j], D[i][k] + D[k][j])

    return D

# Example graph as an adjacency matrix (0 means no edge)
graph1 = [
    [0, 3, 0, 0, 0],
    [3, 0, 1, 6, 0],
    [0, 1, 0, 2, 4],
    [0, 6, 2, 0, 1],
    [0, 0, 4, 1, 0]
]

num_vertices1 = 5

# Running the Floyd-Warshall Algorithm
shortest_paths1 = floyd_warshall(graph1, num_vertices1)
```

```
# Print the shortest path matrix
print("Shortest Path Matrix (Floyd-Warshall) for Graph 1:")
for row in shortest_paths1:
    print(row)
```

Output:

```
Shortest Path Matrix (Floyd-Warshall) for Graph 1:
[0, 3, 4, 6, 7]
[3, 0, 1, 3, 4]
[4, 1, 0, 2, 3]
[6, 3, 2, 0, 1]
[7, 4, 3, 1, 0]
```

---

2. Dijkstra's Algorithm Implementation

For this implementation, we will use Dijkstra's algorithm to find the shortest path from a single vertex to all other vertices. We will run Dijkstra's algorithm for each vertex as the source and construct the full APSP matrix.

Python Code for Dijkstra's Algorithm:

```
import heapq

# Dijkstra's algorithm for finding the shortest path from a source vertex to all other vertices
def dijkstra(graph, start, num_vertices):
    dist = [float('inf')] * num_vertices
    dist[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_dist, u = heapq.heappop(priority_queue)

        # If the current node's distance is already larger than the already found shortest path, skip it
        if current_dist > dist[u]:
            continue

        # Iterate through all neighbors of u
        for v in range(num_vertices):
```

```
            if graph[u][v] != 0:  # There is an edge from u to v
                weight = graph[u][v]
                if dist[u] + weight < dist[v]:
                    dist[v] = dist[u] + weight
                    heapq.heappush(priority_queue, (dist[v], v))

    return dist

# Function to run Dijkstra for all pairs of vertices
def all_pairs_dijkstra(graph, num_vertices):
    apsp_matrix = []
    for i in range(num_vertices):
        apsp_matrix.append(dijkstra(graph, i, num_vertices))
    return apsp_matrix

# Example graph as an adjacency matrix (0 means no edge)
graph2 = [
    [0, 10, 0, 0, 0],
    [10, 0, 5, 0, 0],
    [0, 5, 0, 15, 10],
    [0, 0, 15, 0, 0],
    [0, 0, 10, 0, 0]
]

num_vertices2 = 5

# Running the Dijkstra's Algorithm for all pairs
shortest_paths2 = all_pairs_dijkstra(graph2, num_vertices2)

# Print the shortest path matrix
print("\nShortest Path Matrix (Dijkstra) for Graph 2:")
for row in shortest_paths2:
    print(row)
```

Output:

Shortest Path Matrix (Dijkstra) for Graph 2:

[0, 10, 15, 30, 25]

[10, 0, 5, 20, 15]

[15, 5, 0, 15, 10]

[30, 20, 15, 0, 25]

[25, 15, 10, 25, 0]

Session 10: Chained Matrix Multiplication

Q1. List different orders for evaluating the product of A,B,C,D,E matrices.

When multiplying a sequence of matrices, the order in which you multiply them matters, as matrix multiplication is **associative**, but not **commutative**. This means that the product A×B×C×D×E can be evaluated in different ways, resulting in different computational costs.

To find the optimal order of matrix multiplication, we typically use **dynamic programming** (as done in the **Matrix Chain Multiplication** problem), but here we'll list all possible orders for evaluating the product of 5 matrices, A, B, C, D, and E.

Number of Possible Parenthesizations:

For nn matrices, the number of different ways to parenthesize the product is given by the **Catalan number** C_{n-1}, which is the number of ways to fully parenthesize the product.

For 5 matrices, there are **14 possible ways** to parenthesize the product.

Possible Orders for Evaluating A×B×C×D×E:

1. (A×(B×(C×(D×E))))
2. ((A×(B×(C×D)))×E)
3. ((A×(B×C))×(D×E))
4. (A×((B×C)×(D×E)))
5. (A×((B×(C×D))×E))
6. ((A×B)×(C×(D×E)))
7. ((A×(B×C))×D)×E
8. (A×((B×C)×D))×E
9. ((A×(B×C))×D)×E
10. ((A×B)×(C×D))×E
11. ((A×(B×(C×D)))×E)
12. ((A×B)×(C×(D×E)))
13. (A×((B×C)×D))×E
14. (A×B)×(C×(D×E))

Total Possible Ways to Parenthesize:

These 14 orders represent all possible ways in which the matrices can be parenthesized for matrix multiplication. The correct order of multiplication can be crucial in minimizing the number of scalar multiplications. This is where dynamic programming algorithms come in handy to determine the most efficient multiplication order based on matrix dimensions.

Explanation of Notation:

- Parentheses indicate the order of matrix multiplication.
- The computation proceeds from the innermost parentheses outwards. For instance, (A×(B×C)) means that B and C are multiplied first, and then A is multiplied with the result.

Optimal Parenthesization:

To determine the optimal parenthesization (i.e., the one that minimizes the number of scalar multiplications), you would need the dimensions of the matrices. If the dimensions are not given, it is impossible to tell which of the 14 parenthesizations is optimal purely from the order itself.

Q2. Find the optimal order for evaluating the product A * B * C *D *E where

- A is (10 * 4)
- B is (4 *5)
- C is (5 * 20)
- D is (20 * 2)
- E is (2* 50)

To determine the **optimal order** for evaluating the matrix product A×B×C×D×EA \times B \times C \times D \times E, we can apply **dynamic programming** based on the dimensions of the matrices.

Matrix Dimensions:

- Matrix AA is 10×410 \times 4
- Matrix BB is 4×54 \times 5
- Matrix CC is 5×205 \times 20
- Matrix DD is 20×220 \times 2
- Matrix EE is 2×502 \times 50

We want to minimize the total number of scalar multiplications required to compute the product. The number of scalar multiplications required to multiply two matrices is the product of their dimensions. For example, multiplying a matrix of dimension p×qp \times q with a matrix of dimension q×rq \times r results in a matrix of dimension p×rp \times r, and it requires p×q×rp \times q \times r scalar multiplications.

Step 1: Define the Problem

Let the matrices have the following dimensions:

- A(10×4)
- B(4×5)
- C(5×20)
- D(20×2)
- E(2×50)

We need to determine the optimal parenthesization to minimize the number of scalar multiplications for the product A×B×C×D×E.

Step 2: Set Up the Dynamic Programming Table

We define an array m where m[i][j] will store the minimum number of scalar multiplications required to compute the product of matrices $M_i \times M_{i+1} \times \cdots \times M_j$.

Step 3: Algorithm for Matrix Chain Multiplication

We will use the following approach:

1. Initialize m[i][i]=0 for all i, since the cost of multiplying a single matrix is zero.
2. For chains of length l= 2 to n, calculate the minimum cost for each chain and store it in m[i][j].
3. The formula to compute the number of scalar multiplications for a chain is:

$$m[i][j] = \min_{(i \le k < j)} (m[i][k] + m[k+1][j] + p_i \times p_k \times p_j)$$

Where:

- p_i, p_k, p_j are the matrix dimensions, i.e., matrix M_i is p_i x p_{i+1}, and so on.
- m[i][j] is the minimum number of scalar multiplications required to multiply matrices M_i through M_j.

Step 4: Apply the Matrix Chain Multiplication Algorithm

Dimensions:

- A is 10×4
- B is 4× 5
- C is 5× 20
- D is 20× 2
- E is 2× 50

So, the dimension array p=[10,4,5,20,2,50].

Step 5: Dynamic Programming Calculation

Let's calculate the number of scalar multiplications for each pair of matrices using dynamic programming.

```
import numpy as np


# Function to compute the minimum number of scalar multiplications and track the computation
def matrix_chain_order(p):
    n = len(p) - 1  # number of matrices
    m = np.zeros((n, n), dtype=float)  # m[i][j] stores the minimum cost for multiplying A_i to A_j
    s = np.zeros((n, n), dtype=int)    # s[i][j] stores the index of the split point

    # Initialize the matrix with infinity
    for i in range(n):
        for j in range(n):
            if i != j:
```

```
            m[i][j] = float('inf')  # Set all values to infinity except the diagonal

    # l is the chain length
    for length in range(2, n+1):  # Lengths of subproblems (chain lengths from 2 to n)
        print(f"\nEvaluating chains of length {length}:")
        for i in range(n - length + 1):
            j = i + length - 1
            print(f"\tCalculating m[{i}][{j}]...")  # Start evaluating subproblem m[i][j]
            # Try all possible splits
            for k in range(i, j):
                q = m[i][k] + m[k+1][j] + p[i] * p[k+1] * p[j+1]
                print(f"\t\tTrying k = {k}: Cost = {m[i][k]} + {m[k+1][j]} + ({p[i]} * {p[k+1]} * {p[j+1]}) = {q}")
                if q < m[i][j]:
                    m[i][j] = q
                    s[i][j] = k  # Record the split point
                    print(f"\t\tUpdated m[{i}][{j}] = {m[i][j]} with split at {k}")

    return m, s

# Function to print the optimal order of matrix multiplication using the split table
def print_optimal_parenthesization(s, i, j):
    if i == j:
        return f"A{i}"
    else:
        k = s[i][j]
        left = print_optimal_parenthesization(s, i, k)
        right = print_optimal_parenthesization(s, k + 1, j)
        return f"({left} x {right})"

# Define the dimensions of the matrices
p = [10, 4, 5, 20, 2, 50]

# Call the function to compute the minimum number of scalar multiplications and the split table
m, s = matrix_chain_order(p)
```

```
# Print the final result and the optimal multiplication order
print(f"\nMinimum number of scalar multiplications: {m[0][len(p)-2]}")
print("\nOptimal parenthesization:")
print(print_optimal_parenthesization(s, 0, len(p)-2))
```

Output:

```
Evaluating chains of length 2:
    Calculating m[0][1]...
        Trying k = 0: Cost = 0.0 + 0.0 + (10 * 4 * 5) = 200.0
        Updated m[0][1] = 200.0 with split at 0
    Calculating m[1][2]...
        Trying k = 1: Cost = 0.0 + 0.0 + (4 * 5 * 20) = 400.0
        Updated m[1][2] = 400.0 with split at 1
    Calculating m[2][3]...
        Trying k = 2: Cost = 0.0 + 0.0 + (5 * 20 * 2) = 200.0
        Updated m[2][3] = 200.0 with split at 2
    Calculating m[3][4]...
        Trying k = 3: Cost = 0.0 + 0.0 + (20 * 2 * 50) = 2000.0
        Updated m[3][4] = 2000.0 with split at 3

Evaluating chains of length 3:
    Calculating m[0][2]...
        Trying k = 0: Cost = 0.0 + 400.0 + (10 * 4 * 20) = 1200.0
        Updated m[0][2] = 1200.0 with split at 0
        Trying k = 1: Cost = 200.0 + 0.0 + (10 * 5 * 20) = 1200.0
    Calculating m[1][3]...
        Trying k = 1: Cost = 0.0 + 200.0 + (4 * 5 * 2) = 240.0
        Updated m[1][3] = 240.0 with split at 1
        Trying k = 2: Cost = 400.0 + 0.0 + (4 * 20 * 2) = 560.0
    Calculating m[2][4]...
        Trying k = 2: Cost = 0.0 + 2000.0 + (5 * 20 * 50) = 7000.0
        Updated m[2][4] = 7000.0 with split at 2
        Trying k = 3: Cost = 200.0 + 0.0 + (5 * 2 * 50) = 700.0
        Updated m[2][4] = 700.0 with split at 3

Evaluating chains of length 4:
```

Calculating m[0][3]...

Trying k = 0: Cost = 0.0 + 240.0 + (10 * 4 * 2) = 320.0

Updated m[0][3] = 320.0 with split at 0

Trying k = 1: Cost = 200.0 + 200.0 + (10 * 5 * 2) = 500.0

Trying k = 2: Cost = 1200.0 + 0.0 + (10 * 20 * 2) = 1600.0

Calculating m[1][4]...

Trying k = 1: Cost = 0.0 + 700.0 + (4 * 5 * 50) = 1700.0

Updated m[1][4] = 1700.0 with split at 1

Trying k = 2: Cost = 400.0 + 2000.0 + (4 * 20 * 50) = 6400.0

Trying k = 3: Cost = 240.0 + 0.0 + (4 * 2 * 50) = 640.0

Updated m[1][4] = 640.0 with split at 3

Evaluating chains of length 5:

Calculating m[0][4]...

Trying k = 0: Cost = 0.0 + 640.0 + (10 * 4 * 50) = 2640.0

Updated m[0][4] = 2640.0 with split at 0

Trying k = 1: Cost = 200.0 + 700.0 + (10 * 5 * 50) = 3400.0

Trying k = 2: Cost = 1200.0 + 2000.0 + (10 * 20 * 50) = 13200.0

Trying k = 3: Cost = 320.0 + 0.0 + (10 * 2 * 50) = 1320.0

Updated m[0][4] = 1320.0 with split at 3

Minimum number of scalar multiplications: 1320.0

Optimal parenthesization:

((A0 x (A1 x (A2 x A3))) x A4)

Q3. Implement the chained matrix multiplication and print the optimal parentheses algorithms and study the performance of the algorithms on different problem instances.

To implement **Chained Matrix Multiplication** with **Optimal Parenthesization**, we need to use the **Matrix Chain Multiplication** algorithm. This involves dynamic programming to find the minimum number of scalar multiplications required to multiply a chain of matrices. Additionally, we need to print the optimal parenthesization of the matrices.

Steps:

1. **Dynamic Programming Table (m)**: This table stores the minimum number of scalar multiplications needed for multiplying matrices from AiA_i to AjA_j.
2. **Split Table (s)**: This table stores the index kk where the matrix chain should be split for optimal multiplication.

The algorithm computes:

- The minimum number of scalar multiplications.

- The optimal parenthesization.

Algorithm:

1. **Matrix Chain Multiplication** using dynamic programming.
2. **Print Optimal Parenthesization** using the split table.
3. **Study the Performance** by testing with different problem instances.

Code Implementation:

```
import numpy as np

# Function to compute the minimum number of scalar multiplications and the optimal parenthesization
def matrix_chain_order(p):
    n = len(p) - 1  # number of matrices
    m = np.zeros((n, n), dtype=float)  # m[i][j] stores the minimum cost for multiplying A_i to A_j
    s = np.zeros((n, n), dtype=int)    # s[i][j] stores the index of the split point

    # Initialize the matrix with infinity for non-diagonal elements
    for i in range(n):
        for j in range(n):
            if i != j:
                m[i][j] = float('inf')  # Set all values to infinity except the diagonal

    # l is the chain length
    for length in range(2, n+1):  # Lengths of subproblems (chain lengths from 2 to n)
        for i in range(n - length + 1):
            j = i + length - 1
            # Try all possible splits
            for k in range(i, j):
                q = m[i][k] + m[k+1][j] + p[i] * p[k+1] * p[j+1]
                if q < m[i][j]:
                    m[i][j] = q
                    s[i][j] = k  # Record the split point

    return m, s

# Function to print the optimal order of matrix multiplication using the split table
def print_optimal_parenthesization(s, i, j):
```

```
    if i == j:
        return f"A{i+1}"
    else:
        k = s[i][j]
        left = print_optimal_parenthesization(s, i, k)
        right = print_optimal_parenthesization(s, k + 1, j)
        return f"({left} x {right})"

# Function to evaluate and print performance on different problem instances
def evaluate_performance(p):
    print(f"Evaluating matrix chain multiplication for dimensions: {p}")

    # Compute minimum scalar multiplications and optimal parenthesization
    m, s = matrix_chain_order(p)

    # Print the final result and the optimal multiplication order
    print(f"\nMinimum number of scalar multiplications: {m[0][len(p)-2]}")
    print("\nOptimal parenthesization:")
    print(print_optimal_parenthesization(s, 0, len(p)-2))

# Problem instances with different matrix dimensions
problem_instances = [
    [10, 4, 5, 20, 2, 50],  # Instance 1: 5 matrices
    [30, 35, 15, 5, 10, 20, 25],  # Instance 2: 6 matrices
    [50, 20, 1, 10, 100],  # Instance 3: 4 matrices
    [5, 10, 20, 15, 25]  # Instance 4: 4 matrices
]

# Evaluate performance for different problem instances
for i, p in enumerate(problem_instances, 1):
    print(f"\nProblem Instance {i}:")
    evaluate_performance(p)
```

Explanation:

1. **Matrix Chain Order**: This function computes the minimum number of scalar multiplications needed for a given chain of matrices.

- The matrix m stores the minimum number of scalar multiplications for each subproblem.
- The matrix s stores the index where the matrix chain is split for optimal multiplication.

2. **Optimal Parenthesization**: Using the split matrix s, we recursively construct and print the optimal parenthesization by dividing the matrix chain based on the split points.
3. **Evaluate Performance**: This function evaluates the performance of the algorithm on different problem instances, prints the minimum scalar multiplications, and displays the optimal parenthesization.

Example Problem Instances:

1. **Problem Instance 1**: Dimensions: [10, 4, 5, 20, 2, 50] (5 matrices)
2. **Problem Instance 2**: Dimensions: [30, 35, 15, 5, 10, 20, 25] (6 matrices)
3. **Problem Instance 3**: Dimensions: [50, 20, 1, 10, 100] (4 matrices)
4. **Problem Instance 4**: Dimensions: [5, 10, 20, 15, 25] (4 matrices)

Output:

Problem Instance 1:

Evaluating matrix chain multiplication for dimensions: [10, 4, 5, 20, 2, 50]

Minimum number of scalar multiplications: 1320.0

Optimal parenthesization:

((A1 x (A2 x (A3 x A4))) x A5)

Problem Instance 2:

Evaluating matrix chain multiplication for dimensions: [30, 35, 15, 5, 10, 20, 25]

Minimum number of scalar multiplications: 15125.0

Optimal parenthesization:

((A1 x (A2 x A3)) x ((A4 x A5) x A6))

Problem Instance 3:

Evaluating matrix chain multiplication for dimensions: [50, 20, 1, 10, 100]

Minimum number of scalar multiplications: 7000.0

Optimal parenthesization:

((A1 x A2) x (A3 x A4))

Problem Instance 4:

Evaluating matrix chain multiplication for dimensions: [5, 10, 20, 15, 25]

Minimum number of scalar multiplications: 4375.0

Optimal parenthesization:

(((A1 x A2) x A3) x A4)

Performance Study:

You can evaluate how the performance (in terms of minimum scalar multiplications) varies across different problem instances. Larger chains of matrices lead to higher computational complexity due to the increased number of splits to consider.

- **Smaller instances**: For smaller chains, the computation is faster and the optimal solution is computed quickly.
- **Larger instances**: As the number of matrices increases, the dynamic programming approach evaluates more subproblems, leading to longer execution times.

This algorithm helps us understand how optimal parenthesization can drastically reduce the computational cost of matrix chain multiplication, especially as the problem size grows.

Session 11: Optimal Binary Search Tree

Determine the cost and structure of an optimal binary search tree for a set of n=7 keys with the following properties. Show the step by step processes.

Q1.

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

To solve for the Optimal Binary Search Tree (OBST) for the given set of 7 keys, we need to follow the dynamic programming approach and compute both the cost table and root table.

Problem Setup:

- **n=7n = 7, meaning there are 7 keys, labeled k1,k2,…,k7k_1, k_2, …, k_7.**
- **The probabilities of accessing each key are given by:**
  - **p_0 = 0.04, p_1 = 0.06, p_2 = 0.08, p_3 = 0.02, p_4 = 0.10, p_5 = 0.12, p_6 = 0.14**
- **The probabilities of failing to access each key (i.e., the probability of a "null" node or unsuccessful search) are:**
  - **q_0 = 0.06, q_1 = 0.06, q_2 = 0.06, q_3 = 0.06, q_4 = 0.05, q_5 = 0.05, q_6 = 0.05, q_7 = 0.05**

Step 1: Formulation of the DP Tables

We define two tables:

- **cost[i][j]: Minimum search cost for the sub-tree with keys k_i through k_j.**
- **root[i][j]: The root of the optimal binary search tree for the keys from k_i to k_j.**

We initialize the base case where each cost[i][i] is simply the sum of the probabilities for that particular key and its associated null search probabilities (i.e., q_i + p_i + q_{i+1}).

Step 2: Base Case Calculation

For each key k_i, the cost of a tree consisting only of that key is: cost[i][i] = q_{i-1} + p_i + q_i where p_i is the probability of accessing key k_i, and q_{i-1} and q_i are the failure probabilities before and after the key.

Step 3: Recursive Case

For each subproblem (range of keys), we need to compute the minimum cost by trying all possible roots in the range k_i to k_j and recursively solving the cost for the left and right subtrees.

Step 4: Fill the Tables

Let's compute the cost and root tables step by step:

1. Initialize Base Case:

We compute the base cases where i = j, i.e., a single key subtree.

cost[i][i] = q_{i-1} + p_i + q_i

For i = 1 to 77, the values of cost[i][i] are:

- **cost[0][0]=0.06+0.04+0.06=0.16**
- **cost[1][1]=0.06+0.06+0.06=0.18**
- **cost[2][2]=0.06+0.08+0.06=0.20**
- **cost[3][3]=0.06+0.02+0.06=0.14**
- **cost[4][4]=0.05+0.10+0.05=0.20**
- **cost[5][5]=0.05+0.12+0.05=0.22**
- **cost[6][6]=0.05+0.14+0.05=0.24**

2. Fill Tables for Larger Subtrees:

Next, we calculate the values of cost[i][j] for larger subtrees. For each range, we calculate the total probability and compute the cost for each possible root r, where the root divides the range into left and right subtrees.

Example Calculations for Larger Ranges:

Let’s calculate for the range k_0 to k_2.

- **The total probability for keys k_0, k_1, k_2 is: total_prob=q0+p0+p1+p2+q3=0.06+0.04+0.06+0.08+0.06=0.30**

Now we calculate the costs for each possible root:

- **For root k_0:**

  cost[0][2]=cost[0][0]+cost[1][2]+0.30=0.16+0.22+0.30=0.68

  For root k_1:

  cost[0][2]=cost[0][1]+cost[2][2]+0.30=0.18+0.20+0.30=0.68

For root k_2:

cost[0][2]=cost[0][1]+cost[3][2]+0.30=0.18+0.14+0.30=0.62

So, the optimal root for this range is k_2, and the cost is 0.62.

3. Repeat the Calculation:

We continue the same process for all other ranges, increasing the length of the range and trying all possible roots.

Step 5: Final Results

Once we fill out the cost and root tables, we find:

- **The minimum cost for the entire tree will be stored in cost[0][6] (which covers all keys from k_0 to k_6).**
- **The root table root[0][6] will tell us which key is the root of the optimal binary search tree for all keys.**

Code:

```
def optimal_bst(p, q, n):
    # Initialize the tables
    e = [[0] * (n + 1) for _ in range(n + 1)]  # e[i][j]: cost of optimal BST for keys i to j
    w = [[0] * (n + 1) for _ in range(n + 1)]  # w[i][j]: weight of BST for keys i to j
    root = [[0] * (n + 1) for _ in range(n + 1)]  # root[i][j]: optimal root for keys i to j

    # Base case initialization for a single key (i = j)
    for i in range(1, n + 1):
        e[i][i] = q[i - 1] + p[i - 1] + q[i]
        w[i][i] = p[i - 1] + q[i - 1] + q[i]

    # Fill in the tables for ranges of keys greater than 1
    for length in range(2, n + 1):  # length of the range of keys
        for i in range(1, n - length + 2):  # starting index of the range
            j = i + length - 1  # ending index of the range
            e[i][j] = float('inf')  # Set initial cost to infinity
            w[i][j] = w[i][j - 1] + p[j - 1] + q[j]  # Update weight

            # Try every possible root
            for r in range(i, j + 1):
                # Check boundaries for e[i][r-1] and e[r+1][j]
                left_cost = e[i][r - 1] if r > i else 0
                right_cost = e[r + 1][j] if r < j else 0
```

```
            cost = left_cost + right_cost + w[i][j]
            if cost < e[i][j]:
                e[i][j] = cost
                root[i][j] = r

    # Return the minimum cost of the BST and the root structure
    return e[1][n], root

# Given probabilities
p = [0.04, 0.06, 0.08, 0.02, 0.10, 0.12, 0.14]
q = [0.06, 0.06, 0.06, 0.06, 0.05, 0.05, 0.05, 0.05]
n = len(p)

# Compute optimal BST cost and structure
cost, root = optimal_bst(p, q, n)

print("Minimum cost of the optimal BST:", cost)
print("Root structure of the optimal BST:", root)
```

Final Output:

Minimum cost of the optimal BST: 2.68

Root structure of the optimal BST: [[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 2, 2, 2, 3, 3, 5], [0, 0, 0, 3, 3, 3, 5, 5], [0, 0, 0, 0, 3, 4, 5, 5], [0, 0, 0, 0, 0, 5, 5, 6], [0, 0, 0, 0, 0, 0, 6, 6], [0, 0, 0, 0, 0, 0, 0, 7], [0, 0, 0, 0, 0, 0, 0, 0]].

Q2.

I	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Input:

p = [0.04, 0.06, 0.08, 0.02, 0.10, 0.12, 0.14]

q = [0.06, 0.06, 0.06, 0.06, 0.05, 0.05, 0.05, 0.05]

Output:

Minimum cost of the optimal BST: 2.3500000000000005

Root structure of the optimal BST: [[0, 0, 0, 0, 0, 0], [0, 0, 1, 2, 2, 2], [0, 0, 0, 2, 2, 4], [0, 0, 0, 0, 4, 5], [0, 0, 0, 0, 0, 5], [0, 0, 0, 0, 0, 0]]

Q3 Implement the optimal binary search tree algorithm on your system and study the performance of the algorithm on different problem instances .

The **Optimal Binary Search Tree (OBST)** algorithm constructs a binary search tree where the average search cost is minimized. It is typically solved using dynamic programming.

To implement the **Optimal Binary Search Tree** algorithm, we use the following steps:

1. **Problem Definition:** Given a set of keys and their corresponding probabilities of being accessed, the goal is to construct a binary search tree that minimizes the expected search cost.
2. **Dynamic Programming Approach:** The expected search cost for a tree with keys k_i to k_j is calculated recursively. We want to find the root that minimizes the total cost (which is the sum of the costs for the left and right subtrees plus the cost of accessing the root).

Steps to Implement OBST:

1. **Input:**
   - A list of keys K = {k_1, k_2, ..., k_n}.
   - A corresponding list of probabilities P = \{p_1, p_2, ..., p_n\}, where pip_i represents the probability of accessing key kik_i.
2. **Dynamic Programming Table Construction:**
   - Let cost[i][j] represent the minimum cost for a subtree containing keys from k_i to k_j.
   - Let root[i][j] represent the root of the subtree that minimizes the search cost for the keys from k_i to k_j.
3. **Base Case:** The cost of a single key is simply the probability of accessing that key.
4. **Recursive Case:** For each range of keys, compute the minimum cost by trying each key as the root and calculating the total cost.

Python code for the **Optimal Binary Search Tree (OBST)**:

```
def optimal_bst(keys, prob, n):
    # cost[i][j] will store the minimum cost of binary search tree
    # that can be constructed from keys[i..j]
    cost = [[0 for x in range(n)] for y in range(n)]

    # root[i][j] will store the index of the root of subtree keys[i..j]
    root = [[0 for x in range(n)] for y in range(n)]

    # Fill the diagonal entries (cost of single keys)
    for i in range(n):
        cost[i][i] = prob[i]
        root[i][i] = i

    # Now fill the cost and root tables for chains of length 2 to n
    for length in range(2, n+1):
```

```
        for i in range(n-length+1):
            j = i + length - 1
            cost[i][j] = float('inf')
            total_prob = sum(prob[k] for k in range(i, j+1))  # Total probability for keys[i..j]

            # Try all possible roots for the current subproblem
            for r in range(i, j+1):
                # Calculate cost when r is the root of the subtree
                c = (cost[i][r-1] if r > i else 0) + (cost[r+1][j] if r < j else 0) + total_prob

                # If this cost is smaller, update the cost and root
                if c < cost[i][j]:
                    cost[i][j] = c
                    root[i][j] = r

    return cost[0][n-1], root

# Utility function to print the constructed OBST
def print_bst(root, i, j, level=0):
    if i > j:
        return
    r = root[i][j]
    print("  " * level * 4 + f"Key {keys[r]} is root at level {level}")
    print_bst(root, i, r-1, level+1)
    print_bst(root, r+1, j, level+1)

# Example usage
keys = [10, 12, 20]
prob = [0.34, 0.08, 0.50]
n = len(keys)

min_cost, root = optimal_bst(keys, prob, n)
print(f"Minimum cost of the OBST: {min_cost}")
print("The structure of the optimal binary search tree:")
print_bst(root, 0, n-1)
```

Explanation of the Code:

1. **Input:**
    - keys: The list of keys that will be stored in the BST.
    - prob: The list of probabilities associated with each key.
2. **Tables:**
    - cost[i][j]: The minimum cost of the BST that can be constructed from keys kik_i to kjk_j.
    - root[i][j]: The index of the root of the subtree containing keys from kik_i to kjk_j.
3. **Dynamic Programming Approach:**
    - We fill the tables for chains of length 1 to nn (where nn is the number of keys).
    - For each subproblem, we calculate the minimum cost by trying all possible roots and picking the one that gives the least cost.
4. **Output:**

Minimum cost of the OBST: 1.42

The structure of the optimal binary search tree:

Key 20 is root at level 0

  Key 10 is root at level 1

    Key 12 is root at level 2

Performance Study:

To study the performance, you can experiment with different sets of keys and prob to observe how the algorithm scales with increasing input sizes. You can time the execution for larger datasets to evaluate the algorithm's efficiency.

SECTION-2: WEB DESIGN LAB

Session 1: Preliminaries

---

1. Write a JavaScript code to perform the two-digit arithmetic operations: Addition, Subtraction, Multiplication and Division.

Code:

```
// Function to perform addition
function add(num1, num2) {
  return num1 + num2;
}

// Function to perform subtraction
function subtract(num1, num2) {
```

```
    return num1 - num2;
}

// Function to perform multiplication
function multiply(num1, num2) {
    return num1 * num2;
}

// Function to perform division
function divide(num1, num2) {
    if (num2 === 0) {
        return "Error: Division by zero is not allowed!";
    }
    return num1 / num2;
}

// Taking two numbers as input
let num1 = prompt("Enter the first two-digit number: ");
let num2 = prompt("Enter the second two-digit number: ");

// Converting input strings to integers
num1 = parseInt(num1);
num2 = parseInt(num2);

// Performing the operations and displaying the results
console.log(`Addition of ${num1} and ${num2}: ${add(num1, num2)}`);
console.log(`Subtraction of ${num1} and ${num2}: ${subtract(num1, num2)}`);
console.log(`Multiplication of ${num1} and ${num2}: ${multiply(num1, num2)}`);
console.log(`Division of ${num1} and ${num2}: ${divide(num1, num2)}`);
```

Output:

```
Addition of 23 and 12: 35
Subtraction of 23 and 12: 11
Multiplication of 23 and 12: 276
Division of 23 and 12: 1.9166666666666667
```

2. Write a JavaScript code to print 1 for 1 time, 2 for 2 times, 3 for 3 times and so on upto 10.

Code:

```
// Loop to print numbers from 1 to 10
for (let i = 1; i <= 10; i++) {
   // Print the number 'i', 'i' times
   for (let j = 1; j <= i; j++) {
      console.log(i);
   }
}
```

Output:

```
1
2
2
3
3
3
4
4
4
4
5
5
5
5
5
6
6
6
6
6
6
7
7
7
```

7

7

7

7

8

8

8

8

8

8

8

8

9

9

9

9

9

9

9

9

9

9

10

10

10

10

10

10

10

10

10

10

---

3. Design a web page that describes you. Your page must include following details (extra details and creativity are welcome):

 Your personal details, i.e. name, gender etc.

 2-3 lines describing yourself as About Me. Make the most important word bold to emphasize them.

 Details of classes in tabular form you are taking right now (as per your time table).

 Details of your favourite movies, books, or TV shows, in reverse order(using CSS), list atleast 3 from each. You can use images, hyperlinks, colourful text to make this section attractive.

 Include a section showing your interest and also not of your interest jobs. In this section also you can use images, showing you're happy in doing your interest jobs and the other to represent you when you're sad.

 In this section show something interesting about one or more people of your neighbours.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>About Me</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      background-color: #f0f0f0;
      color: #333;
      margin: 0;
      padding: 0;
    }

    header {
      background-color: #4CAF50;
      color: white;
      padding: 20px;
      text-align: center;
    }

    section {
      padding: 20px;
    }

    .bold-word {
      font-weight: bold;
```

```
        color: #ff5733;
    }

    .timetable-table {
        width: 100%;
        border-collapse: collapse;
    }

    .timetable-table, th, td {
        border: 1px solid #ddd;
    }

    th, td {
        padding: 10px;
        text-align: center;
    }

    th {
        background-color: #4CAF50;
        color: white;
    }

    .favorites-list {
        list-style-type: none;
        padding: 0;
    }

    .favorites-list li {
        background-color: #f2f2f2;
        margin: 5px 0;
        padding: 10px;
        border-radius: 5px;
    }

    .reverse-order {
```

```
        direction: rtl;
    }

    .interest-jobs img {
        width: 50px;
        height: 50px;
        margin: 10px;
    }

    .interest-jobs img.happy {
        border-radius: 50%;
        border: 2px solid green;
    }

    .interest-jobs img.sad {
        border-radius: 50%;
        border: 2px solid red;
    }

    .neighbor-info {
        background-color: #e0f7fa;
        padding: 20px;
        margin-top: 20px;
        border-radius: 5px;
    }

    footer {
        background-color: #4CAF50;
        color: white;
        text-align: center;
        padding: 10px;
        position: fixed;
        width: 100%;
        bottom: 0;
    }
```

```
    </style>
</head>
<body>

<header>
    <h1>Welcome to My Personal Page</h1>
</header>

<section>
    <h2>Personal Details</h2>
    <p><strong>Name:</strong> Giridharan B Venkatesh Murthy</p>
    <p><strong>Gender:</strong> Male</p>
    <p><strong>Location:</strong> India</p>
</section>

<section>
    <h2>About Me</h2>
    <p>Hello! I am <span class="bold-word">Giridharan</span>, a passionate tech enthusiast. I love solving problems and constantly learning new skills, especially in programming and design. I am also into photography, music, and graphic designing!</p>
</section>

<section>
    <h2>Classes I'm Taking Right Now</h2>
    <table class="timetable-table">
        <tr>
            <th>Class Name</th>
            <th>Day</th>
            <th>Time</th>
        </tr>
        <tr>
            <td>Advanced Programming</td>
            <td>Monday</td>
            <td>10:00 AM - 12:00 PM</td>
        </tr>
        <tr>
```

```
        <td>Web Development</td>
        <td>Wednesday</td>
        <td>1:00 PM - 3:00 PM</td>
    </tr>
    <tr>
        <td>Data Science</td>
        <td>Friday</td>
        <td>9:00 AM - 11:00 AM</td>
    </tr>
  </table>
</section>

<section>
  <h2>My Favorite Movies, Books, and TV Shows</h2>

  <h3>Movies (Reverse Order)</h3>
  <ul class="favorites-list reverse-order">
    <li>Inception</li>
    <li>The Matrix</li>
    <li>Interstellar</li>
  </ul>

  <h3>Books (Reverse Order)</h3>
  <ul class="favorites-list reverse-order">
    <li>Atomic Habits</li>
    <li>The Alchemist</li>
    <li>Thinking, Fast and Slow</li>
  </ul>

  <h3>TV Shows (Reverse Order)</h3>
  <ul class="favorites-list reverse-order">
    <li>Breaking Bad</li>
    <li>Stranger Things</li>
    <li>Black Mirror</li>
  </ul>
```

```
</section>

<section>
    <h2>Jobs I Am Interested In vs. Jobs I Am Not Interested In</h2>

    <div class="interest-jobs">
        <h3>Jobs I Love</h3>
        <img src="https://via.placeholder.com/50" class="happy" alt="Happy">
        <img src="https://via.placeholder.com/50" class="happy" alt="Happy">
        <img src="https://via.placeholder.com/50" class="happy" alt="Happy">
        <p>I love jobs related to programming, problem-solving, and web development.</p>

        <h3>Jobs I Don't Like</h3>
        <img src="https://via.placeholder.com/50" class="sad" alt="Sad">
        <img src="https://via.placeholder.com/50" class="sad" alt="Sad">
        <img src="https://via.placeholder.com/50" class="sad" alt="Sad">
        <p>I'm not interested in jobs related to manual labor or repetitive tasks.</p>
    </div>
</section>

<section class="neighbor-info">
    <h2>Interesting Neighbor Facts</h2>
    <p>My neighbor, Mr. Sharma, is a retired scientist who has an impressive collection of rare books on space science. He loves sharing interesting facts about the universe with anyone who is curious.</p>
</section>

<footer>
    <p>&copy; 2025 Giridharan B Venkatesh Murthy</p>
</footer>

</body>
</html>
```

Question paper:

Implement multiplication of two matrices A[4, 4] and B[4, 4] and calculate following:

(i) How many times the innermost and the outermost loop will run?

(ii) Total number of multiplication and additions in computing the multiplication of given matrices.

Code:

```
import numpy as np

# Create 4x4 matrices A and B
A = np.random.randint(1, 10, size=(4, 4))
B = np.random.randint(1, 10, size=(4, 4))

# Initialize a result matrix C with zeros
C = np.zeros((4, 4))

# Initialize counters for multiplications and additions
multiplications = 0
additions = 0

# Matrix multiplication
for i in range(4):
    for j in range(4):
        for k in range(4):
            C[i][j] += A[i][k] * B[k][j]
            multiplications += 1
            if k > 0:  # For each k after the first one, we perform an addition
                additions += 1

print("Matrix A:")
print(A)
print("\nMatrix B:")
print(B)
print("\nMatrix C (Result):")
print(C)

# Analysis
innermost_loop_runs = 4 * 4 * 4
outermost_loop_runs = 4 * 4
```

```
total_multiplications = multiplications
total_additions = additions

print("\n(i) How many times the innermost and the outermost loop will run?")
print(f"Innermost loop runs: {innermost_loop_runs} times")
print(f"Outermost loop runs: {outermost_loop_runs} times")

print("\n(ii) Total number of multiplication and additions in computing the multiplication of given matrices:")
print(f"Total multiplications: {total_multiplications}")
print(f"Total additions: {total_additions}")
```

Output:

```
Matrix A:
[[3 3 5 4]
 [8 1 7 3]
 [7 7 6 8]
 [7 1 3 2]]

Matrix B:
[[2 8 1 7]
 [3 5 9 1]
 [9 2 6 1]
 [5 4 4 5]]

Matrix C (Result):
[[ 80.  65.  76.  49.]
 [ 97.  95.  71.  79.]
 [129. 135. 138. 102.]
 [ 54.  75.  42.  63.]]

(i) How many times the innermost and the outermost loop will run?
Innermost loop runs: 64 times
Outermost loop runs: 16 times
```

(ii) Total number of multiplication and additions in computing the multiplication of given matrices:

Total multiplications: 64

Total additions: 48

Design a form for booking room through a Hotel website. The form should have relevant fields (make suitable assumptions). Further, the form should have Submit and Reset button. Now, perform following: 20

(a) Use java script to validate all the fields in the form.

(b) Submit button should enter the data of fields in the database.

(c) Error message should be shown if text field is left blank.

(d) Reset button should reset all the fields to blank.

Code:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Hotel Room Booking Form</title>
  <script type="text/javascript">
    // JavaScript Validation function
    function validateForm() {
        var name = document.getElementById("name").value;
        var email = document.getElementById("email").value;
        var phone = document.getElementById("phone").value;
        var checkin = document.getElementById("checkin").value;
        var checkout = document.getElementById("checkout").value;
        var rooms = document.getElementById("rooms").value;

        // Check if fields are empty
        if (name == "" || email == "" || phone == "" || checkin == "" || checkout == "" || rooms == "") {
            alert("Please fill out all the fields.");
            return false; // Prevent form submission
        }

        // Check if phone number is valid (simple validation for 10 digits)
        var phonePattern = /^[0-9]{10}$/;
```

```
        if (!phone.match(phonePattern)) {
            alert("Please enter a valid 10-digit phone number.");
            return false;
        }

        // Check if email is valid
        var emailPattern = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
        if (!email.match(emailPattern)) {
            alert("Please enter a valid email address.");
            return false;
        }

        return true; // Form is valid
    }

    // Function to handle form submission (Simulated with console log for now)
    function submitForm() {
        if (validateForm()) {
            var name = document.getElementById("name").value;
            var email = document.getElementById("email").value;
            var phone = document.getElementById("phone").value;
            var checkin = document.getElementById("checkin").value;
            var checkout = document.getElementById("checkout").value;
            var rooms = document.getElementById("rooms").value;

            // Here, you can send the form data to the database
            console.log("Form Submitted");
            console.log("Name: " + name);
            console.log("Email: " + email);
            console.log("Phone: " + phone);
            console.log("Check-in Date: " + checkin);
            console.log("Check-out Date: " + checkout);
            console.log("Rooms: " + rooms);
```

```
            // Normally, here you would send the form data to the server (using AJAX or submitting to a server-side script)
            // Example: submit the form data to a backend (using fetch or XMLHttpRequest)

            alert("Booking successfully submitted!");
            // You can add actual submission to the database here
        }
    }

    // Function to reset form fields
    function resetForm() {
        document.getElementById("bookingForm").reset();
    }
  </script>
</head>
<body>

  <h2>Hotel Room Booking Form</h2>
  <form id="bookingForm" onsubmit="event.preventDefault(); submitForm();">
    <label for="name">Full Name:</label>
    <input type="text" id="name" name="name" placeholder="Enter your full name"><br><br>

    <label for="email">Email:</label>
    <input type="email" id="email" name="email" placeholder="Enter your email address"><br><br>

    <label for="phone">Phone Number:</label>
    <input type="text" id="phone" name="phone" placeholder="Enter your phone number (10 digits)"><br><br>

    <label for="checkin">Check-in Date:</label>
    <input type="date" id="checkin" name="checkin"><br><br>

    <label for="checkout">Check-out Date:</label>
    <input type="date" id="checkout" name="checkout"><br><br>
```

```
    <label for="rooms">Number of Rooms:</label>
    <input type="number" id="rooms" name="rooms" min="1" max="10" placeholder="Enter number of rooms"><br><br>

    <input type="submit" value="Submit">
    <input type="button" value="Reset" onclick="resetForm()">
  </form>

</body>
</html>
```

Implement Bubble sort algorithm for the following list of numbers:

55, 25, 15, 40, 60, 35, 17, 65, 75, 10

show the following in the output:

(i) Number of exchange operations, performed.
(ii) Number of times comparison operation is performed.
(iii) Number of times inner and outer loop will iterate.

Code:

```
def bubble_sort(arr):
  n = len(arr)
  exchange_operations = 0
  comparison_operations = 0
  outer_loop_iterations = 0
  inner_loop_iterations = 0

  # Bubble Sort
  for i in range(n):
    outer_loop_iterations += 1
    for j in range(0, n - i - 1):
      comparison_operations += 1
      inner_loop_iterations += 1
      if arr[j] > arr[j + 1]:
        # Swap elements
        arr[j], arr[j + 1] = arr[j + 1], arr[j]
        exchange_operations += 1
```

```
    return arr, exchange_operations, comparison_operations, outer_loop_iterations,
inner_loop_iterations


# Given list of numbers

arr = [55, 25, 15, 40, 60, 35, 17, 65, 75, 10]


# Perform Bubble Sort

sorted_arr, exchange_ops, comparison_ops, outer_loops, inner_loops = bubble_sort(arr)


# Display the results

print("Sorted List:", sorted_arr)

print(f"(i) Number of exchange operations performed: {exchange_ops}")

print(f"(ii) Number of comparison operations performed: {comparison_ops}")

print(f"(iii) Number of times inner loop will iterate: {inner_loops}")

print(f"(iii) Number of times outer loop will iterate: {outer_loops}")
```

Output:

Sorted List: [10, 15, 17, 25, 35, 40, 55, 60, 65, 75]

(i) Number of exchange operations performed: 21

(ii) Number of comparison operations performed: 45

(iii) Number of times inner loop will iterate: 45

(iii) Number of times outer loop will iterate: 10

Design a form to reserve seat in a tourist bus, through its website. The forum should have relevant fields (make suitable assumptions). The form should have submit and reset button. Now, perform following:

(a) Use java script to validate all the fields in the form.

(b) Submit button should enter the data of fields in to the database.

(c) Error message should be shown if any text field is left blank.

(d) Reset button should reset all the fields to blank.

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">
```

```
  <title>Tourist Bus Seat Reservation Form</title>
  <script type="text/javascript">
    // JavaScript Validation function
    function validateForm() {
        var name = document.getElementById("name").value;
        var email = document.getElementById("email").value;
        var phone = document.getElementById("phone").value;
        var seat = document.getElementById("seat").value;
        var date = document.getElementById("date").value;
        var destination = document.getElementById("destination").value;

        // Check if fields are empty
        if (name == "" || email == "" || phone == "" || seat == "" || date == "" || destination == "") {
            alert("Please fill out all the fields.");
            return false; // Prevent form submission
        }

        // Validate phone number (10 digits)
        var phonePattern = /^[0-9]{10}$/;
        if (!phone.match(phonePattern)) {
            alert("Please enter a valid 10-digit phone number.");
            return false;
        }

        // Validate email format
        var emailPattern = /^[a-zA-Z0-9._-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,6}$/;
        if (!email.match(emailPattern)) {
            alert("Please enter a valid email address.");
            return false;
        }

        return true; // Form is valid
    }

    // Function to handle form submission (Simulated with console log for now)
```

```
    function submitForm() {
        if (validateForm()) {
            var name = document.getElementById("name").value;
            var email = document.getElementById("email").value;
            var phone = document.getElementById("phone").value;
            var seat = document.getElementById("seat").value;
            var date = document.getElementById("date").value;
            var destination = document.getElementById("destination").value;

            // Here, you can send the form data to the database (simulated)
            console.log("Form Submitted");
            console.log("Name: " + name);
            console.log("Email: " + email);
            console.log("Phone: " + phone);
            console.log("Seat Number: " + seat);
            console.log("Travel Date: " + date);
            console.log("Destination: " + destination);

            // Normally, here you would send the form data to the server (using AJAX or submitting to a server-side script)
            // Example: submit the form data to a backend (using fetch or XMLHttpRequest)

            alert("Seat reservation successfully submitted!");
            // In real-world application, form data would be submitted to a backend for processing (e.g., to a database)
        }
    }

    // Function to reset form fields
    function resetForm() {
        document.getElementById("reservationForm").reset();
    }
  </script>
</head>
<body>
```

```
    <h2>Tourist Bus Seat Reservation</h2>
    <form id="reservationForm" onsubmit="event.preventDefault(); submitForm();">
        <label for="name">Full Name:</label>
        <input type="text" id="name" name="name" placeholder="Enter your full name"><br><br>

        <label for="email">Email:</label>
        <input type="email" id="email" name="email" placeholder="Enter your email address"><br><br>

        <label for="phone">Phone Number:</label>
        <input type="text" id="phone" name="phone" placeholder="Enter your phone number (10 digits)"><br><br>

        <label for="seat">Seat Number:</label>
        <input type="number" id="seat" name="seat" min="1" max="50" placeholder="Enter seat number (1-50)"><br><br>

        <label for="date">Travel Date:</label>
        <input type="date" id="date" name="date"><br><br>

        <label for="destination">Destination:</label>
        <input type="text" id="destination" name="destination" placeholder="Enter your travel destination"><br><br>

        <input type="submit" value="Submit">
        <input type="button" value="Reset" onclick="resetForm()">
    </form>

</body>
</html>
```
