Practical-01

Aim: Write a Program to implement insertion sort and find the running time.

Theory:

Insertion sort is a simple comparison-based sorting algorithm that builds the final sorted array one element at a time. It is efficient for small data sets but becomes less practical as the data size grows. The key idea behind insertion sort is to maintain a "sorted portion" of the array and repeatedly insert the next unsorted element into its correct position within this sorted portion.It is an in-place sorting algorithm, meaning it doesn't require additional memory for sorting.It is a stable sort, meaning it preserves the relative order of equal elements.Its time complexity is O(n^2) in the worst and average cases, where 'n' is the number of elements. This makes it inefficient for large datasets.In the best case (when the input array is already sorted), its time complexity is O(n), which makes it efficient.

Algorithm:

1. import random and import time are used to import the necessary modules for generating random numbers and measuring time.
2. The insertion\_sort function is defined to implement the insertion sort algorithm. Here's how it works:

* It takes a list (arr) as input.
* It iterates through the list from the second element (index 1) to the last element.
* For each element at the current position (designated by i), it temporarily stores its value in the variable key.
* It then compares the key with the elements to the left (indexed by j) and shifts larger elements to the right until it finds the correct position for the key.

1. The sorted subarray is expanded with each iteration until the entire list is sorted.
2. The generate\_random\_list function is defined to create a list of random integers within a specified range (1 to 100). The size of the list is determined by the size parameter.
3. The main function is where the main program logic is executed. Here's what it does:
4. It sets the list\_size variable to determine the size of the list to be sorted. You can adjust this value to change the size of the list.
5. It generates a random list of integers using the generate\_random\_list function.
6. It records the start time using time.time().
7. It calls the insertion\_sort function to sort the random list.
8. It records the end time using time.time().
9. It calculates the elapsed time by subtracting the start time from the end time.
10. It prints the sorted list and the time taken to sort the list in seconds with six decimal places of precision.
11. Finally, the program checks if it's being run as the main script (not imported as a module), and if so, it calls the main function to execute the sorting and timing process.
12. To run this code, you need to have Python installed on your system. Adjust the list\_size variable to change the size of the list you want to sort, and the program will output the sorted list and the time taken to perform the insertion sort.

Code:

import random

import time

def insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i]

j = i - 1

while j >= 0 and key < arr[j]:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

def generate\_random\_list(size):

return [random.randint(1, 100) for \_ in range(size)]

def main():

# Adjust this value to change the size of the list to be sorted

list\_size = 100

random\_list = generate\_random\_list(list\_size)

start\_time = time.time()

insertion\_sort(random\_list)

end\_time = time.time()

elapsed\_time = end\_time - start\_time

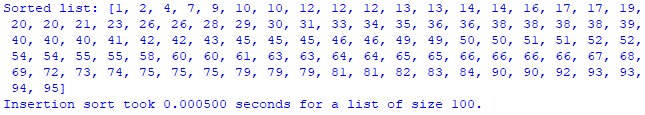
print(f"Sorted list: {random\_list}")

print(f"Insertion sort took {elapsed\_time:.6f} seconds for a list of size {list\_size}.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:



Conclusion:

The program efficiently implements the insertion sort algorithm and accurately measures its running time, providing a practical tool for analyzing algorithm performance with varying input sizes.

Practical-02

Aim:Write the Program to implement merge sort algorithm

Compare the time and memory complexity.

Theory:

Merge Sort is a popular and efficient comparison-based sorting algorithm that follows the divide and conquer paradigm. It works by dividing an unsorted list into smaller sublists, sorting those sublists, and then merging them back together into a single sorted list. The primary steps and key concepts of the Merge Sort algorithm are as follows:

Divide: The unsorted list is divided into two halves. This step continues recursively until each sublist contains only one element, which is, by definition, sorted.

Conquer (Sort): The sublists are sorted. This is often done recursively by applying the Merge Sort algorithm to each sublist. Sorting is accomplished by comparing elements and merging them in sorted order.

Merge: The sorted sublists are merged back together into a single, larger sorted list. The merging process involves comparing elements from the two sublists and placing them in the correct order in the final sorted list.

Algorithm:

1)Merge Sort Function (merge\_sort):

This function takes an array as input and sorts it in ascending order using the Merge Sort algorithm.

It recursively divides the array into smaller subarrays, sorts them individually, and then merges them back together in a sorted manner.

2)Merge Sort Wrapper Function (merge\_sort\_wrapper):

This function is used to measure the running time and memory usage of the merge\_sort function.

It records the start time using time.time().

It calls the memory\_usage function to monitor memory usage while the merge\_sort function is executed.

After the merge\_sort function completes, it records the end time.

It calculates the elapsed time by subtracting the start time from the end time.

It also finds the maximum memory usage during the execution of the merge\_sort function.

3)Example Usage (Inside the if \_\_name\_\_ == '\_\_main\_\_': block):

An example list my\_list is provided with unsorted values.

The merge\_sort\_wrapper function is called to sort the list and measure the performance.

The sorted list is printed to the console.

The elapsed time is printed, indicating how long the Merge Sort took to sort the list.

The maximum memory usage during the sorting process is printed.

Code:

import time

from memory\_profiler import memory\_usage

def merge\_sort(arr):

if len(arr) > 1:

mid = len(arr) // 2

left\_half = arr[:mid]

right\_half = arr[mid:]

merge\_sort(left\_half)

merge\_sort(right\_half)

i = j = k = 0

while i < len(left\_half) and j < len(right\_half):

if left\_half[i] < right\_half[j]:

arr[k] = left\_half[i]

i += 1

else:

arr[k] = right\_half[j]

j += 1

k += 1

while i < len(left\_half):

arr[k] = left\_half[i]

i += 1

k += 1

while j < len(right\_half):

arr[k] = right\_half[j]

j += 1

k += 1

def merge\_sort\_wrapper(arr):

start\_time = time.time() # Record the start time

mem\_usage = memory\_usage((merge\_sort, (arr,)))

end\_time = time.time() # Record the end time

elapsed\_time = end\_time - start\_time

max\_memory\_usage = max(mem\_usage)

return elapsed\_time, max\_memory\_usage

if \_\_name\_\_ == '\_\_main\_\_':

# Example usage:

my\_list = [38, 27, 43, 3, 9, 82, 10]

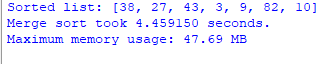
elapsed\_time, max\_memory\_usage = merge\_sort\_wrapper(my\_list)

print("Sorted list:", my\_list)

print(f"Merge sort took {elapsed\_time:.6f} seconds.")

print(f"Maximum memory usage: {max\_memory\_usage:.2f} MB")

Output:



Conclusion: We Have Successfully implemented the merge sort algorithm.

Practical-03

Aim:Write the Program to Strassen’s algorithm for matix multiplication and analyze its complexity.

Theory:

Divide-and-Conquer:Instead of directly computing the product of two matrices A and B in the traditional manner, the algorithm divides each matrix into four submatrices.

Recursive Multiplication:Utilizes these submatrices to perform fewer multiplications.

Combine Submatrices:The algorithm uses these partial products and employs addition and subtraction operations to compute the resulting product.

Time Complexity:

The standard matrix multiplication algorithm has a time complexity of ( 3)O(n3) for two × n×n matrices.

Strassen's algorithm reduces the number of individual multiplications required to approximately ( 27)O(nlog2​7) or about ( 2.81)O(n2.81), significantly reducing the number of multiplications needed.

Space Complexity:

The space complexity for the algorithm is also ( 2)O(n2) due to the need to store submatrices and the final resulting matrix.

Algorithm:

strassen\_matrix\_multiply Function:

1)Base Case:If the matrix size is 1x1, it directly returns the product of the single elements.

2)Divide Step:It divides the input matrices A and B into four submatrices.Creates submatrices A11, A12, A21, A22, B11, B12, B21, and B22.

3)Recursive Multiplication:Performs seven recursive multiplications (P1 to P7) using the submatrices, employing specific combinations of additions and subtractions.

4)Combine Submatrices:Combines the results of recursive multiplications to compute the resulting submatrices C11, C12, C21, and C22.

5)Form the Result Matrix:Constructs the final result matrix by arranging the calculated submatrices.

6)matrix\_add and matrix\_subtract Functions:These functions perform addition and subtraction of matrices element-wise and return the resulting matrix.

Code:

def strassen\_matrix\_multiply(A, B):

n = len(A)

# Base case: if the matrix size is 1x1, return the product

if n == 1:

return [[A[0][0] \* B[0][0]]]

# Divide the matrices into four submatrices

mid = n // 2

A11 = [row[:mid] for row in A[:mid]]

A12 = [row[mid:] for row in A[:mid]]

A21 = [row[:mid] for row in A[mid:]]

A22 = [row[mid:] for row in A[mid:]]

B11 = [row[:mid] for row in B[:mid]]

B12 = [row[mid:] for row in B[:mid]]

B21 = [row[:mid] for row in B[mid:]]

B22 = [row[mid:] for row in B[mid:]]

# Recursive matrix multiplication using Strassen's algorithm

P1 = strassen\_matrix\_multiply(A11, matrix\_subtract(B12, B22))

P2 = strassen\_matrix\_multiply(matrix\_add(A11, A12), B22)

P3 = strassen\_matrix\_multiply(matrix\_add(A21, A22), B11)

P4 = strassen\_matrix\_multiply(A22, matrix\_subtract(B21, B11))

P5 = strassen\_matrix\_multiply(matrix\_add(A11, A22), matrix\_add(B11, B22))

P6 = strassen\_matrix\_multiply(matrix\_subtract(A12, A22), matrix\_add(B21, B22))

P7 = strassen\_matrix\_multiply(matrix\_subtract(A11, A21), matrix\_add(B11, B12))

# Calculate the resulting submatrices

C11 = matrix\_add(matrix\_subtract(matrix\_add(P5, P4), P2), P6)

C12 = matrix\_add(P1, P2)

C21 = matrix\_add(P3, P4)

C22 = matrix\_subtract(matrix\_subtract(matrix\_add(P5, P1), P3), P7)

# Combine the submatrices to form the result matrix

result = [[0] \* n for \_ in range(n)]

for i in range(mid):

for j in range(mid):

result[i][j] = C11[i][j]

result[i][j + mid] = C12[i][j]

result[i + mid][j] = C21[i][j]

result[i + mid][j + mid] = C22[i][j]

return result

def matrix\_add(A, B):

return [[A[i][j] + B[i][j] for j in range(len(A[0]))] for i in range(len(A))]

def matrix\_subtract(A, B):

return [[A[i][j] - B[i][j] for j in range(len(A[0]))] for i in range(len(A))]

if \_\_name\_\_ == "\_\_main\_\_":

A = [[1, 2, 3, 4],

[5, 6, 7, 8],

[9, 10, 11, 12],

[13, 14, 15, 16]]

B = [[17, 18, 19, 20],

[21, 22, 23, 24],

[25, 26, 27, 28],

[29, 30, 31, 32]]

result = strassen\_matrix\_multiply(A, B)

print("Matrix A:")

for row in A:

print(row)

print("Matrix B:")

for row in B:

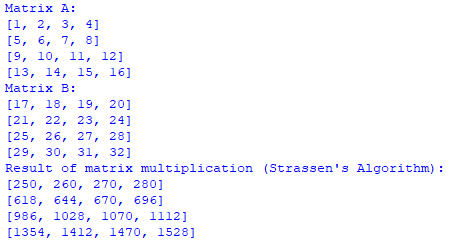
print(row)

print("Result of matrix multiplication (Strassen's Algorithm):")

for row in result:

print(row)

Output:



Conclusion: We Have Successfully implemented Strassen’s algorithm for matrix multiplication.

Practical:04

Aim: Implement hiring problem and analyze its complexity.

Theory:The hiring problem, also known as the Secretary Problem or the Best Choice Problem, is a classic problem in probability theory and decision theory. The scenario involves selecting the best candidate from a sequence of applicants as they arrive for an interview. The objective is to maximize the chance of selecting the best candidate while rejecting candidates before the final decision is made.The problem involves a strategy to decide when to stop interviewing and make a selection, based on partial information about the quality of candidates already interviewed. The typical solution suggests interviewing a fixed percentage of candidates first to gather information about the range of candidate quality and then selecting the first candidate who is better than the previously interviewed ones.The time complexity of the basic solution for the hiring problem is O(n), where n is the number of candidates.

Algorithm:

hiring\_problem function:

1)Parameters: candidates is a list representing the pool of candidates. Each candidate has a numerical value associated with their quality or suitability for the job.

2)Logic: The function iterates through the list of candidates to find the best candidate. It starts with the first candidate and compares it with the remaining candidates to find the one with the highest value.

main function:

1)Generates Candidates: It creates a list of candidates with random suitability values ranging between 0 and 1.

2)Calls hiring\_problem: Passes the list of generated candidates to the hiring\_problem function to determine the best candidate.

3)Prints Result: Displays the best candidate obtained by the hiring\_problem function.

Complexity Analysis:

Time Complexity: The time complexity of the hiring\_problem function is O(n), where n is the number of candidates. It iterates through the list once to find the best candidate, comparing each candidate's suitability against the current best candidate.

Space Complexity: The space complexity of this specific code is O(n) as well, where n is the number of candidates. It's mainly due to the storage of the list of candidates.

Code:

import random

def hiring\_problem(candidates):

best\_candidate = candidates[0]

n = len(candidates)

for i in range(1, n):

if candidates[i] > best\_candidate:

best\_candidate = candidates[i]

return best\_candidate

def main():

num\_candidates = 100 # Change the number of candidates

candidates = [random.uniform(0, 1) for \_ in range(num\_candidates)]

best = hiring\_problem(candidates)

print("Best candidate:", best)

if \_\_name\_\_ == "\_\_main\_\_":

main()

Output:



Conclusion: We Have Successfully implemented the hiring Problem

Practical:05

Aim: Write a program to implement Longest Common Subsequent(LCS) algorithm.

Theory:

The Longest Common Subsequence (LCS) algorithm is a dynamic programming technique used to find the longest subsequence common to two sequences or strings. A subsequence is a sequence that can be derived from another sequence by deleting some elements without changing the order of the remaining elements.Given two sequences or strings, the LCS algorithm finds the longest sequence that is present in both of them. This sequence does not need to be contiguous within the original sequences.

1)Dynamic Programming Approach:The algorithm utilizes a dynamic programming table or matrix to solve the problem efficiently.It builds the table gradually, filling in values based on the comparisons between characters of the input sequences.

2)Recurrence Relation:The core of the LCS algorithm lies in defining a recurrence relation to fill in the table.At each cell (i, j) of the table, it compares the characters of both sequences.If characters at these positions match, it extends the LCS by one and looks at the LCS of the smaller subsequences (i-1, j-1).If the characters don't match, it considers the LCS without one character, either from the first sequence (i-1, j) or from the second sequence (i, j-1).

3)Building the Table:The dynamic programming table gradually builds up these solutions for subproblems.It works from smaller subproblems to larger ones, allowing the algorithm to reuse previously computed results.

4)Backtracking:Once the table is built, it backtracks through the table, tracing the characters that form the LCS.

Algorithm:

1)Initialization:The algorithm initializes a table lcs\_table to store the lengths of the LCS for subproblems.

2)Building the LCS Table:It iterates through both strings str1 and str2 using nested loops.Compares characters of the strings and populates the lcs\_table accordingly.if characters match, it increments the LCS length by one compared to the previous subproblems.If characters do not match, it takes the maximum length achieved by excluding a character from either of the strings.

3)Backtracking:The algorithm then backtracks through the lcs\_table to find the actual sequence that represents the longest common subsequence.It starts from the end of both strings and traces back using the lengths computed in the lcs\_table, storing the characters of the LCS.

4)Result:Finally, it returns the computed LCS, which represents the longest common subsequence between the given strings.

Code:

def longest\_common\_subsequence(str1, str2):

m = len(str1)

n = len(str2)

# Initialize a table to store the lengths of LCS for subproblems

lcs\_table = [[0] \* (n + 1) for \_ in range(m + 1)]

# Building the LCS table

for i in range(1, m + 1):

for j in range(1, n + 1):

if str1[i - 1] == str2[j - 1]:

lcs\_table[i][j] = lcs\_table[i - 1][j - 1] + 1

else:

lcs\_table[i][j] = max(lcs\_table[i - 1][j], lcs\_table[i][j - 1])

# Backtracking to find the longest common subsequence

lcs = []

i, j = m, n

while i > 0 and j > 0:

if str1[i - 1] == str2[j - 1]:

lcs.append(str1[i - 1])

i -= 1

j -= 1

elif lcs\_table[i - 1][j] > lcs\_table[i][j - 1]:

i -= 1

else:

j -= 1

return ''.join(lcs[::-1]) # Reverse the list to get the LCS in correct order

# Example usage:

string1 = "ABCDGH"

string2 = "AEDFHR"

result = longest\_common\_subsequence(string1, string2)

print("Longest Common Subsequence:", result)

Output:



Conclusion: We Have Successfully implemented the Longest Common Subsequent (LCS) Algorithm.

Practical:06

Aim: Write a Program to implement Huffman’s code Algorithm.

**Theory:**

The Huffman coding algorithm is a widely used method for lossless data compression. It was developed by David A. Huffman in 1952 and is based on the principle of assigning variable-length codes to input characters, with shorter codes assigned to more frequent characters.

Frequency Analysis:Huffman coding starts by analyzing the input data to determine the frequency of each character (or symbol) in the data.

Building the Huffman Tree:Characters with higher frequency are given shorter codes to optimize compression.The algorithm constructs a binary tree, known as the Huffman tree or Huffman encoding tree, by repeatedly combining the two least frequent characters or subtrees until all characters are merged into the tree.

Prefix Codes:Huffman codes are prefix codes, meaning no code is a prefix of another. This ensures that the encoded sequence can be unambiguously decoded.

Variable-Length Codes:Characters are assigned variable-length codes, with more frequent characters having shorter codes and less frequent characters having longer codes.

Compression:Once the Huffman tree is constructed, the actual data is encoded using the codes from the tree.The encoded data is then transmitted or stored in a compressed format.

Algorithm:

1)Node class:This class represents a node in the Huffman tree.Each node has attributes: char for the character (or None for internal nodes), freq for the frequency, and left, right for the left and right children.

2)build\_huffman\_tree(freq\_dict):

Purpose: Constructs the Huffman tree.

Input: freq\_dict - a dictionary containing characters and

Their frequencies

Process:It creates nodes for each character frequency pair and appends them to the nodes list.Iteratively combines the two nodes with the lowest frequencies until a single root node remains, forming the Huffman tree.

3)generate\_huffman\_codes(root, current\_code, huffman\_codes):

Purpose: Generates Huffman codes for each character based on the constructed tree.

Input: root - the root node of the Huffman tree, current\_code - the current code being generated, huffman\_codes - stores the Huffman codes.

Process:Traverses the Huffman tree using a recursive approach to assign binary codes to each character based on the path taken to reach them.

3)huffman\_encoding(text):

Purpose: Encodes the input text using Huffman codes.

Input: text - the input text to be encoded.

Process:Calculates the frequency of each character in the input text and builds the Huffman tree using the build\_huffman\_tree function.Generates Huffman codes for each character using the generate\_huffman\_codes function.Encodes the text by replacing each character with its corresponding Huffman code.

4)huffman\_decoding(encoded\_text, huffman\_tree):

Purpose: Decodes the encoded text using the provided Huffman tree.

Input: encoded\_text - the encoded text to be decoded, huffman\_tree - the Huffman tree used for decoding.

Process:Traverses the Huffman tree according to the bits in the encoded text, reconstructing the original text.

Code:

class Node:

def \_\_init\_\_(self, char, freq):

self.char = char

self.freq = freq

self.left = None

self.right = None

def build\_huffman\_tree(freq\_dict):

nodes = [Node(char, freq) for char, freq in freq\_dict.items()]

while len(nodes) > 1:

nodes = sorted(nodes, key=lambda x: x.freq)

left = nodes.pop(0)

right = nodes.pop(0)

merge = Node(None, left.freq + right.freq)

merge.left = left

merge.right = right

nodes.append(merge)

return nodes[0]

def generate\_huffman\_codes(root, current\_code, huffman\_codes):

if root is None:

return

if root.char is not None:

huffman\_codes[root.char] = current\_code

return

generate\_huffman\_codes(root.left, current\_code + "0", huffman\_codes)

generate\_huffman\_codes(root.right, current\_code + "1", huffman\_codes)

def huffman\_encoding(text):

freq\_dict = {}

for char in text:

if char in freq\_dict:

freq\_dict[char] += 1

else:

freq\_dict[char] = 1

huffman\_tree = build\_huffman\_tree(freq\_dict)

huffman\_codes = {}

generate\_huffman\_codes(huffman\_tree, "", huffman\_codes)

encoded\_text = "".join(huffman\_codes[char] for char in text)

return encoded\_text, huffman\_tree

def huffman\_decoding(encoded\_text, huffman\_tree):

decoded\_text = ""

current = huffman\_tree

for bit in encoded\_text:

if bit == '0':

current = current.left

else:

current = current.right

if current.char is not None:

decoded\_text += current.char

current = huffman\_tree

return decoded\_text

# Example usage:

text = "Huffman coding is a data compression algorithm."

encoded\_text, tree = huffman\_encoding(text)

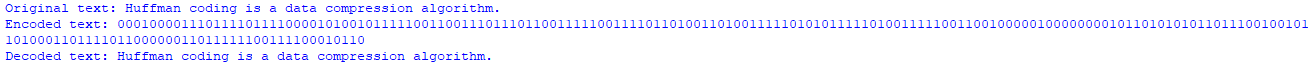
print("Original text:", text)

print("Encoded text:", encoded\_text)

decoded\_text = huffman\_decoding(encoded\_text, tree)

print("Decoded text:", decoded\_text)

Output:



Conclusion: We Have Successfully implemented the Huffman’s Coding Algorithm.

Practical:07

Aim: Write a Program to implement the Kruskal’s Algorithm.

Theory:

Kruskal's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a connected, undirected graph. The MST of a graph is a subset of its edges that forms a tree and includes all the vertices while minimizing the total edge weight.

Minimum Spanning Tree (MST):Kruskal's algorithm finds an MST that has the minimum total edge weight among all possible spanning trees for a given graph.

Greedy Algorithm:Kruskal's algorithm works in a greedy manner, selecting edges with the smallest weights while ensuring that the graph remains acyclic.

Disjoint Set (Union-Find) Data Structure:It employs a data structure like Union-Find to keep track of the connected components and efficiently determine if adding an edge forms a cycle.

Edge Sorting:The algorithm initially sorts the edges in non-decreasing order based on their weights.

Edge Selection:Starting from the edge with the smallest weight, the algorithm selects edges one by one if they don't form a cycle.

Algorithm:

1)DisjointSet class:

\_\_init\_\_(self, vertices): Initializes the Disjoint Set data structure with each vertex as its own parent and a rank of 0 for each vertex. It maintains the parent and rank dictionaries to manage disjoint sets.

find(self, vertex): Implements the find operation using path compression. It finds the root of the set to which the given vertex belongs and compresses the path by making the found root the parent of the vertex.

union(self, vertex1, vertex2): Performs the union operation by merging the disjoint sets represented by vertex1 and vertex2. It compares the ranks of the roots and attaches the set with the smaller rank to the one with the larger rank. If both ranks are the same, it increases the rank of the resulting set by one.

2)kruskal(graph) function:

Input: graph is a dictionary representing the graph with vertices as keys and their neighbors and corresponding edge weights as values.

Process:

Initialize Variables:vertices contains the list of vertices in the graph.disjoint\_set is an instance of DisjointSet.mst is an empty list to store the minimum spanning tree (MST) edges.

Prepare Edges:Create a list of all edges sorted by weight in ascending order.

Find Minimum Spanning Tree:Iterate through the sorted edges.For each edge, if the vertices at its ends do not belong to the same set in the disjoint set, add the edge to the MST, and unite the sets to which the vertices belong.

Output: Returns the list of edges forming the Minimum Spanning Tree.

Code:

class DisjointSet:

def \_\_init\_\_(self, vertices):

self.parent = {vertex: vertex for vertex in vertices}

self.rank = {vertex: 0 for vertex in vertices}

def find(self, vertex):

if self.parent[vertex] != vertex:

self.parent[vertex] = self.find(self.parent[vertex])

return self.parent[vertex]

def union(self, vertex1, vertex2):

root1 = self.find(vertex1)

root2 = self.find(vertex2)

if root1 != root2:

if self.rank[root1] > self.rank[root2]:

self.parent[root2] = root1

else:

self.parent[root1] = root2

if self.rank[root1] == self.rank[root2]:

self.rank[root2] += 1

def kruskal(graph):

vertices = list(graph.keys())

disjoint\_set = DisjointSet(vertices)

mst = []

edges = []

for vertex in graph:

for neighbor, weight in graph[vertex]:

edges.append((weight, vertex, neighbor))

edges.sort() # Sort edges based on their weights

for edge in edges:

weight, vertex1, vertex2 = edge

if disjoint\_set.find(vertex1) != disjoint\_set.find(vertex2):

disjoint\_set.union(vertex1, vertex2)

mst.append((vertex1, vertex2, weight))

return mst

# Example usage:

graph = {

'A': [('B', 3), ('C', 1)],

'B': [('A', 3), ('C', 7), ('D', 5)],

'C': [('A', 1), ('B', 7), ('D', 2)],

'D': [('B', 5), ('C', 2)]

}

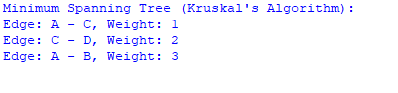
minimum\_spanning\_tree = kruskal(graph)

print("Minimum Spanning Tree (Kruskal's Algorithm):")

for edge in minimum\_spanning\_tree:

print(f"Edge: {edge[0]} - {edge[1]}, Weight: {edge[2]}")

Output:



Conclusion: We Have Successfully Implemented the Kruskal’s Algorithm.

Practical:08

Aim: Write the Program to implement Dijkshtra algorithm.

Theory:

Dijkstra's algorithm solves the Single Source Shortest Path problem, finding the shortest path from a single source vertex to all other vertices in a graph.

Greedy Algorithm:Dijkstra's algorithm is based on a greedy strategy, continually selecting the vertex with the smallest tentative distance from the source vertex and then exploring its adjacent vertices.

Data Structures:The algorithm uses data structures like priority queues (usually implemented using a min-heap) to efficiently select the vertex with the minimum distance.

Relaxation:At each step, it relaxes the distances of the vertices by updating the minimum distance when a shorter path to a vertex is found.

Visited Set:The algorithm maintains a set of visited vertices to ensure that each vertex is visited only once.

Algorithm:

1)extract\_min(distances, visited) function:

Purpose: It identifies the unvisited node with the minimum distance from the source node.

Input: distances - dictionary storing the current shortest distances to nodes, visited - dictionary to track visited nodes.

Process:Initializes min\_distance as infinity and min\_node as None.Iterates through the nodes in distances.Finds the unvisited node with the minimum distance from the source node.Returns the node with the minimum distance.

2)dijkstra(graph, source) function:

Purpose: Calculates the shortest path from the source node to all other nodes in the graph.

Input: graph - dictionary representing the weighted graph, source - the source node.

Process:Initializes distances to infinity for all nodes except the source, which is set to 0.Initializes visited dictionary to track visited nodes.Iterates through the graph nodes (once for each node).Calls extract\_min to find the unvisited node with the minimum distance from the source.Marks the found node as visited and updates the distances to its neighbors if a shorter path is discovered.

Code:

def extract\_min(distances, visited):

min\_distance = float('inf')

min\_node = None

for node in distances:

if not visited[node] and distances[node] < min\_distance:

min\_distance = distances[node]

min\_node = node

return min\_node

def dijkstra(graph, source):

distances = {node: float('inf') for node in graph}

distances[source] = 0

visited = {node: False for node in graph}

for \_ in range(len(graph)):

current\_node = extract\_min(distances, visited)

visited[current\_node] = True

for neighbor, weight in graph[current\_node].items():

if not visited[neighbor]:

distances[neighbor] = min(distances[neighbor], distances[current\_node] + weight)

return distances

# Example usage:

graph = {

'A': {'B': 3, 'C': 1},

'B': {'A': 3, 'C': 7, 'D': 5},

'C': {'A': 1, 'B': 7, 'D': 2},

'D': {'B': 5, 'C': 2}

}

source\_node = 'A'

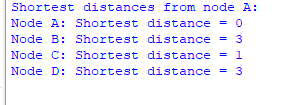
shortest\_distances = dijkstra(graph, source\_node)

print(f"Shortest distances from node {source\_node}:")

for node, distance in shortest\_distances.items():

print(f"Node {node}: Shortest distance = {distance}")

Output:



Conclusion: We Have Successfully implemented the Dijkshtra’s algorithm.

Practical:09

Aim: Implement Chinese reminder theorem to a constraint satisfaction problem.Analyze its complexity.

Theory:

The Chinese Remainder Theorem (CRT) is a mathematical theorem that provides a way to solve a system of simultaneous congruences. In the context of a Constraint Satisfaction Problem (CSP), the Chinese Remainder Theorem offers a method to efficiently solve a set of modular arithmetic equations, often related to combinatorial problems.In the context of a Constraint Satisfaction Problem, the Chinese Remainder Theorem can be applied to solve problems where a solution must satisfy multiple constraints or equations. These problems often involve discrete variables subject to specific constraints, and they can be represented as a system of modular arithmetic equations.

Application to CSP:

Discrete Variables: CSPs often involve discrete variables that need to satisfy certain constraints.

Constraints as Congruences: The constraints in a CSP can be represented as modular congruences or equations.

Solving Simultaneous Constraints: By applying the CRT, a CSP solver can efficiently find a solution that satisfies multiple constraints by computing the unique solution modulo the product of pairwise coprime moduli.

Complexity Analysis:

CRT Complexity:The complexity of solving the Chinese Remainder Theorem for k congruences can be analyzed as follows:

Time Complexity:The process of finding the solution to k congruences using the CRT involves multiple computations. The time complexity of the CRT is generally O(klogM), where M is the product of the moduli.The algorithm typically involves solving k simultaneous congruences, and the dominant factor in the time complexity is usually the computation of the modular inverses and the final reconstruction of the solution.

Space Complexity:The space complexity for solving the CRT is O(k), as it involves storing k congruences.

Algorithm:

1)chinese\_remainder\_theorem Function:This function aims to solve a system of congruences by applying the Chinese Remainder Theorem to find a solution that satisfies the given set of constraints.

Convert Moduli to SymPy's Integer Type:The function first converts the moduli from the input congruences to SymPy's Integer type. This is to ensure the gcd method is available to check pairwise coprimality.

Pairwise Coprimality Check:It checks if the moduli are pairwise coprime by iterating through all pairs of moduli.

The gcd() function is used to find the greatest common divisor of two moduli. If any pair shares a common divisor other than 1, it raises a ValueError.

Solve Congruences Using solve\_congruence:

It uses SymPy's solve\_congruence function to solve the system of congruences provided as input.

This function computes the solution to the congruences using different algorithms (e.g., Dixon's Random Squares, Extended Euclidean Algorithm).

Output:The code computes and prints the solution that satisfies the given constraints to the Constraint Satisfaction Problem.

Code:

from sympy import Integer

from sympy.ntheory.modular import solve\_congruence

def chinese\_remainder\_theorem(congruences):

moduli = [Integer(mod) for (\_, mod) in congruences] # Convert moduli to Integer type

# Ensure moduli are pairwise coprime

for i in range(len(moduli)):

for j in range(i + 1, len(moduli)):

if moduli[i].gcd(moduli[j]) != 1:

raise ValueError("Moduli are not pairwise coprime")

# Solve congruences using SymPy's solve\_congruence

solution = solve\_congruence(\*congruences)

return solution

# Example CSP represented as congruences

congruences = [

(2, 3), # x ≡ 2 (mod 3)

(3, 5), # x ≡ 3 (mod 5)

(2, 7) # x ≡ 2 (mod 7)

]

solution = chinese\_remainder\_theorem(congruences)

print("Solution to the Constraint Satisfaction Problem:")

print(solution)

Output:



Conclusion: We Have successfully implemented the chinese reminder to a constraint satisfaction problem.