Name: Akash Pingali Section: CSE-2 Roll number: 160121733091

EXPERIMENT-1 :

AIM: To Find Maximum value for the items, 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: The given problem is a fractional knapsack problem, where the package of certain capacity is filled in order to maximize profit. Here Greedy approach is used. The basic idea of the greedy approach is to calculate the ratio profit/weight for each item and sort the item based on this ratio using any sorting algorithms such as quick sort or merge sort. Then take the item with the highest ratio and add them as much as we can (can be the whole element or a fraction of it).This will always give the maximum profit because, in each step it adds an element such that this is the maximum possible profit for that much weight.

ALGORITHM :

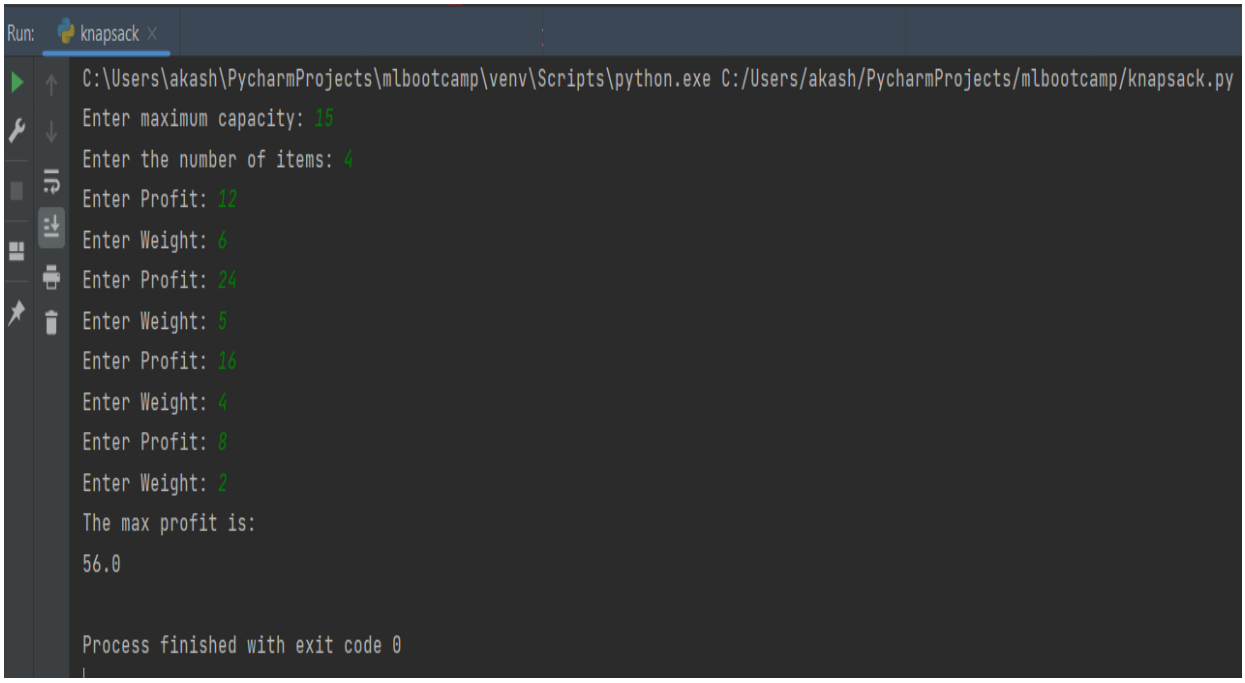
// p[1:n] and w[1:n] consists of profits and weights respectively. Here n is the total number of objects
// C→ maximum capacity of the bag
// r[1:n] consists of the ratios p[1:n]/w[1:n]
1: quicksort(r,p,w)--> sort the ratios in descending order and simultaneously sort p[1:n] and w[1:n]
2: knapsack(p,w)
  2.1: for i:1 to len(profits)
    2.11: if C > weight[i]
        → Do profit= profit+p[i]
        →C=C-weight[i]
    2.12 : else if C > 0
        →Do profit= profit+p[i]*(C/weight[i])
        →C=C-weight[i]
    2.13 : else:
        →break
2: Print maximum profit

CODE :

```
class item:
   def __init__(self,profit,weight):
      self.profit=profit
      self.weight=weight
      self.vw=profit/weight
def fractionalKnapsack(W,sack):
   sack.sort(key=lambda x:x.vw,reverse=True)
   final_profit=0.0
```

```
    for item in sack:
        if item.weight<=W:
            W=W-item.weight
            final_profit=final_profit+item.profit
        else:
            final_profit=final_profit+(item.profit/item.weight)*W
            break
    return final_profit
if __name__ == "__main__":
    W = int(input("Enter maximum capacity: "))
    n = int(input("Enter the number of items: "))

    sack=[]
    for i in range(0,n):
        p=int(input("Enter Profit: "))
        w=int(input("Enter Weight: "))
        sack.append(item(p,w))
    print("The max profit is: ")
    print(fractionalKnapsack(W,sack))
```

OUTPUT:



CONCLUSION AND ANALYSIS:

To take input and arrange the sack in v/w ratio takes a for loop and is of O(n). Sorting the v/w in the descending order is of O(nlogn). And to calculate the maximum profit (Greedy algorithm) another for loop is used which is traversed till n (n- number of objects) so the time complexity is O(n). Checking for remaining item and adding their fractional value takes O(1). Therefore, the overall time complexity for a fractional knapsack problem is O(nlogn).

EXPERIMENT-2 :

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: The given problem can be optimized using job-sequencing. The job-sequencing algorithm is a greedy algorithm, where each project is associated with profits and deadlines. If there are many possible combinations, greedy approach is used. Given an array of jobs where every job has a deadline and associated profit. Initially. the job with maximum profit first is chosen, by sorting the jobs in decreasing order of their profit. Each profit is scheduled on i th slot, where i<= deadline of i th and is free. As they are organised according to greater profit, overall profit is maximized.

ALGORITHM :

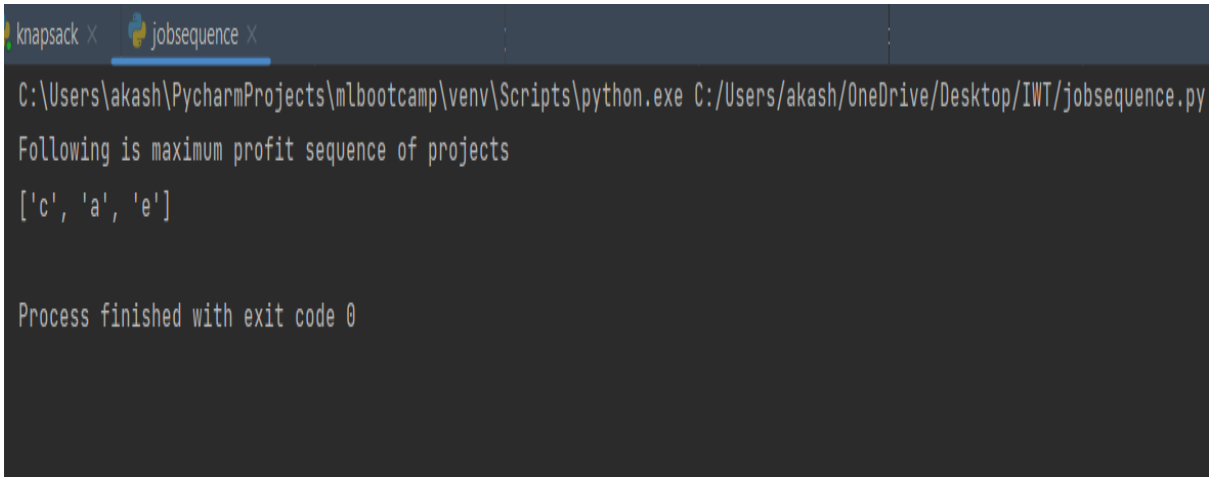
// p[1:n] and d[1:n] are the profits and deadlines respectively for the jobs
1: Sort profits in decreasing order and sort deadlines simultaneously using quick/mergesort.
2: job_sequencing(p,d)
  let arr be an array with 0 entries and with the length of max of deadline
  2.2: for i →0 to len(p)
    2.21: for j→ 0 to len(p)-1-i
      2.211: if
          arr[j][2]<arr[j+1][2]:
          arr[j],arr[j+1]=profit[j+1],profit[j]
  2.3: for i →0 to len(p)
    2.31: for j→ 0 to min(d - 1, profit[i][1] - 1), -1, -1
      2.311: if res[j] is False:
          res[j] = True
          job[j] = arr[i][0]
          break
3: print the required sequence

CODE :

```
def job(profit,dur):
   n=len(profit)
   for i in range(n):
      for j in range(n-1-i):
         if profit[j][2]<profit[j+1][2]:
            profit[j],profit[j+1]=profit[j+1],profit[j]
      res = [False] * dur
```

```
        job = ['-1'] * dur
    for i in range(len(profit)):
        for j in range(min(dur - 1, profit[i][1] - 1), -1, -1):

            if res[j] is False:
                res[j] = True
                job[j] = profit[i][0]
                break

    print(job)


if __name__ == '__main__':
    profit = [['a', 2, 63],
              ['b', 1, 24],
              ['c', 2, 49],
              ['d', 1, 25],
              ['e', 3, 12]]
    print("Following is maximum profit sequence of projects")
    job(profit, 3)
```

OUTPUT:



CONCLUSION AND ANALYSIS :

To sort the profits and deadlines sorting algorithm is used (quick/merge) so the time taken is O(nlogn). To calculate the maximum profit two for loops are used so the time taken is O(n^2). For fixing slots, two for loops are used so O(n^2). So, the overall time complexity for job sequencing problem is O(n^2).

EXPERIMENT-3 :

AIM : To find the maximum points scored by choosing certain questions such that the questions can be completed within the deadline. A test has 'N' questions with a heterogeneous distribution of points. The test-taker has a choice as to which questions can be answered. Each question Qi has points Pi and time Ti to answer the question, where 1<=i<=N. The students are asked to answer the possible subsets of problems whose total point values add up to a maximum score within the time limit 'T'. Determine which subset of questions gives the student the highest possible score.

DESCRIPTION :

This is a 0/1 knapsack problem in which the weights of the knapsack(bag) is the maximum time limit of the test, profits of the items represent the points gained and the weights represent the time required to answer a specific question. Dynamic programming is used to identify the optimal solution evaluating all possible solutions. Here, tabulation is the best approach, where a table is created that stores the points while adding certain item to knapsack. The resultant set is a 0/1 set which tells if the item is included in sack or not.

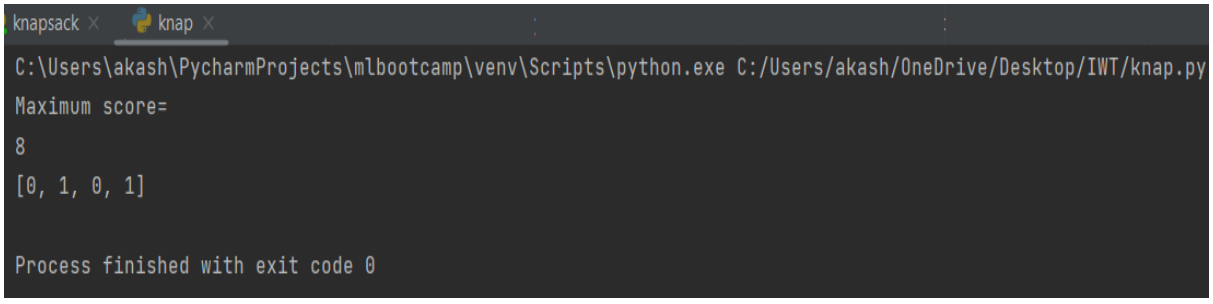
ALGORITHM:

//wt[1:n] and p[1:n] are the time limits and points of n questions
//C is the maximum time limit
// K[C+1][len(p)+1]--> 2D array with number of rows=C+1 and columns=len(p)+1
1: knapsack(wt,p,C)
  1.1: for i →0 to len(p)+1
    1.11: for j →0 to C+1
      1.111 : if i==0 or j==0:
        →Do K[i][j]=0
      1.112 : elif check wt[i-1]<=j:
        →Do K[i][j]=max(K[i-1][j],K[i-1][j-wt[i-1]]+p[i-1])
      1.113 : else:
        →Do K[i][j]=K[i-1][j]
2: Print the maximum profit→ K[i][j]
3: To print the subset of questions let l be the list of len(p) length and has all 0 entries
  3.1:for k →len(p) to 0:
    3.11: if K[k][C]!=K[k-1][C]
      → Do l[k-1]=1, maxprofit-=p[k-1],C-=wt[k-1]
4: Print l which is the subset of questions which gives students the highest possible score.

CODE :

```
def knapSack(W, wt, val, n):
  K = [[0 for x in range(W + 1)] for x in range(n + 1)]

  for i in range(n + 1):
    for w in range(W + 1):
      if i == 0 or w == 0:
        K[i][w] = 0
      elif wt[i-1] <= w:
        K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
      else:
        K[i][w] = K[i-1][w]
  set=[0]*n
  max_val=K[n][W]
  print("Maximum score= ")
  print(K[n][W])
  for i in range(n,0,-1):
    if K[i][W]!=K[i-1][W]:
      set[i-1]=1
      W-=wt[i-1]
      max_val-=val[i-1]

  return set

val = [1,2,5,6]
wt = [2,3,4,5]
W = 8
n = len(val)
print(knapSack(W, wt, val, n))
```

OUTPUT :



CONCLUSION AND ANALYSIS :

To find the maximum profit 2 for loops are used so the time complexity is O(n*W). A for loop is used for arranging subset O(n). Therefore, effective time complexity is O(n*W)

EXPERIMENT-4 :

AIM : An X-ray telescope (XRT) is a telescope that is designed to observe remote objects in the Xray spectrum. In order to get above the Earth's atmosphere, which is opaque to X-rays, X-ray telescopes must be mounted on high altitude rockets, balloons or artificial satellites. Planets, stars and galaxies and the observations are to be made with telescope. Here the process of rotating equipment into position to observe the objects is called slewing. Slewing is a complicated and time-consuming procedure handled by computer driven motors. The problem is to find the tour of the telescope that moves from one object to other by observing each object exactly once with a minimum total slewing time

DESCRIPTION :

This is a travelling salesman problem in which the salesman must travel to every city exactly once such that he has to return to the initial point with minimum distance. In this algorithm, we take a subset N of the required objects that need to be visited, the distance among the objects dist, and starting object i as inputs. All the possibilities are checked in dynamic programming approach and best possible optimal solution is chosen. Each object is identified by a unique id which we say like 1,2,3,4,..n. Here we use a dynamic approach to calculate the minimum distance. Using recursive calls, we calculate the minimum distance for each subset of the original problem.

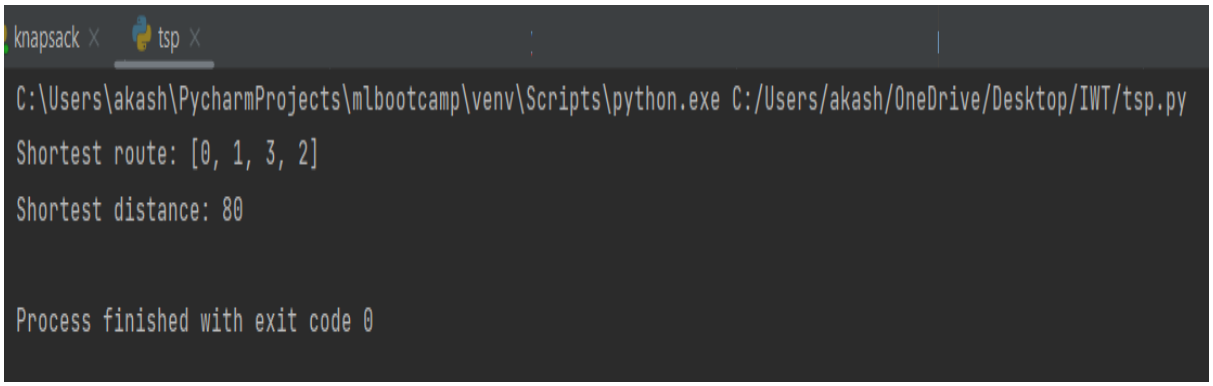
ALGORITHM :

// c -> 2d array representing the distances between every pair of vertices
// s -> unvisited vertices
1. Start
2. TSP(i, s) // initially i = 1, s = [2,3,4]
    2.1. if s = NULL
        2.1.1. return c[i][1], 'i'
    2.2. else
        2.2.1. n ⇐ length of s
        2.2.2. ways ⇐ array of size n
        2.2.3. for j ⇐ 1 : n
            2.2.3.1. temp = s
            2.2.3.2. remove j from temp
            2.2.3.3. dis, p ⇐ TSP(j, temp)
            2.2.3.4. ways[j] ⇐ [c[i][j] + dis, p]
        2.2.4. min_path ⇐ min(ways) based on ways[0]
        2.2.5. path ⇐ 'i' + min_path[1]
        2.2.6. return min_path[0], path
3. Print distance and path
4. Stop

CODE :

```
def total_distance(route, distances):
    dist = 0
    for i in range(len(route)-1):
        dist += distances[route[i]][route[i+1]]
    dist += distances[route[-1]][route[0]]
    return dist

def tsp(distances):
    cities = list(range(len(distances)))
    shortest_route = None
    shortest_distance = float("inf")
    for route in permutations(cities):
        dist = total_distance(route, distances)
        if dist < shortest_distance:
            shortest_distance = dist
            shortest_route = route
    return shortest_route, shortest_distance


def permutations(lst):
    if len(lst) == 1:
        return [lst]
    else:
        result = []
        for i in range(len(lst)):
            rest = lst[:i] + lst[i+1:]
            rest_permutations = permutations(rest)
        for perm in rest_permutations:
                result.append([lst[i]] + perm)
        return result

distances = [
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0]
]

shortest_route, shortest_distance = tsp(distances)
print("Shortest route:", shortest_route)
print("Shortest distance:", shortest_distance)
```

OUTPUT:



CONCLUSION AND ANALYSIS :

This algorithm recursively finds the shortest part starting from each neighbour and returns the minimum of those. Since we are starting from vertex 1 there are 2^n possible subgraphs to cover all vertices and reach the beginning, so, this function will be called on at most 1*2^n times and each call can be recursively called n times. Hence the total time complexity is O(n*2^n).

EXPERIMEN- 5

TITLE: - Huffman Code

AIM: The aim of Huffman coding is to compress data by assigning variable-length codes to characters based on their frequencies, with more frequent characters having shorter codes and less frequent characters having longer codes.

DESCRIPTION:

Huffman coding is a lossless data compression algorithm widely used in applications like file compression and data transmission. The algorithm builds a binary tree called the Huffman tree, where each leaf node represents a character and its associated frequency. The path from the root to a leaf node represents the binary code for that character.

ALGORITHM:

1. Create a frequency table that counts the occurrences of each character in the input data.
2. Create a priority queue (min-heap) to hold the frequency table entries. Each entry consists of a character and its frequency.
3. For each character-frequency pair in the frequency table, create a leaf node and insert it into the priority queue.
4. While the priority queue has more than one node:
    - Remove the two nodes with the lowest frequencies from the priority queue.
    - Create a new internal node with a frequency equal to the sum of the frequencies of the two nodes.
    - Set the two removed nodes as the left and right children of the new internal node.
    - Insert the new internal node back into the priority queue.
5. The remaining node in the priority queue is the root of the Huffman tree.
6. Traverse the Huffman tree to assign binary codes to each character:
    - Start from the root and perform a depth-first search.
    - When visiting a left child, append 0 to the current code. When visiting a right child, append 1 to the current code.
    - Store the binary codes for each character in a lookup table.
7. Encode the input data using the generated Huffman codes by replacing each character with its corresponding binary code.
8. Store the Huffman codes and the encoded data for later decoding.

CODE:

```
import heapq
from collections import Counter


class HuffmanNode:
```

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

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

def build_frequency_table(data):
    frequency_table = Counter(data)
    return frequency_table

def build_huffman_tree(frequency_table):
    pq = []
    for char, freq in frequency_table.items():
        node = HuffmanNode(char, freq)
        heapq.heappush(pq, node)

    while len(pq) > 1:
        node1 = heapq.heappop(pq)
        node2 = heapq.heappop(pq)
        internal_node = HuffmanNode(None, node1.freq + node2.freq)
        internal_node.left = node1
        internal_node.right = node2
        heapq.heappush(pq, internal_node)

    return pq[0]

def build_huffman_codes(root):
    codes = {}
    current_code = ""

    def traverse(node, code):
        if node.char:
            codes[node.char] = code
            return

        traverse(node.left, code + "0")
        traverse(node.right, code + "1")

    traverse(root, current_code)
    return codes
```

```
def encode_data(data, codes):
    encoded_data = ""
    for char in data:
        encoded_data += codes[char]
    return encoded_data

# Example usage
data = "Design & Algorithm"
frequency_table = build_frequency_table(data)
huffman_tree = build_huffman_tree(frequency_table)
huffman_codes = build_huffman_codes(huffman_tree)
encoded_data = encode_data(data, huffman_codes)

print("Original data:", data)
print("Encoded data:", encoded_data)
```

OUTPUT:

CONCLUSION AND ANALYSIS:

The time and space complexity of the Huffman coding algorithm can be analysed as follows:

Time Complexity:

1. Building the frequency table: Constructing the frequency table from the input data requires iterating over all the characters in the data, which takes O(n) time, where n is the length of the data.
2. Building the Huffman tree: Constructing the Huffman tree involves repeatedly extracting the minimum elements from the priority queue, which takes O(n log n) time. This is because the priority queue can have at most n nodes, and each extraction operation takes O(log n) time.
3. Building the Huffman codes: Traversing the Huffman tree to assign codes to each character takes O(n) time, as each character is visited exactly once.

Therefore, the overall time complexity of the Huffman coding algorithm is O(n log n), where n is the length of the input data.

Space Complexity: The space complexity of the Huffman coding algorithm can be analyzed as follows:

1. Frequency table: The frequency table requires space to store the frequency of each character in the input data. In the worst case, when all characters are unique, the frequency table would require O(n) space.
2. Priority queue: The priority queue requires space to store the nodes of the Huffman tree. In the worst case, when all characters are unique, the priority queue would require O(n) space.
3. Huffman tree: The Huffman tree requires space to store the internal nodes and leaf nodes. In the worst case, when all characters are unique, the Huffman tree would require O(n) space.
4. Huffman codes: The Huffman codes require space to store the binary codes for each character. In the worst case, when all characters are unique, the Huffman codes would require O(n) space.

Therefore, the overall space complexity of the Huffman coding algorithm is O(n).

EXPERIMENT- 6

TITLE: - Matrix Chain Multiplication

AIM: The aim of the Matrix Chain Multiplication problem is to find the most efficient way to multiply a chain of matrices, minimizing the total number of scalar multiplications required.

DESCRIPTION:
Given a chain of matrices of various dimensions, the Matrix Chain Multiplication problem aims to determine the order in which to perform the matrix multiplications to minimize the total number of scalar multiplications required. The dimensions of the matrices determine the number of rows and columns of each matrix in the chain.

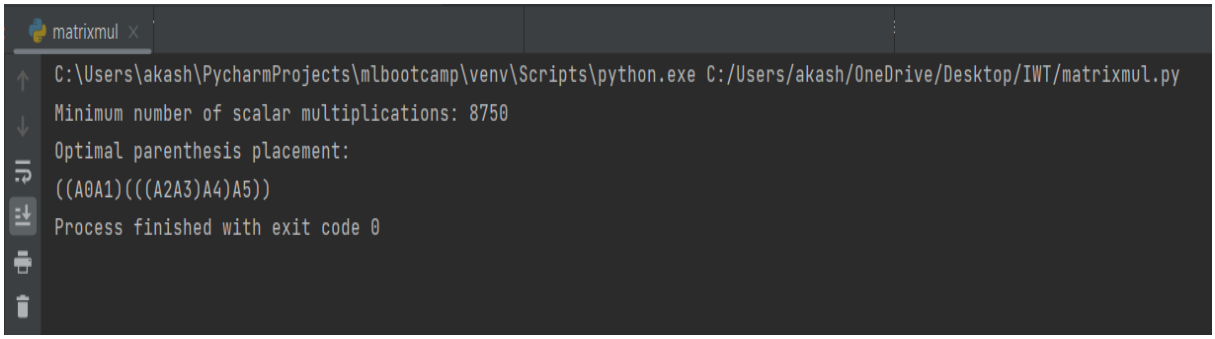
ALGORITHM:

1. Create a dynamic programming table **dp** of size n x n, where n is the number of matrices in the chain.
2. Initialize the diagonal elements of **dp** with 0 since a single matrix requires no multiplication.
3. Iterate over the chain length **l** from 2 to n, representing the length of the subchain being considered.
4. For each subchain length **l**, iterate over the starting index **i** from 1 to n-l+1.
5. Calculate the ending index **j** as **i + l - 1**.
6. Initialize **dp[i][j]** with infinity or a very large value.
7. Iterate over the possible partition index **k** from **i** to **j-1**.
8. Calculate the number of scalar multiplications needed for multiplying the matrices in the left and right subchains.
    - Left subchain: **dp[i][k]**
    - Right subchain: **dp[k+1][j]**
    - Multiplication of the resulting matrices: **dim[i-1] * dim[k] * dim[j]**
    - Total scalar multiplications: **dp[i][k] + dp[k+1][j] + dim[i-1] * dim[k] * dim[j]**
9. Update **dp[i][j]** with the minimum value among all possible **k**.
10. After completing all iterations, the value of **dp[1][n]** represents the minimum number of scalar multiplications required to multiply the entire chain of matrices.
11. To reconstruct the optimal multiplication order, traverse the **dp** table and choose the partition index **k** that achieves the minimum value at each step.

CODE:

```
import sys

def matrix_chain_multiplication(dimensions):
    n = len(dimensions) - 1
    dp = [[0] * n for _ in range(n)]
    parenthesis = [[0] * n for _ in range(n)]

    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1
            dp[i][j] = sys.maxsize

            for k in range(i, j):
                cost = dp[i][k] + dp[k + 1][j] + dimensions[i] * dimensions[k + 1] * dimensions[j +
1]
                if cost < dp[i][j]:
                    dp[i][j] = cost
                    parenthesis[i][j] = k

    return dp, parenthesis

def print_optimal_parenthesis(parenthesis, i, j):
    if i == j:
        print(f"A{i}", end="")
    else:
        print("(", end="")
        print_optimal_parenthesis(parenthesis, i, parenthesis[i][j])
        print_optimal_parenthesis(parenthesis, parenthesis[i][j] + 1, j)
        print(")", end="")

# Example usage
dimensions = [10,20,5,15,50,10,15]
dp, parenthesis = matrix_chain_multiplication(dimensions)

print("Minimum number of scalar multiplications:", dp[0][len(dimensions) - 2])
print("Optimal parenthesis placement:")
print_optimal_parenthesis(parenthesis, 0, len(dimensions) - 2)
```

OUTPUT:



CONCLUSION AND ANALYSIS:

Time Complexity: The algorithm uses dynamic programming to solve the problem, and it involves nested loops for filling the **dp** matrix. The time complexity can be calculated as O(n^3), where n is the number of matrices in the chain. This is because there are two nested loops for iterating over the subchain length and the starting index, and an additional loop for iterating over the possible partition index. Each loop has a maximum of n iterations.

Space Complexity: The space complexity of the algorithm is O(n^2), where n is the number of matrices in the chain. This is because the algorithm uses two matrices, **dp** and **parenthesis**, each of size n x n, to store the intermediate results and optimal parenthesis placement. Thus, the space required is proportional to the square of the number of matrices in the chain.

EXPERIMENT-7

TITLE: - Optimal Binary Search Tree

AIM: The aim of Optimal Binary Search Tree (OBST) is to construct a binary search tree that minimizes the average search time for a given set of keys and their respective frequencies.

DESCRIPTION:

Optimal Binary Search Tree is a variation of the binary search tree where the keys are arranged in such a way that the expected search time is minimized. In an optimal binary search tree, frequently accessed keys are placed closer to the root, reducing the average search time.

ALGORITHM:

1. Create a dynamic programming table **dp** of size (n+1) x (n+1), where n is the number of keys.
2. Initialize the diagonal elements of **dp** with the frequency values, representing single-key subtrees.
3. Iterate over the chain length **l** from 2 to n, representing the length of the subchain being considered.
4. For each subchain length **l**, iterate over the starting index **i** from 1 to n-l+1.
5. Calculate the ending index **j** as **i + l - 1**.
6. Initialize **dp[i][j]** with infinity or a very large value.
7. Iterate over the possible root index **r** from **i** to **j**.
8. Calculate the sum of frequencies from **i** to **j**.
9. Calculate the cost of the current root **r** as:
    - Left subtree cost: **dp[i][r-1]**
    - Right subtree cost: **dp[r+1][j]**
    - Sum of frequencies: **sum(freq, i, j)**
    - Total cost: **left subtree cost + right subtree cost + sum of frequencies**
10. If the current cost is less than **dp[i][j]**, update **dp[i][j]** with the new cost.
11. After completing all iterations, the value of **dp[1][n]** represents the minimum cost of the optimal binary search tree.
12. To reconstruct the optimal binary search tree, use the **dp** table and root indices **i** and **j**:

- Find the root index **r** with the minimum cost within the range **i** to **j**.
- Create a node with the key at index **r**.
- Recursively construct the left subtree using the range **i** to **r-1** and the right subtree using the range **r+1** to **j**.

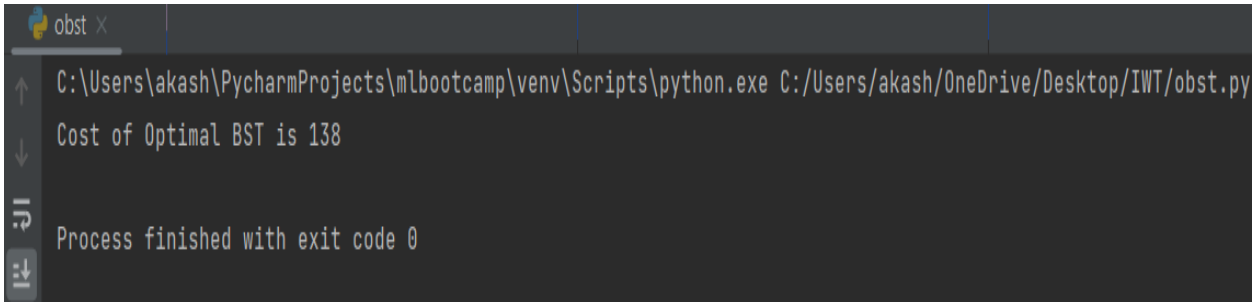
13. Return the root node of the constructed optimal binary search tree.

CODE:

```
def optimal_cost(keys, frequency):
```

```
    n = len(keys)
    dp = [[0] * n for _ in range(n)]

    # Initialize the diagonal with frequencies
    for i in range(n):
        dp[i][i] = frequency[i]

    # Compute optimal costs
    for l in range(2, n + 1):
        for i in range(n - l + 1):
            j = i + l - 1
            dp[i][j] = float('inf')
            freq_sum = sum(frequency[i:j + 1])

            for r in range(i, j + 1):
                cost = freq_sum + (dp[i][r - 1] if r > i else 0) + (dp[r + 1][j] if r < j else 0)
                if cost < dp[i][j]:
                    dp[i][j] = cost

    return dp[0][n - 1]


def optimal_search_tree(keys, frequency):
    return optimal_cost(keys, frequency)


keys = [10, 20, 30]
frequency = [24, 18, 36]
print("Cost of Optimal BST is", optimal_search_tree(keys, frequency))
```

OUTPUT:



CONCLUSION AND ANALYSIS:

Time Complexity: The OBST algorithm uses dynamic programming to solve the problem. It involves nested loops to fill in the dynamic programming table. The time complexity can be

calculated as O(n^3), where n is the number of keys. This is because there are three nested loops: one for the chain length, one for the starting index, and one for the root index. Each loop iterates up to n times.

Space Complexity: The space complexity of the OBST algorithm is O(n^2), where n is the number of keys. This is because the algorithm requires a dynamic programming table of size (n+1) x (n+1) to store the intermediate results. Additionally, there is additional space required for the frequency array and other constant-sized variables.

EXPERIMENT-8

TITLE: - Longest Common Subsequence

AIM: The aim of the Longest Common Subsequence (LCS) problem is to find the longest subsequence that is common to two given sequences

DESCRIPTION:

Given two sequences, the LCS problem involves finding the longest subsequence that appears in both sequences. A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

ALGORITHM:

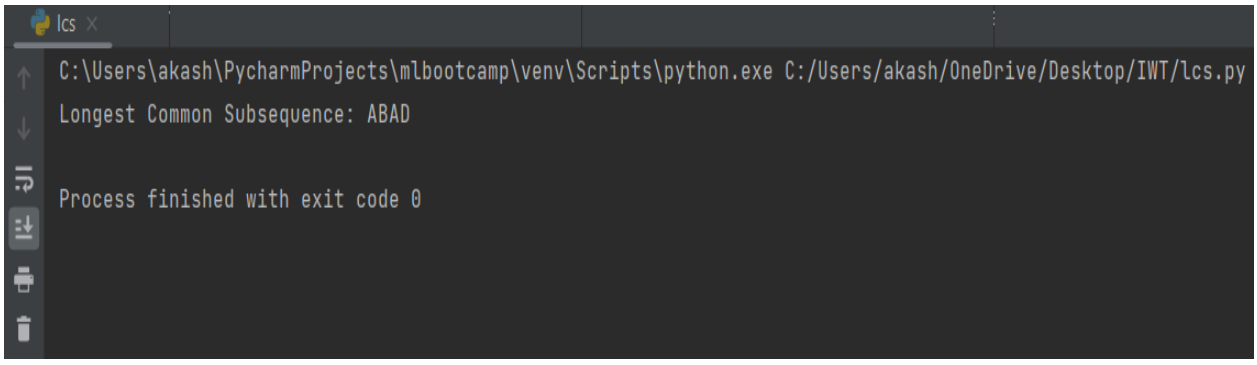
1. Create a dynamic programming table **dp** of size (m+1) x (n+1), where m and n are the lengths of the two sequences.
2. Initialize the first row and the first column of the **dp** table as 0. These represent the base cases when one of the sequences is empty.
3. Iterate over the rows and columns of the **dp** table starting from the second row and the second column.
4. If the characters at the corresponding positions in the two sequences are equal, set **dp[i][j] = dp[i-1][j-1] + 1**. This means that the current characters contribute to the length of the LCS.
5. If the characters are not equal, set **dp[i][j] = max(dp[i-1][j], dp[i][j-1])**. This means that we consider the LCS length obtained by excluding either of the characters.
6. After completing the iteration, **dp[m][n]** will contain the length of the LCS.
7. To reconstruct the LCS itself, start from **dp[m][n]** and backtrack using the following rules:
    - If the characters at the corresponding positions are equal, add the character to the LCS and move diagonally up-left (**i-1**, **j-1**).
    - If **dp[i-1][j] > dp[i][j-1]**, move up (**i-1**, **j**).
    - If **dp[i-1][j] <= dp[i][j-1]**, move left (**i**, **j-1**).
    - Repeat until reaching the first row or the first column.
8. Reverse the obtained LCS to get the correct order of characters.

CODE:

```
def longest_common_subsequence(seq1, seq2):
    m = len(seq1)
    n = len(seq2)

    # Initialize the dp table
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Fill in the dp table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if seq1[i - 1] == seq2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
```

```
            else:
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    # Reconstruct the LCS
    lcs = []
    i = m
    j = n
    while i > 0 and j > 0:
        if seq1[i - 1] == seq2[j - 1]:
            lcs.append(seq1[i - 1])
            i -= 1
            j -= 1
        elif dp[i - 1][j] > dp[i][j - 1]:
            i -= 1
        else:
            j -= 1

    lcs.reverse()
    return lcs

seq1 = "ABAZDC"
seq2 = "BACBAD"

lcs = longest_common_subsequence(seq1, seq2)
print("Longest Common Subsequence:", "".join(lcs))
```

OUTPUT:



CONCLUSION AND ANALYSIS:

Time Complexity: The LCS algorithm uses a dynamic programming approach to fill in a 2D table of size (m+1) x (n+1), where m and n are the lengths of the two input sequences. The filling of the table requires iterating through all the elements in the table once, which takes O(m

* n) time. Therefore, the time complexity of the LCS algorithm is O(m * n), where m and n are the lengths of the input sequences.

Space Complexity: The space complexity of the LCS algorithm is determined by the size of the 2D table used to store the lengths of the common subsequences. The table has dimensions (m+1) x (n+1), where m and n are the lengths of the input sequences. Hence, the space complexity is O(m * n).

EXPERIMENT-09

TITLE: N-Queen

AIM: N-Queen is the problem of placing 'N' chess queens on an N×N chessboard. Design a solution for this problem so that no two queens attack each other.
Note: A queen can attack when an opponent is on the same row, column or diagonal

DESCRIPTION:
The N-Queens problem is a classic puzzle that involves placing N queens on an N x N chessboard in such a way that no two queens threaten each other. In chess, a queen can attack in any direction - horizontally, vertically, or diagonally. The goal of the N-Queens problem is to find all possible arrangements of the queens on the board that satisfy this constraint. One common approach to solving the N-Queens problem is by using backtracking. Backtracking is a depth-first search algorithm that systematically explores all possible configurations of queens on the board. It places a queen in one column of the first row and then recursively tries to place the remaining queens in the subsequent rows. If at any point a conflict arises, it backtracks and tries a different position. The N-Queens problem is not only a challenging puzzle but also has practical applications in various fields such as chess, computer vision, and optimization.

ALGOTIHM:

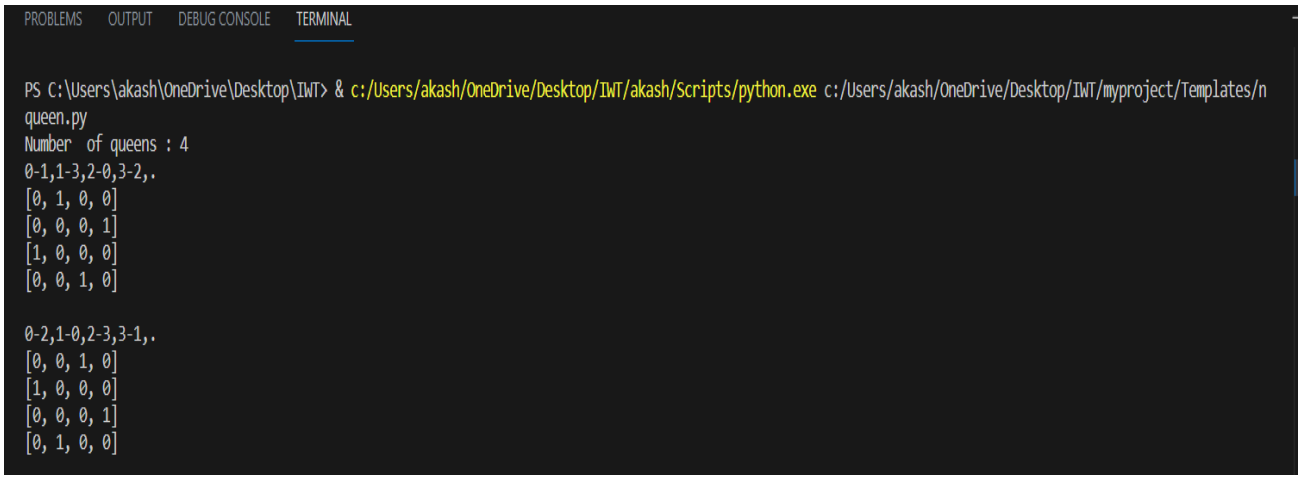
1. Start with an empty chessboard of size N×N.
2. Initialize a variable row to 0, representing the current row being processed.
3. If row is equal to N, it means all queens have been successfully placed on the board. Print or store this configuration as a valid solution and backtrack to find other solutions.
4. Iterate over each column in the current row (col = 0 to N-1).
5. Check if placing a queen at the current position (row, col) is safe. Use a helper function isQueenSafe(board, row, col) to check if the current position is not under attack by any other queen on the board.
6. In the isQueenSafe function, check if there is any queen in the same column, same row, or diagonals as the current position. If a queen is found, return False.
7. If placing a queen at the current position is safe, mark that position as occupied on the board (board[row][col] = 1) to represent the queen.
8. Recursively call the algorithm for the next row (row+1).
9. If the recursive call returns True (indicating that a valid solution was found in the subsequent rows), return True.
10. If the recursive call returns False, it means there is no valid solution for the remaining rows. Backtrack by undoing the current move (board[row][col] = 0) and try the next column in the current row.
11. If none of the columns in the current row lead to a valid solution, return False.
12. Repeat steps 4-10 until all valid solutions have been found.
13. Print or return all the valid solutions found.

CODE:

```
def print_n_queens(board,ans,row):
    if(row==len(board)):
        print(ans+'.')
        for row in board:
            print(row)
        print ()
        return
    for col in range(len(board)):
        if(board[row][col]==0 and is_queen_safe(board, row, col)==True):
            board[row][col]=1
            print_n_queens(board,ans+str(row)+"-"+str(col)+",",row+1)
            board[row][col]=0

def is_queen_safe(board, row, col):
    # Check upper-left diagonal
    i = row - 1
    j = col - 1
    while i >= 0 and j >= 0:
        if board[i][j] == 1:
            return False
        i -= 1
        j -= 1
    # Check upper row
    i = row - 1
    j = col
    while i >= 0:
        if board[i][j] == 1:
            return False
        i -= 1

    # Check upper-right diagonal
    i = row - 1
    j = col + 1
    while i >= 0 and j < len(board):
        if board[i][j] == 1:
            return False
        i -= 1
        j += 1

    # Check left column
    i = row
    j = col - 1
```

```
    while j >= 0:
        if board[i][j] == 1:
            return False
        j -= 1

    return True

n=int(input("Number  of queens : "))
board=[[0]*n for _ in range(n)]

print_n_queens(board,"",0)
```

OUTPUT:



CONCLUSION AND ANALYSIS:

Time Complexity:
The time complexity of the N-Queens problem using backtracking is typically expressed in terms of the number of recursive function calls and the number of operations performed within each call. In worst case, where all valid configurations/solutions are considered, the total number of recursive calls made by the algorithm is exponential in the size of the chessboard (N). It can be approximated as O(N!). This is because for each row, we have N choices (columns) to place the queen, resulting in N possibilities at each level of recursion. Therefore, the overall time complexity of the N-Queens problem is approximately O(N!).

Space Complexity:
The space complexity of the N-Queens problem primarily depends on the space used to represent the chessboard and the recursive call stack. Space required to store the chessboard is O(N^2) since it is a 2D array of size N×N. The space complexity of the recursive algorithm is O(N). Therefore the overall space complexity is approximately O(N^2).

EXPERIMENT-10:

TITLE: Graph Coloring

AIM: CSE department of CBIT want to generate a timetable for "N" subjects. The following information is given- subject name, subject code and list of subject's code which clashes with this subject. The problem is to identify the list of subjects which can be scheduled on the same timeline such that clashes among them do not exist.

DESCRIPTION:

The Graph Coloring Problem is a classic problem in graph theory and computer science. It involves assigning colors to the vertices of a graph such that no two adjacent vertices share the same color. The goal is to find the minimum number of colors required to color the graph. The problem has various applications in real-world scenarios, such as scheduling, map coloring, register allocation in compilers, and solving Sudoku puzzles.
The Graph Coloring Problem is known to be NP-complete, which means that there is no known polynomial-time algorithm that can solve all instances of the problem. Therefore, various heuristics and approximation algorithms are commonly used to find feasible or near-optimal solutions. The algorithm recursively explores different color assignments for the vertices and backtracks whenever it encounters a conflict. The backtracking algorithm explores the search space of possible colorings until a valid coloring is found or all possibilities are exhausted.

ALGORITHM:

1. Initialize an empty dictionary called clashes to store the subject codes and their corresponding neighbors.
2. Iterate over each subject in the subjects list.
3. Retrieve the subject code and the list of neighbors from the subject tuple.
4. Add the subject code as a key in the clashes dictionary and set its value to the set of neighbors.
5. Initialize an empty dictionary called timetable to store the timetable with colors as keys and subject codes as values.
6. Initialize an empty dictionary called colors to store the assigned color for each subject code.
7. Iterate over each subject in the subjects list again.
8. Retrieve the subject code.
9. Create a set called available_colors by iterating over the neighbors of the subject code and retrieving their assigned colors from the colors dictionary.
10. Assign a color to the subject by finding the smallest positive integer that is not present in the available_colors set.
11. Store the assigned color in the colors dictionary for the subject code.

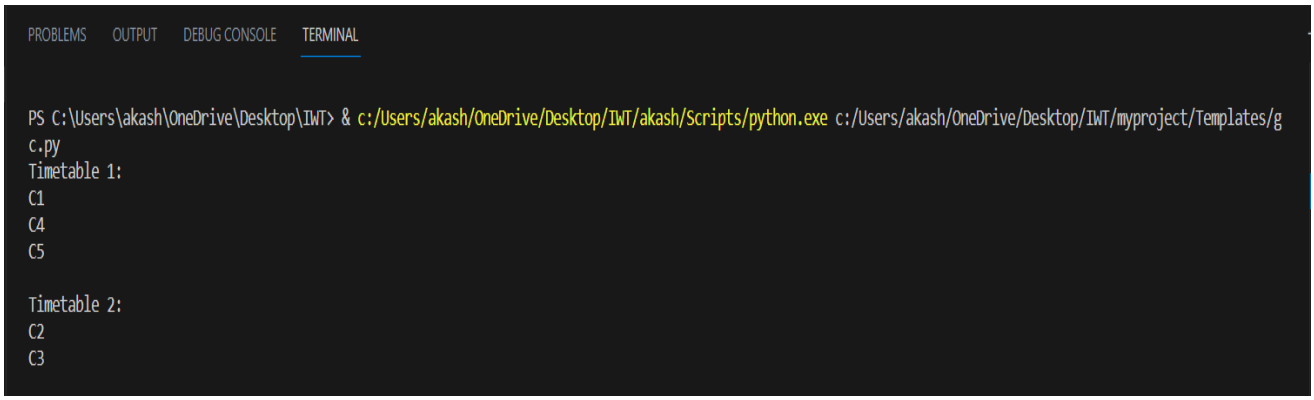
12. If the assigned color is not already present in the timetable dictionary, add it as a key with an empty list as its value.
13. Append the subject code to the list of subjects for the assigned color in the timetable dictionary.
14. Finally, the function returns the generated timetable.

CODE:

```
def generate_timetable(subjects):
    clashes = {}
    for subject in subjects:
        subject_code = subject[1]
        clashes[subject_code] = set(subject[2])  # Fix: Use subject[2] for the list of neighbors

    timetable = {}
    colors = {}

    for subject in subjects:
        subject_code = subject[1]
        available_colors = set(colors.get(neighbour, 0) for neighbour in clashes[subject_code])
# Fix: Use colors.get(neighbour, 0) to default to 0

        color = 1
        while color in available_colors:
            color += 1
        colors[subject_code] = color

        if color not in timetable:
            timetable[color] = []
        timetable[color].append(subject_code)

    return timetable

subjects = [
    ("A", "C1", ["C2", "C3"]),
    ("B", "C2", ["C1"]),
    ("C", "C3", ["C1", "C4"]),
    ("D", "C4", ["C3"]),
    ("E", "C5", []),
]

timetable = generate_timetable(subjects)
```

```
for color, subjects in timetable.items():
    print(f"Timetable {color}:")
    for subject in subjects:
        print(subject)
    print()
```

OUTPUT:



CONCLUSION AND ANALYSIS:

Time Complexity:
In the greedy approach to the graph colouring problem, we are greedily assigning a colour to each vertex of the graph as well as checking if the assigned colour meets the criteria or not (no two adjacent vertexes have the same colour). In the greedy approach to the graph colouring problem, the time complexity is O(V^2+E) in the worst case.

Space Complexity:
As the algorithm needs to store the colors assigned to each vertex it is O(V).

EXPERIMENT-11:

TITLE: Shortest Path

AIM: You are given the task of choosing the optimal path to connect "N" devices. The devices are connected with the minimum required N-1 wires into a tree structure, and each device is connected with the other with a wire of length "L" i.e.; "D1" connected to "D2" with a wire of length "L1". This information will be available for all "N" devices.
a) Determine the minimum length of the wire which consists of N-1 wires that will connect all devices.
b) Determine the minimum length of the wire which connects Di and Dj
c) Determine the minimum length of the wire which connects Di to all other devices.
d) Determine the minimum length of the wire which connects Di to all other devices where $1 \leq i \leq N$.

DESCRIPTION:

The problem described is about connecting a given number of devices in an optimal way using a minimum number of wires. The devices are represented as vertices in a tree structure, where each device is connected to the other devices with wires of different lengths.

a) The task is to determine the minimum length of the wire required to connect all the devices using N-1 wires, where N is the total number of devices. This can be solved by finding the minimum spanning tree (MST) of the graph formed by the devices and their connections. The MST algorithm finds a subset of edges that connects all the vertices with the minimum total edge weight.

b) The goal is to find the minimum length of the wire that directly connects two specific devices, Di and Dj, in the tree structure. This can be solved by applying Dijkstra's algorithm, which finds the shortest path between two vertices in a weighted graph. By setting Di as the source vertex and Dj as the target vertex, the algorithm determines the shortest path and its length, which represents the minimum wire length between Di and Dj.

c) The objective is to find the minimum length of the wire that connects a particular device Di to all other devices in the tree structure. This can be solved using Bellman Ford algorithm by setting Di as the source vertex and finding the shortest paths to all other vertices. The sum of the lengths of these shortest paths gives the minimum wire length connecting Di to all other devices.

d) The task is to find the minimum length of the wire connecting each device Di to all other devices in the tree structure, where $1 \leq i \leq N$. This can be solved by applying Floyd Warshall algorithm for each device Di in the tree. By setting Di as the source vertex and finding the shortest paths to all other vertices, the algorithm determines the minimum wire lengths connecting each device to all other devices.

a)
ALGORITHM:

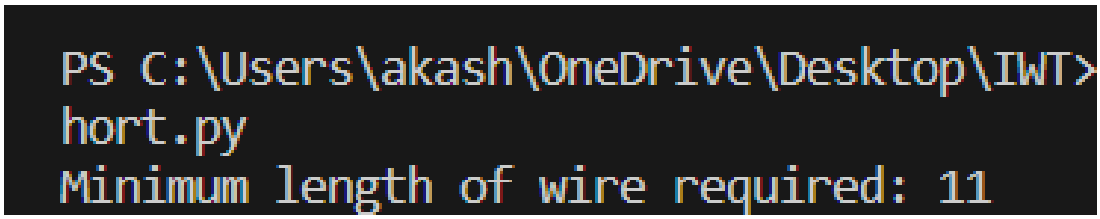
1. Sort all the edges (wires) in ascending order based on their lengths.
2. Initialize an empty set to store the selected edges of the MST.
3. Iterate through the sorted edges, starting from the smallest length:
4. If adding the current edge to the MST set does not create a cycle, add it to the set.
5. To check for a cycle, you can use a disjoint-set data structure (such as the Union-Find algorithm) to keep track of connected components.
6. Continue the iteration until you have N-1 edges in the MST set or you have processed all the edges.
7. The total length of the N-1 selected edges in the MST set will be the minimum length of the wire required to connect all devices.

CODE:

```
import heapq

def minimum_wire_length(N,connections):
    graph=[[] for _ in range(N)]
    for D1,D2,L in connections:
        graph[D1].append((D2,L))
        graph[D2].append((D1,L))
    visited=[False]*N
    min_heap = [(0, 0)]  # (length, device)
    total_length = 0
    while min_heap:
        length, device = heapq.heappop(min_heap)
        if visited[device]:
            continue
        visited[device] = True
        total_length += length
        for neighbor, neighbor_length in graph[device]:
            if not visited[neighbor]:
                heapq.heappush(min_heap, (neighbor_length, neighbor))
    return total_length

def main():
    # Example input
    N = 5  # Number of devices
    connections = [
        (0, 1, 2),  # D0 connected to D1 with wire length 2
        (0, 2, 3),  # D0 connected to D2 with wire length 3
        (1, 3, 1),  # D1 connected to D3 with wire length 1
```

```
        (2, 3, 4),  # D2 connected to D3 with wire length 4
        (2, 4, 5),  # D2 connected to D4 with wire length 5
    ]

    minimum_length = minimum_wire_length(N, connections)
    print("Minimum length of wire required:", minimum_length)

if __name__ == "__main__":
    main()
```

OUTPUT:



CONCLUSION AND ANALYSIS:

Time Complexity:
The time complexity of Kruskal's algorithm is O(E log E), where E is the number of edges. Sorting the edges initially takes O(E log E) time, and each edge is processed once in the iteration.

Space Complexity:
The space complexity is O(V + E), where V is the number of vertices and E is the number of edges.

b)
ALGORITHM:

1. Define a function dijkstra(N, graph) to implement Dijkstra's algorithm:

   -Initialize an array distances of size N with all elements set to infinity.
   -Set distances[0] = 0 to mark the starting device.
   -Create a min heap min_heap and push the tuple (0, 0) into it.
   -While min_heap is not empty:
   -Pop the minimum distance device (distance, device) from min_heap.
   -If distance is greater than distances[device], continue to the next iteration.
   -Iterate over the neighbors of device in the graph:
   -Calculate the new distance new_distance as the sum of the current distance and the -length of the edge to the neighbor.
   -If new_distance is less than the current distance to the neighbor, update distances[neighbor] and push (new_distance, neighbor) into min_heap.
2. Define the main function minimum_wire_length(N, connections):

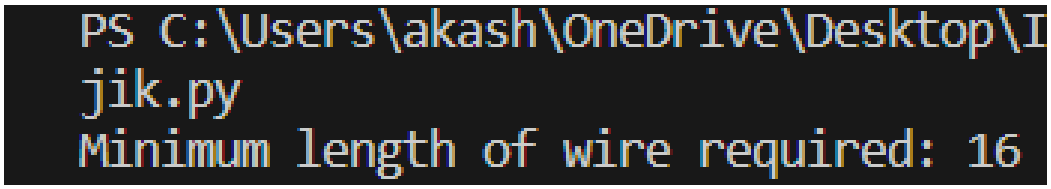
   -Create an empty graph graph with N empty lists.
   -Iterate over the connections (D1, D2, L):
   -Append (D2, L) to the list at index D1 in the graph.
   -Append (D1, L) to the list at index D2 in the graph.
   -Call the dijkstra function with N and graph to get the shortest distances.
   -Calculate the sum of all distances in the distances array to get the minimum wire length.
   -Return the minimum wire length.
3. In the main program, specify the number of devices N and the list of connections.
4. Call the minimum_wire_length function with N and connections.
5. Print the minimum length of wire required.

CODE:

```
import heapq


def dijkstra(N, graph):
    distances = [float('inf')] * N  # Initialize distances with infinity
    distances[0] = 0  # Start from device 0
    min_heap = [(0, 0)]  # (distance, device)

    while min_heap:
        distance, device = heapq.heappop(min_heap)

        if distance > distances[device]:
```

```
            continue

        for neighbor, neighbor_length in graph[device]:
            new_distance = distance + neighbor_length
            if new_distance < distances[neighbor]:
                distances[neighbor] = new_distance
                heapq.heappush(min_heap, (new_distance, neighbor))

    return distances


def minimum_wire_length(N, connections):
    graph = [[] for _ in range(N)]

    for D1, D2, L in connections:
        graph[D1].append((D2, L))
        graph[D2].append((D1, L))

    distances = dijkstra(N, graph)
    minimum_length = sum(distances)

    return minimum_length


# Example usage
N = 5  # Number of devices
connections = [
        (0, 1, 2),  # D0 connected to D1 with wire length 2
        (0, 2, 3),  # D0 connected to D2 with wire length 3
        (1, 3, 1),  # D1 connected to D3 with wire length 1
        (2, 3, 4),  # D2 connected to D3 with wire length 4
        (2, 4, 5),  # D2 connected to D4 with wire length 5
]

minimum_length = minimum_wire_length(N, connections)
print("Minimum length of wire required:", minimum_length)
```

OUTPUT:



CONCLUSION AND ANALYSIS:

Time Complexity:
Building the graph: O(E), where E is the number of connections (edges) between devices.
Dijkstra's algorithm: O((V + E) log V), where V is the number of devices (vertices) and E is the number of connections (edges).
Calculating the sum of distances: O(V), as we iterate over all devices.
Therefore, the overall time complexity is O((V + E) log V).

Space Complexity:
Graph representation: O(V + E), as we store the connections in the graph.
distances array: O(V), as we store the shortest distances from the starting device to all other devices.
min_heap: O(V), as the maximum number of devices in the min heap can be V.
Therefore, the overall space complexity is O(V + E).

c)
ALGORITHM:

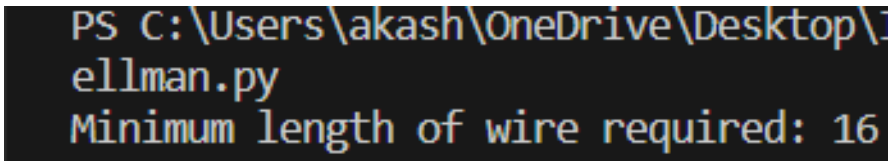
1. Initialize the distance array, distances, with infinity for all vertices except the source vertex, which is set to 0.
2. Iterate N-1 times, where N is the number of vertices.
3. For each edge (D1, D2, L) in the list of connections:
   -If distances[D1] is not infinity and distances[D1] + L is less than distances[D2], update distances[D2] to distances[D1] + L.
4. After N-1 iterations, the distances array will contain the shortest distances from the source vertex to all other vertices.
5. If there is a negative weight cycle in the graph, an additional iteration will further update the distances. If any distance is updated in this iteration, it indicates the presence of a negative weight cycle.
6. The minimum wire length required to connect all devices is the sum of the distances in the distances array.

CODE:

```
def bellman_ford(N, connections, source):
    distances = [float('inf')] * N
    distances[source] = 0

    for _ in range(N - 1):
        for D1, D2, L in connections:
            if distances[D1] != float('inf') and distances[D1] + L < distances[D2]:
                distances[D2] = distances[D1] + L

    return distances


def minimum_wire_length(N, connections, Di):
    distances = bellman_ford(N, connections, Di)
    minimum_length = sum(distances)
    return minimum_length
# Example usage
N = 5  # Number of devices
connections = [
    (0, 1, 2),  # D0 connected to D1 with wire length 2
    (0, 2, 3),  # D0 connected to D2 with wire length 3
    (1, 3, 1),  # D1 connected to D3 with wire length 1
    (2, 3, 4),  # D2 connected to D3 with wire length 4
    (2, 4, 5),  # D2 connected to D4 with wire length 5
```

```
]
Di = 0  # Device to connect to all others

minimum_length = minimum_wire_length(N, connections, Di)
print("Minimum length of wire required:", minimum_length)
```

OUTPUT:



CONCLUSION AND ANALYSIS:

Time Complexity:
The time complexity of the Bellman-Ford algorithm is O(N * M), where N is the number of vertices (devices) and M is the number of edges (connections). In the worst case, the algorithm performs N-1 iterations, each iteration checking all M edges.

Space Complexity:
The space complexity of the algorithm is O(N), as it requires an array to store the distances for each device. Additionally, it requires some additional space for variables and temporary storage, but their space requirements are typically negligible compared to the size of the graph.

d)
ALGORITHM:

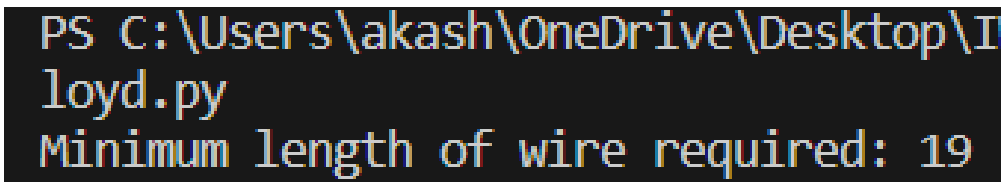
1. Create a 2D array distances of size N × N and initialize all entries to infinity.
2. For each direct connection (D1, D2, L) in the list of connections, update distances[D1][D2] and distances[D2][D1] with the respective wire lengths.
3. Apply the Floyd-Warshall algorithm to update the distances between all pairs of vertices.
   -Iterate over a middle vertex k from 0 to N-1.
   -For each pair of vertices (i, j) from 0 to N-1, update distances[i][j] by taking the minimum of the current distance distances[i][j] and the sum of the distances through vertex k (distances[i][k] + distances[k][j]).
4. After the algorithm finishes, distances[i][j] represents the minimum wire length to connect device i and device j.
5. Compute the sum of distances for each device, sum(distances[i]), and find the minimum value among them.
6. The minimum length of wire required to connect all devices is the obtained minimum value.

CODE:

```
def floyd_marshall(N, connections):
    INF = float('inf')
    distances = [[INF] * N for _ in range(N)]

    # Initialize distances with direct connections
    for D1, D2, L in connections:
        distances[D1][D2] = L
        distances[D2][D1] = L

    # Update distances using the Floyd-Warshall algorithm
    for k in range(N):
        for i in range(N):
            for j in range(N):
                distances[i][j] = min(distances[i][j], distances[i][k] + distances[k][j])

    return distances


def minimum_wire_length(N, connections):
    distances = floyd_marshall(N, connections)
    min_distances = [sum(distances[i]) for i in range(N)]
    minimum_length = min(min_distances)
```

```
    return minimum_length


# Example usage
N = 5  # Number of devices
connections = [
    (0, 1, 2),  # D0 connected to D1 with wire length 2
    (0, 2, 3),  # D0 connected to D2 with wire length 3
    (1, 3, 1),  # D1 connected to D3 with wire length 1
    (2, 3, 4),  # D2 connected to D3 with wire length 4
    (2, 4, 5),  # D2 connected to D4 with wire length 5
]

minimum_length = minimum_wire_length(N, connections)
print("Minimum length of wire required:", minimum_length)
```

OUTPUT:



CONCLUSION AND ANALYSIS:

Time Complexity:
The time complexity of the Floyd-Warshall algorithm is O(N^3), where N is the number of devices. This is because the algorithm iterates over all pairs of vertices and considers all possible intermediate vertices.

Space Complexity:
The space complexity of the algorithm is O(N^2) since it requires a 2D array of size N × N to store the distances between all pairs of vertices.

Overall, the time and space complexity of the Floyd-Warshall algorithm is O(N^3) and O(N^2) respectively.

EXPERIMENT-12:

TITLE: Bi-Connected Graphs

AIM: Bi-connected graphs are used in the design of power grid networks. Consider the nodes as cities and the edges as electrical connections between them, you would like the network to be robust and a failure at one city should not result in a loss of power in other cities.

DESCRIPTION:

An undirected graph is called Biconnected if there are two vertex-disjoint paths between any two vertices. In a Biconnected Graph, there is a simple cycle through any two vertices.
By convention, two nodes connected by an edge form a biconnected graph, but this does not verify the above properties. For a graph with more than two vertices, the above properties must be there for it to be Biconnected.
Or in other words:
A graph is said to be Biconnected if:

- It is connected, i.e., it is possible to reach every vertex from every other vertex, by a simple path.
- Even after removing any vertex the graph remains connected.

To design a robust power grid network, we can model it as a bi-connected graph, where each city is represented as a node, and the electrical connections between cities are represented as edges. By ensuring that the graph is bi-connected, we can guarantee that there are at least two distinct paths between any pair of cities, thereby providing redundancy and minimizing the impact of a single city failure on the overall network.

ALGORITHM:
1.Create a class PowerGridGraph with the following attributes:

- num_cities: The total number of cities in the power grid network.
- adj_list: An adjacency list to store the connections between cities.
- visited: A boolean array to keep track of visited cities during graph traversal.
- disc: An array to store the discovery time of each city during the bi-connectedness check.
- low: An array to store the lowest discovery time of each city during the bi-connectedness check.
- time: A variable to keep track of the current time during the bi-connectedness check.

2.Implement the add_connection method to add a connection between two cities in the power grid network. This method should update the adjacency list.

3.Implement the is_bi_connected method:

- Check if the graph is connected by calling the is_connected method. If it returns False, the power grid network is not robust, and we can return False.
- Initialize the visited, disc, and low arrays to keep track of city visits, discovery time, and lowest discovery time.
- Perform a Depth-First Search (DFS) starting from the first city in the power grid network.
- During the DFS, mark each visited city and update the discovery time and lowest discovery time for each city.
- Track the number of children in the DFS tree.
- If the current city is the root (parent is -1) and has more than one child, it is an articulation point, and the power grid network is not robust. Return False.
- If the current city is not the root and its lowest discovery time is greater than or equal to its parent's discovery time, it is an articulation point, and the power grid network is not robust. Return False.
- If the DFS completes without finding any articulation points, return True.

4.Implement the is_connected method:

- Perform a Depth-First Search (DFS) starting from the first city in the power grid network.
- Mark each visited city during the DFS.
- If all cities are visited, return True.
- If there are unvisited cities after the DFS, return False.

5.Create an instance of the PowerGridGraph class.
6.Add connections between cities using the add_connection method.
7.Call the is_bi_connected method to check if the power grid network is robust.
8.If the method returns True, print "The power grid network is bi-connected." Otherwise, print "The power grid network is not bi-connected."

CODE:

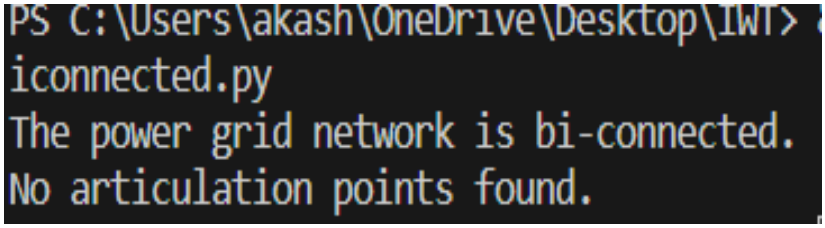
```
class PowerGridGraph:
    def __init__(self, num_cities):
        self.num_cities = num_cities
        self.adj_list = defaultdict(list)
        self.visited = [False] * num_cities
        self.disc = [0] * num_cities
        self.low = [0] * num_cities
        self.time = 0
        self.articulation_points = set()

    def add_connection(self, city1, city2):
        self.adj_list[city1].append(city2)
        self.adj_list[city2].append(city1)
```

```
    def is_bi_connected(self):
        # Check if the graph is connected
        if not self.is_connected():
            return False

        # Initialize visited, discovery time, and low value arrays
        self.visited = [False] * self.num_cities
        self.disc = [0] * self.num_cities
        self.low = [0] * self.num_cities

        # Perform a Depth-First Search from the first city
        self.is_bi_connected_util(0, -1)

        return len(self.articulation_points) == 0

    def is_bi_connected_util(self, current_city, parent):
        # Mark the current city as visited
        self.visited[current_city] = True

        # Initialize discovery time and low value for the city
        self.disc[current_city] = self.time
        self.low[current_city] = self.time
        self.time += 1

        # Count the number of children in the DFS tree
        children = 0

        # Iterate through all adjacent cities of the current city
        for next_city in self.adj_list[current_city]:
            if not self.visited[next_city]:
                children += 1
                self.is_bi_connected_util(next_city, current_city)

                # Update the low value of the current city
                self.low[current_city] = min(self.low[current_city], self.low[next_city])

                # Check if the current city is an articulation point
                if parent != -1 and self.low[next_city] >= self.disc[current_city]:
                    self.articulation_points.add(current_city)
                if parent == -1 and children > 1:
                    self.articulation_points.add(current_city)
            elif next_city != parent:
                # Update the low value of the current city for the back edge
```

```
                self.low[current_city] = min(self.low[current_city], self.disc[next_city])

    def is_connected(self):
        # Perform a Depth-First Search to check if the graph is connected
        self.visited = [False] * self.num_cities
        self.dfs(0)
        return all(self.visited)

    def dfs(self, current_city):
        self.visited[current_city] = True
        for next_city in self.adj_list[current_city]:
            if not self.visited[next_city]:
                self.dfs(next_city)

    def print_articulation_points(self):
        if len(self.articulation_points) > 0:
            print("Articulation Points:")
            for point in self.articulation_points:
                print(point)
        else:
            print("No articulation points found.")

# Example usage
power_grid = PowerGridGraph(3)
power_grid.add_connection(0, 1)
power_grid.add_connection(1, 2)
power_grid.add_connection(2, 0)

if power_grid.is_bi_connected():
    print("The power grid network is bi-connected.")
else:
    print("The power grid network is not bi-connected.")

power_grid.print_articulation_points()
```

OUTPUT:

```
PS C:\Users\akash\OneDrive\Desktop\IWT> & c:
iconnected.py
The power grid network is not bi-connected.
Articulation Points:
1
2
```



CONCLUSION AND ANALYSIS:

Time Complexity:
The time complexity of the algorithm is determined by the Depth-First Search (DFS) performed in the is_bi_connected method. In the worst-case scenario, where all cities are connected to each other, the time complexity is O(V + E), where V is the number of cities (vertices) and E is the number of connections (edges) in the power grid network. This is because the DFS visits each city once and explores all its adjacent connections.

Space Complexity:
The space complexity of the algorithm is determined by the storage of various data structures. In the PowerGridGraph class, the space complexity is O(V), where V is the number of cities, as it requires arrays to store the visited status, discovery time, and lowest discovery time for each city. Additionally, the adjacency list requires O(E) space to store the connections between cities. Overall, the general space complexity of the bi-connected graph algorithm is typically O(V^2) or O(V + E), depending on the graph representation, along with some additional constant space.