Aim :
Given N items with their corresponding weights and values, and a package of capacity C, choose either the entire item or fractional part of the item among these N unique items to fill the package such that the package has maximum value.

Description :
Knapsack Problem: Optimization of Package Value Using Greedy Approach

Solution Approach:
The Knapsack Problem can be solved using a greedy approach by sorting the items based on their value-to-weight ratio in descending order. The items with higher value-to-weight ratios are prioritized for inclusion in the package.
In the context of the Knapsack Problem, the greedy approach involves sorting the items based on their value-to-weight ratio in descending order. The rationale behind this approach is to prioritize the inclusion of items that provide the highest value per unit weight. By making greedy choices at each step, the algorithm attempts to maximize the total value obtained within the given weight capacity.

Algorithm
Sort the items: Sort the items in descending order based on their value-to-weight ratio (value / weight).
Initialize variables: Initialize the remaining weight capacity of the package to C, and the total value to 0.
Iterate through the sorted items:
If the weight of the current item is less than or equal to the remaining weight capacity, include the entire item in the package. Update the remaining weight capacity by subtracting the item's weight, and increment the total value by the item's value.
If the weight of the current item is greater than the remaining weight capacity, include a fractional part of the item in the package. The fraction to be included is equal to the remaining weight capacity divided by the item's weight. Update the remaining weight capacity to 0, and increment the total value by the value of the fractional part.
Return the total value: The maximum value that can be obtained by filling the package is stored in the total value variable.

Code :
```
class Item:
    def __init__(self, profit, weight):
        self.profit = profit
        self.weight = weight
        self.ratio = profit / weight

def knapsack(capacity : int, profits : list, weights : list):
    items = [Item(i,j) for i,j in zip(profits, weights)]
    items.sort(key = lambda x:x.ratio, reverse = True)
    value = 0
    for item in items:
        if item.weight <= capacity:
            capacity -= item.weight
            value += item.profit
        else:
            temp = item.profit  * capacity / item.weight
            value += temp
            capacity = 0
    return value

capacity = int(input("Enter the capacity of the knapsack: "))
num_items = int(input("Enter the number of items: "))

print("Enter the profits and weights as space-separated arrays:")
profits = list(map(int, input("Profits: ").split()))
weights = list(map(int, input("Weights: ").split()))

assert num_items == len(profits) == len(weights), "Ensure that the lengths of
profits and weights arrays match the number of items"

value = knapsack(capacity, profits, weights)
print(f"The maximum value that can be obtained is: {value}")
```

Output :

```
Enter the capacity of the knapsack: 40
Enter the number of items: 4
Enter the profits and weights as space-separated arrays:
Profits: 10 20 30 40
Weights: 30 10 40 20
The maximum value that can be obtained is: 67.5
```

Analysis:

Time Complexity :

The maximumValue function first sorts a vector of pairs using the sort function and the compare function as the comparator. The time complexity of the sort operation is O(n log n), where n is the size of the vector. Hence, the overall time complexity is : O(nlogn)

Space Complexity :

The space complexity of the sorting algorithm used is O(log n) due to the auxiliary space used.

Aim:
Given a bunch of projects, where every project has a deadline and associated profit if the project is finished before the deadline. It is also given that every project takes one month duration, so the minimum possible deadline for any project is 1 month. In what way the total profits can be maximized if only one project can be scheduled at a time.

Description:
This problem can be solved using a greedy approach by scheduling the projects in descending order of their profit-to-deadline ratio. This approach ensures that the projects with higher profits and shorter deadlines are prioritized, thereby maximizing the total profit while meeting the deadlines.
The algorithm is similar to a knapsack problem and requires the use of additional space , sorting etc..

Algorithm

```
Step 1: Sort the jobs in descending order of profit
Step 2: Initialize maxProfit = 0, maxDeadline = 0
Step 3: Iterate through each job in the sorted list
        For each job:
        Update maxDeadline = max(maxDeadline, job.end_time)
Step 4: Create an array slots of size maxDeadline + 1, initialized with False
Step 5: Initialize numberOfJobs = 0
Step 6: Iterate through each job in the sorted list
        For each job:
        Set x = job.end_time
        While x > 0:
        If slots[x] is False:
                Update maxProfit += job.profit
                Update numberOfJobs += 1
                Set slots[x] = True
                Break the inner loop
        Else:
                Decrement x
Step 7: Return [numberOfJobs, maxProfit]
```

Code:

```
class Job:
    def __init__(self, name, deadline, profit):
        self.name = name
        self.deadline = deadline
        self.profit = profit

def sequence(names, deadlines, profits):
    jobs = [Job(i,j,k) for i,j,k in zip (names, deadlines, profits)]
    jobs.sort(key = lambda x:x.profit, reverse = True)

    maxtime = max(deadlines)
    result = [False] * maxtime
    outjob = ['-1'] * maxtime

    for job in jobs:
        for j in range(min(maxtime - 1, job.deadline - 1), -1, -1):
            if result[j] is False:
                result[j] = True
                outjob[j] = job.name
                break
    return outjob

num_jobs = int(input("Enter the number of jobs: "))
print("Enter the names, deadlines, and profits as space-separated arrays:")
names = input("Names: ").split()
deadlines = list(map(int, input("Deadlines: ").split()))
profits = list(map(int, input("Profits: ").split()))

assert(num_jobs == len(names) == len(deadlines) == len(profits)), 'Ensure all
arrays must have the same length'

job = sequence(names, deadlines, profits)
print(f"The sequence of jobs to be executed is: {job}")
```

Output:

```
Enter the number of jobs: 5
Enter the names, deadlines, and profits as space-separated arrays:
Names: a b c d e
Deadlines: 2 1 2 1 3
Profits: 100 19 27 25 15
The sequence of jobs to be executed is: ['c', 'a', 'e']
```

Output Analysis:

The code outputs the maximum number of non-overlapping jobs that can be scheduled and the maximum profit obtainable from scheduling those jobs. The output is determined by the `jobScheduling` function, which sorts the jobs by profit and greedily schedules them without overlapping time slots.

Time Complexity :

1. Sorting jobs: O(n log n)
2. Finding max deadline: O(n)
3. Creating slots array: O(maxDeadline), which is O(n) in the worst case.
4. Scheduling jobs: O(n * maxDeadline), which is O(n^2) in the worst case.

Overall time complexity: O(n log n + n^2) = O(n^2)

Space Complexity:

1. Input jobs vector: O(n)
2. Sorted jobs vector: O(n)
3. Slots array: O(maxDeadline), which is O(n) in the worst case.

Overall space complexity: O(n)

Aim:
To find the maximum and minimum value from the given array using the divide and conquer approach.

Description:
The task is to find the maximum and minimum element in an array using the divide and conquer approach. The findMaxMin function is implemented to do the same . It achieves this by employing a divide-and-conquer strategy. The algorithm recursively breaks down the problem by splitting the array in half. It then delves deeper, finding the maximum and minimum elements in each half of the array through further recursion. Finally, it compares the maximum and minimum values obtained from both halves to determine the overall maximum and minimum elements for the entire array.

The divide and conquer approach is generally used for sorting and searching and it reduces the time complexity from O(n^2) and O(n) to O(nlogn) and O(logn) by eliminating one half of the array in each pass.

Algorithm :

```
Step 1. If low equals high:
i. Set max and min to arr[low].
ii. Return.
Step 2. If high equals low plus 1:
    i. If arr[low] is greater than arr[high]:
      1. Set max to arr[low] and min to arr[high].
    ii. Else:
      1. Set max to arr[high] and min to arr[low].
    iii. Return.
Step 3. Set mid to (low + high) / 2.
Step 4. Recursively call findMaxMin with arguments arr, low, mid, maxLeft, minLeft.
Step 5. Recursively call findMaxMin with arguments arr, mid + 1, high, maxRight, minRight.
Step 6. Set max to the maximum of maxLeft and maxRight.
Step 7. Set min to the minimum of minLeft and minRight.
```

Code:

```
def findMaxMin(arr, low, high):
    if low == high:
        return arr[low], arr[low]

    if high == low + 1:
        if arr[low] > arr[high]:
            return arr[low], arr[high]
        else:
            return arr[high], arr[low]

    mid = (low + high) // 2

    maxLeft, minLeft = findMaxMin(arr, low, mid)
    maxRight, minRight = findMaxMin(arr, mid + 1, high)

    max_val = max(maxLeft, maxRight)
    min_val = min(minLeft, minRight)
    return max_val, min_val

n = int(input("Enter the size of the array: "))
arr = list(map(int, input("Enter the array elements: ").split()))

max_val, min_val = findMaxMin(arr, 0, n - 1)

print(f"Maximum element: {max_val}")
print(f"Minimum element: {min_val}")
```

Output:

```
Enter the size of the array: 5
Enter the array elements: 27 81 76 19 03
Maximum element: 81
Minimum element: 3
```

Output Analysis:

Time Complexity: O(n log n)

The algorithm uses a divide-and-conquer approach, breaking the problem into subproblems and recombining the results. The recursive calls contribute to the logarithmic factor, while the work done per level (finding middle element and comparisons) results in the linear factor.

Space Complexity: O(log n)

The recursive nature of the algorithm leads to function call stack overhead. The call stack depth grows logarithmically with the input size, resulting in logarithmic space complexity.

Aim:
Apply 0/1 knapsack problem to find the maximum profit using dynamic approach and analyse its time complexity.

Description:
Given N items where each item has some weight and profit associated with it and also given a bag with capacity W, [i.e., the bag can hold at most W weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

•Initialization: Create a 2D array to store maximum values for different capacities, initializing the first row and column to zeros.
•Dynamic Programming: Iterate over items and capacities.
•For each item: If the item's weight is greater than the current capacity, skip it. Otherwise, choose the maximum of including or excluding the item.
•Output: The value in the bottom-right cell of the array represents the maximum value that can be obtained.

Algorithm :

```
Dynamic-0-1-knapsack (v, w, n, W)
  for w = 0 to W do
    c[0, w] = 0
  for i = 1 to n do
    c[i, 0] = 0
    for w = 1 to W do
      if wi ≤ w then
        if vi + c[i-1, w-wi] then
          c[i, w] = vi + c[i-1, w-wi]
        else c[i, w] = c[i-1, w]
      else
        c[i, w] = c[i-1, w]
```

Code:

```
def knapSack(W, wt, val, n):

    if n == 0 or W == 0:
        return 0
    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)
    else:
        return max(
            val[n-1] + knapSack(
                W-wt[n-1], wt, val, n-1),
            knapSack(W, wt, val, n-1))
capacity = int(input("Enter the capacity of the knapsack: "))
num_items = int(input("Enter the number of items: "))
print("Enter the profits and weights as space-separated arrays:")
profits = list(map(int, input("Profits: ").split()))
weights = list(map(int, input("Weights: ").split()))
print("\nMax Profit: ", knapSack(capacity, weights, profits, num_items))
```

Output:

```
Enter the capacity of the knapsack: 50
Enter the number of items: 3
Enter the profits and weights as space-separated arrays:
Profits: 60 100 120
Weights: 10 20 30

Max Profit:  220
```

Output Analysis:

Time Complexity: O(n×W)

n: number of items
W: capacity of the knapsack
The time complexity arises from the nested loops in the dynamic programming solution, where we iterate through each item and each possible weight capacity of the knapsack.

Space Complexity: O(n×W)

n: number of items
W: capacity of the knapsack
We use a 2D array of size (n+1) x (W+1) to store the solutions to subproblems. Therefore, the space complexity is proportional to the product of the number of items and the capacity of the knapsack.

Aim:
Apply Huffman coding to find out the code of each word, percentage of maximum profit and analyze its time complexity.

Description:
Huffman coding is a lossless data compression algorithm. It assigns variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

1. Frequency Analysis: Determine the frequency of each symbol in the input data.
2. Build Huffman Tree: Create a binary tree where more frequent symbols have shorter binary codes.
3. Generate Huffman Codes: Assign binary codes to each symbol based on their position in the Huffman tree.
4. Encode Data: Replace symbols in the input data with the corresponding Huffman codes.
5. Output Encoding: Include necessary information for decoding, such as the Huffman tree structure.
6. Decoding: Reconstruct the original data using the Huffman tree.
7. Compression Ratio: Compare the size of the original data with the compressed data to measure efficiency.

Algorithm :
```
Step 1: create a priority queue Q consisting of each unique character.
Step 2: sort then in ascending order of their frequencies.
Step 3: for all the unique characters:
Step 4: create a newNode
Step 5: extract minimum value from Q and assign it to leftChild of newNode
Step 6: extract minimum value from Q and assign it to rightChild of newNode
Step 7: calculate the sum of these two minimum values and assign it to the value of   newNode
Step 8: insert this newNode into the tree
Step 9: return rootNode
```

```
Algorithm Huffman (c)
{
   n= |c|
 Q = c
 for i<-1 to n-1
 do
 {
      temp <- get node ()
      left (temp] Get_min (Q) right [temp] Get Min (Q)
      a = left [templ b = right [temp]
      F [temp]<- f[a] + [b]
```

```
    insert (Q, temp)
 }
return Get_min (0)
}
```

Code:

```
from collections import Counter, deque

class Node:
      def __init__(self, char=None, freq=None):
      self.char = char
      self.freq = freq
      self.left = None
      self.right = None

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

def build_huffman_tree(text):
      freq_map = Counter(text)
      nodes = deque([Node(char, freq) for char, freq in freq_map.items()])

      while len(nodes) > 1:
          left = nodes.popleft()
          right = nodes.popleft()
          root = Node(freq=left.freq + right.freq)
          root.left = left
          root.right = right
          nodes.append(root)
      return nodes[0]

def traverse_huffman_tree(root, code, codes):
      if root.char:
          codes[root.char] = code
      else:
          traverse_huffman_tree(root.left, code + "0", codes)
          traverse_huffman_tree(root.right, code + "1", codes)
            return codes

def huffman_encoding(text):
      root = build_huffman_tree(text)
      codes = traverse_huffman_tree(root, "", {})
      encoded = "".join(codes[char] for char in text)
      return encoded, codes
text = input("Enter Text to encode")
encoded, codes = huffman_encoding(text)
print(f"Encoded text: {encoded}")
print(f"Character codes: {codes}")
```

Output:

```
Enter the capacity of the knapsack: 50
Enter the number of items: 3
Enter the profits and weights as space-separated arrays:
Profits: 60 100 120
Weights: 10 20 30

Max Profit:  220
```

Output Analysis:

Time Complexity:

- The overall time complexity of the Huffman coding algorithm is O(n + k log k), where:
  - n is the length of the input text
  - k is the number of unique characters in the input text

This time complexity is primarily driven by the build_huffman_tree function, which takes O(k log k) time to construct the Huffman tree using a priority queue.

Space Complexity:

- The overall space complexity of the Huffman coding algorithm is O(n + k), where:
  - n is the length of the input text
  - k is the number of unique characters in the input text

AIM:
Apply Matrix chain multiplication to find maximum profit and to analyze the time and space complexity.

Description :
The problem relates to finding the most efficient way to multiply a sequence of matrices by determining the optimal order of parenthesization that minimizes the number of scalar multiplications required. This optimization is important because matrix multiplication is an expensive operation, and the number of computations can vary significantly depending on how the matrices are parenthesized. The given context involves using dynamic programming to build a table that stores the minimum costs for multiplying matrix chains of different lengths and start indices. By iterating through the table and filling in the values based on optimal substructure properties, the algorithm can determine the most efficient parenthesization that yields the maximum profit (minimum number of multiplications). Additionally, the time and space complexity analysis is requested to understand the algorithm's efficiency in terms of running time and memory usage as the input scales.

Algorithm :

```
function MatrixChainOrder(p, n):
        // Create a 2D table to store the minimum number of multiplications
        // required to multiply matrices from i to j.
        m = new 2D array of size [n][n]

        // Fill the diagonal entries with 0
        for i = 1 to n:
        m[i][i] = 0

        // Fill the table in bottom-up manner
        for L = 2 to n:   // Chain Length
        for i = 1 to n-L+1:
        j = i+L-1
        m[i][j] = INFINITY
        for k = i to j-1:
        q = m[i][k] + m[k+1][j] + p[i-1]*p[k]*p[j]
        if q < m[i][j]:
                m[i][j] = q

        // The minimum number of multiplications required
        // is the entry at m[1][n]
        return m[1][n]

// Input: Array p[] representing the dimensions of the matrices
//      p[i] represents the number of rows in i'th matrix
//      p[i+1] represents the number of columns in i'th matrix
// n = number of matrices
```

Code :

```
def matrix_chain_order(p):
    n = len(p) - 1
    m = [[0] * n for _ in range(n)]
    s = [[0] * n for _ in range(n)]

    for l in range(2, n + 1):
        for i in range(n - l + 1):
            j = i + l - 1
            m[i][j] = float('inf')
            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
    return m, s

def print_optimal_parens(s, i, j):
    if i == j:
        print("A" + str(i), end="")
    else:
        print("(", end="")
        print_optimal_parens(s, i, s[i][j])
        print_optimal_parens(s, s[i][j]+1, j)
        print(")", end="")

# Get matrix dimensions from user input
n = int(input("Enter the number of matrices: "))
matrix_dimensions = []
for i in range(n + 1):
    dim = int(input(f"Enter dimension {i + 1}: "))
    matrix_dimensions.append(dim)

print("Matrix dimensions:", matrix_dimensions)
m, s = matrix_chain_order(matrix_dimensions)
print("Minimum number of scalar multiplications:", m[0][n - 1])
print("Optimal parenthesization:", end="")
print_optimal_parens(s, 0, n - 1)
```

Output:

```
Enter the number of matrices: 4
Enter dimension 1: 5
Enter dimension 2: 4
Enter dimension 3: 6
Enter dimension 4: 2
Enter dimension 5: 7
Matrix dimensions: [5, 4, 6, 2, 7]

Minimum number of scalar multiplications: 158
Optimal parenthesization:((A0(A1A2))A3)
```

Output Analysis :

Time Complexity:

- The algorithm has a time complexity of O(n^3), where n is the number of matrices in the chain.
- This cubic time complexity is due to the nested loop structure used to fill the 2D table for memoization.
- The outermost loop iterates over the chain length, and the inner two loops iterate over the starting and ending indices of the subchains.

Space Complexity:

- The algorithm has a space complexity of O(n^2), where n is the number of matrices in the chain.
- This quadratic space complexity is due to the use of a 2D table of size n x n to store the minimum costs for multiplying sub chains of matrices.
- Apart from the table, the algorithm uses a constant amount of extra space for variables and temporary calculations.

Aim :
Apply travelling salesman problem to find maximum profit and analyze the time and space complexity.

Description :
The Travelling Salesman Problem (TSP) involves finding the shortest possible route that visits a set of cities and returns to the starting point. In the context of maximizing profit, each city represents a potential business opportunity, and the distances between them are the associated travel costs. The input is a weighted graph with cities as nodes and distances as edge weights. Approaches like brute force for small instances, dynamic programming, branch and bound, and approximation algorithms like nearest neighbor can be employed. The desired output is the optimal sequence of cities forming the most profitable tour, while minimizing the total travel cost. Implementing an efficient solution requires careful consideration of data structures and algorithms to handle larger problem instances.

Algorithm :

```
function tsp(graph, start):
    n = number of cities
    memo = 2^n x n matrix filled with infinity

    function dfs(node, visited):
        if visited == all cities visited:
            return distance from node to start

        if (node, visited) in memo:
            return memo[(node, visited)]

        min_distance = infinity
        for next_node in unvisited cities:
            distance = graph[node][next_node] + dfs(next_node, visited | (1 << next_node))
            min_distance = min(min_distance, distance)

        memo[(node, visited)] = min_distance
        return min_distance

    return dfs(start, 1 << start)
```

Code :
```
def tsp(graph, start):
    n = len(graph)
    max_distance = float('inf')
    memo = {}
    shortest_path = []

    def dfs(node, visited, path):
        if visited == (1 << n) - 1:
            path.append(start)
```

```
                return graph[node][start], path.copy()

            if (node, visited) in memo:
                return memo[(node, visited)]

            min_distance = max_distance
            optimal_path = []

            for next_node in range(n):
                if not visited & (1 << next_node):
                    distance, temp_path = dfs(next_node, visited | (1 <<
next_node), path + [next_node])
                    distance += graph[node][next_node]

                    if distance < min_distance:
                        min_distance = distance
                        optimal_path = temp_path

            memo[(node, visited)] = (min_distance, optimal_path)
            return min_distance, optimal_path

    distance, path = dfs(start, 1 << start, [start])

    return distance, path

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

print("Enter the distances between cities (use space-separated
values):")
graph = []
for _ in range(n):
    row = list(map(int, input().split()))
    graph.append(row)

start_node = int(input("Enter the start city (0-indexed): "))

print("\nDistances between cities:")
for row in graph:
    print(row)

distance, path = tsp(graph, start_node)

print("\nMinimum distance for TSP:", distance)
print("Optimal path:", path)
```

Output:

```
Enter the number of cities: 4
Enter the distances between cities (use space-separated values):
0 10 15 20
10 0 35 25
15 35 0 30
20 25 30 0
Enter the start city (0-indexed): 1

Distances between cities:
[0, 10, 15, 20]
[10, 0, 35, 25]
[15, 35, 0, 30]
[20, 25, 30, 0]

Minimum distance for TSP: 80
Optimal path: [1, 0, 2, 3, 1]
```

Analysis:

Time Complexity:

O(n^2×2^n)

n: Number of cities

We need to fill an n×2^n memoization table, and for each entry, we consider n possible previous cities to compute the optimal distance.

Space Complexity:

O(n×2^n)

n: Number of cities

We need to store an n×2^n memoization table.