

| | |
|-----------------|---|
| EX.No:1A | IMPLEMENT RECURSIVE AND NON-RECURSIVE ALGORITHMS AND STUDY THE ORDER OF GROWTH FROM $\log_2 n$ to $n!$ |
| | |

AIM:

To implement recursive and non-recursive algorithms for various functions and study their order of growth from $\log_2 n$ to $n!$.

ALGORITHM:

1. Recursive Algorithm:

- ❖ Base Case: If n is 0 or 1, return 1.
- ❖ Recursive Case: Return n multiplied by the factorial of $(n-1)$.

The algorithm is linear, takes the running time $O(n)$.

2. Non-Recursive Algorithm:

- ❖ Initialize a variable to 1.
- ❖ Iterate from 1 to n , multiplying the current value by the iterator.

Iterator takes the running time $O(n)$

PROGRAM:

```
import time
```

```
import math
```

```
# Recursive Algorithm
```

```
def recursive_factorial(n):
```

```
    if n == 0 or n == 1:
```

```
        return 1
```

```
return n * recursive_factorial(n - 1)
```

```
# Non-Recursive Algorithm
```

```
def non_recursive_factorial(n):
```

```
    result = 1
```

```
    for i in range(1, n + 1):
```

```
        result *= i
```

```
    return result
```

```
# Measure time complexity
```

```
def measure_time_complexity(algorithm, n):
```

```
    start_time = time.time()
```

```
    algorithm(n)
```

```
    end_time = time.time()
```

```
    return end_time - start_time
```

```
# Main program
```

```
for i in range(1, 6):
```

```
    n = int(math.pow(2, i))
```

```
    recursive_time = measure_time_complexity(recursive_factorial, n)
```

```
    non_recursive_time = measure_time_complexity(non_recursive_factorial,  
n)
```

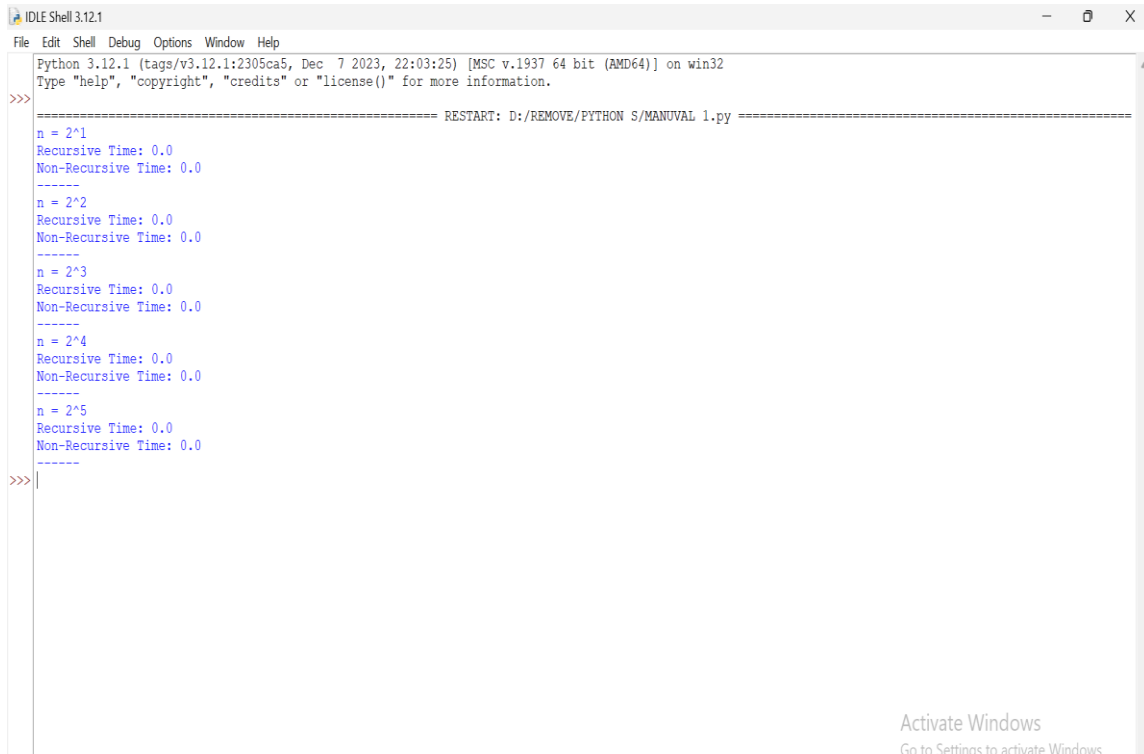
```
    print(f'n = 2^{i}')
```

```
    print(f'Recursive Time: {recursive_time}')
```

```
print(f"Non-Recursive Time: {non_recursive_time}")

print("-----")
```

OUTPUT:



```
IDLE Shell 3.12.1
File Edit Shell Debug Options Window Help
Python 3.12.1 (tags/v3.12.1:2305ca5, Dec 7 2023, 22:03:25) [MSC v.1937 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> ===== RESTART: D:/REMOVE/PYTHON S/MANUVAL 1.py =====
n = 2^1
Recursive Time: 0.0
Non-Recursive Time: 0.0
-----
n = 2^2
Recursive Time: 0.0
Non-Recursive Time: 0.0
-----
n = 2^3
Recursive Time: 0.0
Non-Recursive Time: 0.0
-----
n = 2^4
Recursive Time: 0.0
Non-Recursive Time: 0.0
-----
n = 2^5
Recursive Time: 0.0
Non-Recursive Time: 0.0
-----
>>> |
```

Activate Windows
Go to Settings to activate Windows.

RESULT:

Thus, the recursive and non-recursive algorithms for various functions and study their order of growth was implemented successfully.

| | |
|----------------|--|
| EX.No:2 | DIVIDE AND CONQUER – STRASSEN’S MATRIX MULTIPLICATION |
| | |

AIM:

To implement Strassen’s Matrix Multiplication using the Divide and Conquer approach and demonstrate its application on directed acyclic graphs (DAGs).

ALGORITHM:

Strassen's algorithm is an efficient method for multiplying two matrices using a divide and conquer strategy. Given two square matrices A and B of order $n \times n$, the goal is to compute their product $C = A \times B$.

1. Strassen's Matrix Multiplication

- Break down the matrix multiplication into subproblems using Strassen's approach
- Utilize recursive calls to solve these subproblems.
- Combine the results to obtain the final product.

PROGRAM:

```
import numpy as np
```

```
def strassen_multiply(A, B):
```

```
    n = len(A)
```

```
    # Base case: If the matrices are 1x1, perform standard multiplication
```

```
    if n == 1:
```

```
        return np.array([[A[0][0] * B[0][0]]])
```

```

# Split matrices into four equal-sized submatrices

a11, a12, a21, a22 = A[:n//2, :n//2], A[:n//2, n//2:], A[n//2:, :n//2], A[n//2:,
n//2:]

b11, b12, b21, b22 = B[:n//2, :n//2], B[:n//2, n//2:], B[n//2:, :n//2], B[n//2:,
n//2:]

# Recursive computation of seven products

p1 = strassen_multiply(a11 + a22, b11 + b22)
p2 = strassen_multiply(a21 + a22, b11)
p3 = strassen_multiply(a11, b12 - b22)
p4 = strassen_multiply(a22, b21 - b11)
p5 = strassen_multiply(a11 + a12, b22)
p6 = strassen_multiply(a21 - a11, b11 + b12)
p7 = strassen_multiply(a12 - a22, b21 + b22)

# Combine the products to get the result matrix

c11 = p1 + p4 - p5 + p7
c12 = p3 + p5
c21 = p2 + p4
c22 = p1 - p2 + p3 + p6

# Combine the submatrices into the result matrix

result = np.vstack((np.hstack((c11, c12)), np.hstack((c21, c22))))

```

```
return result
```

```
# Main program
```

```
if __name__ == "__main__":
```

```
    # Example matrices A and B
```

```
    A = np.array([[1, 2], [3, 4]])
```

```
    B = np.array([[5, 6], [7, 8]])
```

```
    # Perform Strassen's Matrix Multiplication
```

```
    result = strassen_multiply(A, B)
```

```
    # Display the result
```

```
    print("Matrix A:")
```

```
    print(A)
```

```
    print("\nMatrix B:")
```

```
    print(B)
```

```
    print("\nResultant Matrix C (Strassen's Multiplication):")
```

```
    print(result)
```

OUTPUT:

Matrix A:

[[1 2]

[3 4]]

Matrix B:

[[5 6]

[7 8]]

Resultant Matrix C (Strassen's Multiplication):

[[19 22]

[43 50]]

RESULT:

Thus, the Strassen's Matrix Multiplication using the Divide and Conquer approach was implemented successfully.

| | |
|----------------|---|
| EX.No:3 | DECREASE AND CONQUER – TOPOLOGICAL SORTING |
| | |

AIM:

To implement Topological Sorting using the Decrease and Conquer approach in Python and analyze its performance

ALGORITHM:

Topological Sorting is an ordering of vertices in a directed acyclic graph (DAG) such that for every directed edge (u, v), vertex u comes before v in the ordering. The objective is to find a topological ordering of the vertices.

1.Decrease and Conquer - Topological Sorting:

- Find a vertex with in-degree 0 (a vertex with no incoming edges).
- Remove the vertex and its outgoing edges from the graph.
- Repeat the process until all vertices are processed

PROGRAM:

```
from collections import defaultdict
```

```
class Graph:
```

```
    def __init__(self, vertices):
```

```
        self.vertices = vertices
```

```
        self.graph = defaultdict(list)
```

```
    def add_edge(self, u, v):
```

```
        self.graph[u].append(v)
```



```
def topological_sort_util(self, v, visited, stack):  
    visited[v] = True  
    for neighbor in self.graph[v]:  
        if not visited[neighbor]:  
            self.topological_sort_util(neighbor, visited, stack)  
    stack.append(v)
```

```
def topological_sort(self):  
    visited = [False] * self.vertices  
    stack = []  
  
    for i in range(self.vertices):  
        if not visited[i]:  
            self.topological_sort_util(i, visited, stack)  
    return stack[::-1]
```

Main program

```
if __name__ == "__main__":  
    # Create a graph  
    g = Graph(6)  
    g.add_edge(5, 2)  
    g.add_edge(5, 0)  
    g.add_edge(4, 0)
```

```
g.add_edge(4, 1)
g.add_edge(2, 3)
g.add_edge(3, 1)

# Perform Topological Sorting
result = g.topological_sort()

# Display the result
print("Topological Sorting Order:")
print(result)
```

OUTPUT:

Topological Sorting Order:
[5, 4, 2, 3, 1, 0]

RESULT:

Thus, the Topological Sorting using the Decrease and Conquer approach was implemented successfully.

| | |
|----------------|--|
| EX.No:4 | TRANSFORM AND CONQUER – HEAP SORT |
| | |

AIM:

To implement Heap Sort using the Transform and Conquer approach in Python and analyze performance.

ALGORITHM:

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to build a max-heap (or min-heap) and then perform a heap-based sorting. The objective is to sort an array in ascending (or descending) order.

Transform and Conquer - Heap Sort:

- Transform the input array into a max-heap.
- Repeatedly extract the maximum element (root of the heap) and swap it with the last element of the heap.
- Reduce the heap size by 1 and heapify the remaining elements.
- Repeat the process until the heap is empty.

PROGRAM:

```
def heapify(arr, n, i):
    largest = i
    left_child = 2 * i + 1
    right_child = 2 * i + 2

    if left_child < n and arr[i] < arr[left_child]:
        largest = left_child
```

```

    if right_child < n and arr[largest] < arr[right_child]:
        largest = right_child

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(arr):
    n = len(arr)

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements one by one
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i] # swap
        heapify(arr, i, 0)

# Main program
if __name__ == "__main__":
    # Example array for testing
    arr = [12, 11, 13, 5, 6, 7]

```

```
# Perform Heap Sort
```

```
heap_sort(arr)
```

```
# Display the result
```

```
print("Sorted array using Heap Sort:")
```

```
print(arr)
```

OUTPUT:

Original array: [12, 11, 13, 5, 6, 7]

Sorted array using Heap Sort: [5, 6, 7, 11, 12, 13]

RESULT:

Thus, the Heap Sort using the Transform and Conquer approach was implemented successfully.

| | |
|-----------------|--|
| EX.No:5a | DYNAMIC PROGRAMMING – COIN CHANGE PROBLEM |
| | |

AIM:

To implement the Coin Change Problem using dynamic programming approach in Python.

ALGORITHM:

Given a set of coins and a target sum, the objective is to find the number of ways to make the target sum using any combination of the given coins.

Program Logic:

1. Create a table to store the solutions to subproblems.
2. Initialize the table with base cases.
3. Fill in the table using the recurrence relation.
4. The final value in the table represents the solution to the original problem

PROGRAM:

```
def count_ways_to_make_change(coins, target):
    n = len(coins)
    table = [0] * (target + 1)
    table[0] = 1 # There is one way to make a change of 0

    for coin in coins:
        for i in range(coin, target + 1):
            table[i] += table[i - coin]
```

```
    return table[target]

# Main program
if __name__ == "__main__":
    coins = [1, 2, 5]
    target = 5
    ways = count_ways_to_make_change(coins, target)
    print(f"Number of ways to make change for {target} is: {ways}")
```

OUTPUT:

Number of ways to make change for 5 is: 4

RESULT:

Thus, the Coin Change Problem using dynamic programming approach was implemented successfully.

| | |
|-----------------|---|
| EX.No:5b | DYNAMIC PROGRAMMING – WARSHALL’S AND FLOYD’S ALGORITHM |
| | |

AIM:

To implement the Warshall’s and Floyd’s Algorithm using dynamic programming approach in Python.

ALGORITHM:

- Initialize the solution matrix same as the input graph matrix as a first step.
- Then update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number k as an intermediate vertex, we already have considered vertices $\{0, 1, 2, \dots, k-1\}$ as intermediate vertices.
- For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
 - k is not an intermediate vertex in shortest path from i to j . We keep the value of $\text{dist}[i][j]$ as it is.
 - k is an intermediate vertex in shortest path from i to j . We update the value of $\text{dist}[i][j]$ as $\text{dist}[i][k] + \text{dist}[k][j]$, if $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$

PROGRAM:

Number of vertices in the graph

$V = 4$


```
# Define infinity as the large
# enough value. This value will be
# used for vertices not connected to each other
INF = 99999
```

```
# Solves all pair shortest path
# via Floyd Warshall Algorithm
```

```
def floydWarshall(graph):
```

```
    """ dist[][] will be the output matrix that will finally have the shortest
    distances between every pair of vertices """
```

```
    """ initializing the solution matrix same as input graph matrix OR we can
    say that the initial values of shortest distances are based on shortest paths
    considering no intermediate vertices """
```

```
    dist = list(map(lambda i: list(map(lambda j: j, i)), graph))
```

```
    """ Add all vertices one by one to the set of intermediate vertices.
```

```
    ---> Before start of an iteration,
```

```
    we have shortest distances between all pairs of vertices such that the
    shortest distances consider only the vertices in the set {0, 1, 2, .. k-1 } as
    intermediate vertices.
```

```
    ----> After the end of a iteration, vertex no. k is added to the set of
    intermediate vertices and the set becomes {0, 1, 2, .. k} """
```

```
    for k in range(V):
```

```
        # pick all vertices as source one by one
```

```
        for i in range(V):
```

```

# Pick all vertices as destination for the
# above picked source
for j in range(V):

    # If vertex k is on the shortest path from
    # i to j, then update the value of dist[i][j]
    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j] )

printSolution(dist)

# A utility function to print the solution
def printSolution(dist):
    print("Following matrix shows the shortest distances\
between every pair of vertices")
    for i in range(V):
        for j in range(V):
            if(dist[i][j] == INF):
                print("%7s" % ("INF"), end=" ")
            else:
                print("%7d\t" % (dist[i][j]), end=' ')
            if j == V-1:
                print()

# Driver's code

```

```
if __name__ == "__main__":
```

```
    """
```

```
        10
```

```
        (0)----->(3)
```

```
        |         /\
```

```
5 |         |
```

```
    |         | 1
```

```
    \/\         |
```

```
        (1)----->(2)
```

```
        3         """
```

```
graph = [[0, 5, INF, 10],
```

```
         [INF, 0, 3, INF],
```

```
         [INF, INF, 0, 1],
```

```
         [INF, INF, INF, 0]
```

```
    ]
```

```
# Function call
```

```
    floydWarshall(graph)
```

OUTPUT:

The following matrix shows the shortest distances between every pair of vertices

| | | | |
|-----|-----|-----|---|
| 0 | 5 | 8 | 9 |
| INF | 0 | 3 | 4 |
| INF | INF | 0 | 1 |
| INF | INF | INF | 0 |

RESULT:

Thus, the Warshall's and Floyd's Algorithm using dynamic programming approach was implemented successfully.

| | |
|-----------------|---|
| EX.No:5c | DYNAMIC PROGRAMMING – KNAPSACK PROBLEM |
| | |

AIM:

To implement the Knapsack Problem using dynamic programming approach in Python.

ALGORITHM:

- Knapsack dynamic programming is to store the answers to solved subproblems in a table.
- All potential weights from '1' to 'W' are the columns in the table, and weights are the rows.
- The state $DP[i][j]$ reflects the greatest value of 'j-weight' considering all values from '1 to ith'. So, if we consider 'wi' (weight in 'ith' row), it is added to all columns with 'weight values $> w_i$ '.
- There are two options: fill or leave 'wi' blank in that particular column.
- If we do not enter the 'ith' weight in the 'jth' column, the $DP[i][j]$ will be same as $DP[i-1][j]$.
- However, if we fill the weight, $DP[i][j]$ equals the value of 'wi'+ the value of the column weighing 'j-wi' on the former row.
- As a result, we choose the best of these two options to fill the present condition.

PROGRAM:

```
def knapSack(W, wt, val, n):
    K = [[0 for x in range(W + 1)] for x in range(n + 1)]
    # Build table K[][] in bottom up manner
    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]
return K[n][W]
# Driver code
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)
print(knapSack(W, wt, val, n))

```

OUTPUT:

220

RESULT:

Thus, the Knapsack Problem using dynamic programming approach was implemented successfully.

| | |
|-----------------|---|
| EX.No:6a | GREEDY TECHNIQUE – DIJKSTRA'S ALGORITHM HUFFMAN TREE AND CODES |
| | |

AIM:

To implement Dijkstra's Algorithm using Greedy Technique in Python for finding the shortest path in a weighted graph.

ALGORITHM:

Given a weighted graph and a source vertex, the objective is to find the shortest path from the source to all other vertices.

Program Logic:

1. Initialize the distance of all vertices from the source as infinity, and the distance of the source vertex to itself as 0.
2. Create a priority queue to store vertices and their distances.
3. While the priority queue is not empty, extract the vertex with the minimum distance.
4. Update the distances of adjacent vertices if a shorter path is found.
5. Repeat until all vertices are processed.

PROGRAM:

```
import heapq
```

```
def dijkstra(graph, source):
```

```
    distances = {vertex: float('infinity') for vertex in graph}
```

```
    distances[source] = 0
```

```
    priority_queue = [(0, source)]
```

```

while priority_queue:
    current_distance, current_vertex = heapq.heappop(priority_queue)
    if current_distance > distances[current_vertex]:
        continue
    for neighbor, weight in graph[current_vertex].items():
        distance = current_distance + weight
        if distance < distances[neighbor]:
            distances[neighbor] = distance
            heapq.heappush(priority_queue, (distance, neighbor))
return distances

```

Main program

```

if __name__ == "__main__":
    # Example graph represented as an adjacency list
    graph = {
        'A': {'B': 1, 'C': 4},
        'B': {'A': 1, 'C': 2, 'D': 5},
        'C': {'A': 4, 'B': 2, 'D': 1},
        'D': {'B': 5, 'C': 1}
    }
    source_vertex = 'A'
    shortest_distances = dijkstra(graph, source_vertex)
    print(f"Shortest distances from {source_vertex}: {shortest_distances}")

```


OUTPUT:

Shortest distances from A: {'A': 0, 'B': 1, 'C': 3, 'D': 4}

RESULT:

Thus, the Dijkstra's Algorithm using Greedy Technique was implemented successfully.

| | |
|-----------------|--|
| EX.No:6b | GREEDY TECHNIQUE – HUFFMAN TREE AND CODES |
| | |

AIM:

To implement Huffman tree and codes using Greedy Technique in Python for finding the shortest path in a weighted graph.

ALGORITHM:

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.
3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

PROGRAM:

```
# A Huffman Tree Node
```

```
import heapq
```

```
class node:
```

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

```
# frequency of symbol
```

```
self.freq = freq
```

```
# symbol name (character)
```

```
self.symbol = symbol
```

```
# node left of current node
```

```
self.left = left
```

```
# node right of current node
```

```
self.right = right
```

```
# tree direction (0/1)
```

```
self.huff = "
```

```
def __lt__(self, nxt):
```

```
    return self.freq < nxt.freq
```

```
# utility function to print huffman
```

```
# codes for all symbols in the newly
```

```
# created Huffman tree
```

```
def printNodes(node, val=""):
```

```
# huffman code for current node
```

```
newVal = val + str(node.huff)
```

```
# if node is not an edge node
```

```
# then traverse inside it
```

```
if(node.left):
```

```
    printNodes(node.left, newVal)
```

```
if(node.right):
```

```
    printNodes(node.right, newVal)
```

```
# if node is edge node then
```

```
# display its huffman code
```

```
if(not node.left and not node.right):
```

```
    print(f"{node.symbol} -> {newVal}")
```

```
# characters for huffman tree
```

```
chars = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
# frequency of characters
```

```
freq = [5, 9, 12, 13, 16, 45]
```

```
# list containing unused nodes
```

```
nodes = []
```

```
# converting characters and frequencies
# into huffman tree nodes
for x in range(len(chars)):
    heapq.heappush(nodes, node(freq[x], chars[x]))
while len(nodes) > 1:

    # sort all the nodes in ascending order
    # based on their frequency
    left = heapq.heappop(nodes)
    right = heapq.heappop(nodes)

    # assign directional value to these nodes
    left.huff = 0
    right.huff = 1

    # combine the 2 smallest nodes to create
    # new node as their parent
    newNode = node(left.freq+right.freq, left.symbol+right.symbol, left,
right)
    heapq.heappush(nodes, newNode)

# Huffman Tree is ready!
printNodes(nodes[0])
```

OUTPUT:

f: 0

c: 100

d: 101

a: 1100

b: 1101

e: 111

RESULT:

Thus, the Huffman tree and codes using Greedy Technique was implemented successfully.

| | |
|----------------|---|
| EX.No:7 | ITERATIVE IMPROVEMENT – SIMPLEX METHOD |
| | |

AIM:

To implement the Simplex Method using Iterative Improvement in Python for solving linear programming problems.

ALGORITHM:

1. Formulate the initial tableau.
2. Iterate through the tableau until an optimal solution is found or it is determined that the solution is unbounded.
3. Choose a pivot column and pivot row.
4. Update the tableau using pivot operations.
5. Repeat until an optimal solution is achieved.

PROGRAM:

```
import numpy as np
```

```
def simplex_method(c, A, b):
```

```
    m, n = A.shape
```

```
    tableau = np.hstack((A, np.eye(m)))
```

```
    c = np.hstack((c, np.zeros(m)))
```

```
    basic_vars = np.arange(n, n + m)
```

```
    while True:
```

```

    # Check if the current solution
    #reaches the optimal solution
    if np.all(c <= 0):
        optimal_solution = tableau[:, -1]
        optimal_value = -tableau[-1, -1]
        return optimal_solution[:n], optimal_value

    # Choose the pivot column (minimum coefficient in the objective
    function)
    pivot_col = np.argmin(c)

    # Check for unbounded solution
    if np.all(tableau[:, pivot_col] <= 0):
        raise Exception("The solution is unbounded.")

    # Choose the pivot row (minimum ratio test)
    pivot_row = np.argmin(tableau[:-1, -1] / tableau[:-1, pivot_col])

    # Update the tableau using pivot operations
    pivot = tableau[pivot_row, pivot_col]
    tableau[pivot_row, :] /= pivot
    for i in range(m + 1):
        if i != pivot_row:
            tableau[i, :] -= tableau[i, pivot_col] * tableau[pivot_row, :]

```



```

    # Update basic variables
    basic_vars[pivot_row] = pivot_col

# Main program
if __name__ == "__main__":
    # Example linear programming problem
    c = np.array([-2, -3, 0, 0])
    A = np.array([[1, -1, 1, 0],
                  [3, 1, 0, 1]])
    b = np.array([2, 5])
    optimal_solution, optimal_value = simplex_method(c, A, b)
    print("Optimal Solution:", optimal_solution)
    print("Optimal Value:", optimal_value)

```

OUTPUT:

Optimal Solution: [3. 2. 0. 0.]

Optimal Value: -13.0

RESULT:

Thus, the Simplex Method using Iterative Improvement was implemented successfully.

| | |
|-----------------|---------------------------------------|
| EX.No:8a | BACKTRACKING – N-QUEEN PROBLEM |
| | |

AIM:

To implement the N-Queen Problem using the Backtracking algorithm in Python.

ALGORITHM:

The N-Queen problem is to place N chess queens on an $(N \times N)$ chessboard in such a way that no two queens threaten each other.

1. Start with an empty chessboard.
2. Place queens one by one in different columns, starting from the leftmost column.
3. Check if the current placement is safe. If not, backtrack and try the next position.
4. Repeat the process until all queens are placed or it's determined that no solution exists.

PROGRAM:

```
def is_safe(board, row, col, n):

    # Check if there is a queen in the same row
    if any(board[row]):
        return False

    # Check if there is a queen in the same column
    if any(board[i][col] for i in range(n)):
```

```
    return False
```

```
    # Check if there is a queen in the same diagonal (left-top to right-bottom)
```

```
    if any(board[i][j] for i, j in zip(range(row, -1, -1), range(col, -1, -1))):
```

```
        return False
```

```
    # Check if there is a queen in the same diagonal (right-top to left-bottom)
```

```
    if any(board[i][j] for i, j in zip(range(row, -1, -1), range(col, n))):
```

```
        return False
```

```
    return True
```

```
def solve_n_queens_util(board, row, n, solutions):
```

```
    if row == n:
```

```
        # Found a solution, add it to the list
```

```
        solutions.append([".join(row) for row in board])
```

```
    return
```

```
    for col in range(n):
```

```
        if is_safe(board, row, col, n):
```

```
            board[row][col] = 'Q'
```

```
            solve_n_queens_util(board, row + 1, n, solutions)
```

```
            board[row][col] = '.' # Backtrack
```

```
def solve_n_queens(n):  
    board = [['.' for _ in range(n)] for _ in range(n)]  
    solutions = []  
    solve_n_queens_util(board, 0, n, solutions)  
    return solutions  
  
# Main program  
if __name__ == "__main__":  
    n = 4  
    solutions = solve_n_queens(n)  
  
    print(f"Number of solutions for {n}-Queens problem: {len(solutions)}")  
    for i, solution in enumerate(solutions):  
        print(f"\nSolution {i + 1}:")  
        for row in solution:  
            print(row)
```

OUTPUT:

Number of solutions for 4-Queens problem: 2

Solution 1:

| | | | |
|---|---|---|---|
| . | Q | . | . |
| . | . | . | Q |
| Q | . | . | . |
| . | . | Q | . |

Solution 2:

| | | | |
|---|---|---|---|
| . | . | Q | . |
| Q | . | . | . |
| . | . | . | Q |
| . | Q | . | . |

RESULT:

Thus, the N-Queen Problem using the Backtracking algorithm was implemented successfully.

| | |
|-----------------|--|
| EX.No:8b | BACKTRACKING – SUBSET SUM PROBLEM |
| | |

AIM:

To implement the Subset Sum Problem using the Backtracking algorithm in Python.

ALGORITHM:

Given a set of positive integers and a target sum, determine if there is a subset of the set that adds up to the target sum.

1. Start with an empty subset.
2. Include an element in the subset and recursively check if the remaining sum can be obtained.
3. Exclude the element from the subset and recursively check if the sum can be obtained without the element.
4. Repeat the process for each element in the set.

PROGRAM:

```
def subset_sum_util(nums, target, current_sum, start, path, result):
```

```
    if current_sum == target:
```

```
        result.append(path[:])
```

```
    return
```

```
    for i in range(start, len(nums)):
```

```
        if current_sum + nums[i] <= target:
```

```
            path.append(nums[i])
```

```
        subset_sum_util(nums, target, current_sum + nums[i], i + 1, path,
result)
        path.pop()
```

```
def subset_sum(nums, target):
    result = []
    subset_sum_util(nums, target, 0, 0, [], result)
    return result
```

```
# Main program
if __name__ == "__main__":
    nums = [1, 2, 3, 4, 5]
    target = 7
    subsets = subset_sum(nums, target)
    print(f"Subsets with sum equal to {target}:")
    for subset in subsets:
        print(subset)
```

OUTPUT:

Subsets with sum equal to 7:

[1, 2, 4]

[2, 5]

[3, 4]

RESULT:

Thus, the Subset Sum Problem using the Backtracking algorithm was implemented successfully.

| | |
|-----------------|--|
| EX.No:9a | BRANCH AND BOUND – ASSIGNMENT PROBLEM |
| | |

AIM:

To implement the Assignment Problem using the Branch and Bound algorithm in Python.

ALGORITHM:

Given a set of cities and the distances between each pair of cities, find the shortest possible tour that visits each city exactly once and returns to the starting city

1. Create a cost matrix for the assignment problem.
2. Implement the Branch and Bound algorithm to find the optimal assignment.

PROGRAM:

```
import numpy as np
from itertools import permutations

def assignment_cost(assignment, cost_matrix):
    total_cost = 0
    for worker, task in enumerate(assignment):
        total_cost += cost_matrix[worker, task]
    return total_cost

def branch_and_bound_assignment(cost_matrix):
```

```

n = len(cost_matrix)
min_cost = float('inf')
optimal_assignment = None

for assignment in permutations(range(n)):
    current_cost = assignment_cost(assignment, cost_matrix)
    if current_cost < min_cost:
        min_cost = current_cost
        optimal_assignment = assignment
return optimal_assignment, min_cost

# Main program
if __name__ == "__main__":
    # Example cost matrix for the assignment problem
    cost_matrix = np.array([
        [9, 2, 7, 8],
        [6, 4, 3, 7],
        [5, 8, 1, 8],
        [7, 6, 9, 4] ])

    optimal_assignment, min_cost = branch_and_bound_assignment(cost_matrix)
    print("Optimal Assignment:", optimal_assignment)
    print("Minimum Cost:", min_cost)

```

OUTPUT:

Optimal Assignment: (2, 3, 0, 1)

Minimum Cost: 13

RESULT:

Thus, the Assignment Problem using the Branch and Bound algorithm was implemented successfully.

| | |
|-----------------|--|
| EX.No:9b | BRANCH AND BOUND – TRAVELING SALESMAN PROBLEM |
| | |

AIM:

To implement the Traveling Salesman Problem using the Branch and Bound algorithm in Python.

ALGORITHM:

Given a set of cities and the distances between each pair of cities, find the shortest possible tour that visits each city exactly once and returns to the starting city.

1. Create a distance matrix for the TSP.
2. Implement the Branch and Bound algorithm to find the optimal tour.

PROGRAM:

```
import numpy as np
```

```
def tsp_cost(tour, distance_matrix):
```

```
    total_cost = 0
```

```
    for i in range(len(tour) - 1):
```

```
        total_cost += distance_matrix[tour[i], tour[i + 1]]
```

```
    total_cost += distance_matrix[tour[-1], tour[0]]
```

```
# Return to the starting city
```

```
    return total_cost
```

```

def branch_and_bound_tsp(distance_matrix):
    n = len(distance_matrix)
    optimal_tour = None
    min_cost = float('inf')
    initial_tour = list(range(n))

    def tsp_recursive(tour, cost):
        nonlocal optimal_tour, min_cost

        if len(tour) == n:
            # Completed a full tour, update optimal solution if needed
            if cost < min_cost:
                min_cost = cost
                optimal_tour = tour[:]
            return

        for city in range(n):
            if city not in tour:
                tsp_recursive(tour + [city], cost + distance_matrix[tour[-1], city])

    tsp_recursive(initial_tour, 0)
    return optimal_tour, min_cost

# Main program
if __name__ == "__main__":

```

```
# Example distance matrix for the TSP

distance_matrix = np.array([
    [0, 10, 15, 20],
    [10, 0, 35, 25],
    [15, 35, 0, 30],
    [20, 25, 30, 0] ])

optimal_tour, min_cost = branch_and_bound_tsp(distance_matrix)
print("Optimal Tour:", optimal_tour)
print("Minimum Cost:", min_cost)
```

OUTPUT:

Optimal Tour: [0, 1, 3, 2]

Minimum Cost: 80

RESULT:

Thus, the Traveling Salesman Problem using the Branch and Bound algorithm was implemented successfully.