**DAA Practicals**

**By Shivam Tomar**

**20211455**

**B.Sc. (H) Computer Science**

**For the algorithms at S.No 1 to 3 test run the algorithm on 100 different inputs of sizes varying from 30 to 1000. Count the number of comparisons and draw the graph. Compare it with a graph of nlogn.**

**1.**

**(a). Implement Insertion Sort (The program should report the number of comparisons)**

#

import random

import time

import matplotlib.pyplot as plt

def insertion\_sort(arr):

comparisons = 0

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

key = arr[i]

j = i - 1

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

arr[j + 1] = arr[j]

j -= 1

comparisons += 1

arr[j + 1] = key

return arr, comparisons

def test\_insertion\_sort():

num\_tests = 100

min\_size = 30

max\_size = 100

total\_comparisons = [0] \* (max\_size - min\_size + 1)

for i in range(num\_tests):

size = random.randint(min\_size, max\_size)

arr = [random.randint(1, 1000) for \_ in range(size)]

\_, num\_comparisons = insertion\_sort(arr)

total\_comparisons[size - min\_size] += num\_comparisons

avg\_comparisons = [count / num\_tests for count in total\_comparisons]

return avg\_comparisons

def nlogn(size):

return size \* (size - 1) / 2 \* (2 / size)

sizes = list(range(30, 101))

avg\_comparisons = test\_insertion\_sort()

nlogn\_values = [nlogn(size) for size in sizes]

plt.plot(sizes, avg\_comparisons, label="Insertion Sort")

plt.plot(sizes, nlogn\_values, label="nlogn")

plt.xlabel("Input size")

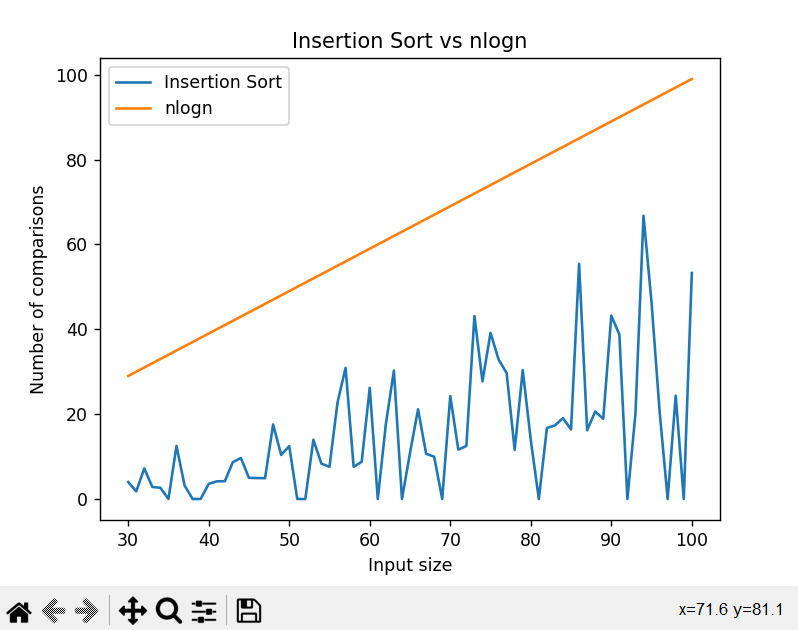
plt.ylabel("Number of comparisons")

plt.title("Insertion Sort vs nlogn")

plt.legend()

plt.show()

**Output:-**

****

**(b). Implement merge Sort (The program should report the number of comparisons)**.

#

import random

import time

import matplotlib.pyplot as plt

import numpy as np

def merge\_sort(arr):

if len(arr) <= 1:

return arr, 0

mid = len(arr) // 2

left, left\_comparisons = merge\_sort(arr[:mid])

right, right\_comparisons = merge\_sort(arr[mid:])

merged, merge\_comparisons = merge(left, right)

return merged, left\_comparisons + right\_comparisons + merge\_comparisons

def merge(left, right):

i = j = comparisons = 0

merged = []

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

if left[i] <= right[j]:

merged.append(left[i])

i += 1

else:

merged.append(right[j])

j += 1

comparisons += 1

while i < len(left):

merged.append(left[i])

i += 1

while j < len(right):

merged.append(right[j])

j += 1

return merged, comparisons

def test\_merge\_sort():

num\_tests = 100

min\_size = 30

max\_size = 100

total\_comparisons = [0] \* (max\_size - min\_size + 1)

for i in range(num\_tests):

size = random.randint(min\_size, max\_size)

arr = [random.randint(1, 1000) for \_ in range(size)]

\_, num\_comparisons = merge\_sort(arr)

total\_comparisons[size - min\_size] += num\_comparisons

avg\_comparisons = [count / num\_tests for count in total\_comparisons]

return avg\_comparisons

# Test merge sort and report the average number of comparisons for each input size

avg\_comparisons = test\_merge\_sort()

for i, count in enumerate(avg\_comparisons):

print(f"Input size {i+30}: Average number of comparisons = {count:.2f}")

# Generate graph of the average number of comparisons made by merge sort

plt.plot(range(30, 101), avg\_comparisons, label="Merge sort")

# Generate graph of expected number of comparisons (proportional to n log n)

nlogn = [n \* np.log2(n) for n in range(30, 101)]

plt.plot(range(30, 101), nlogn, label="n log n")

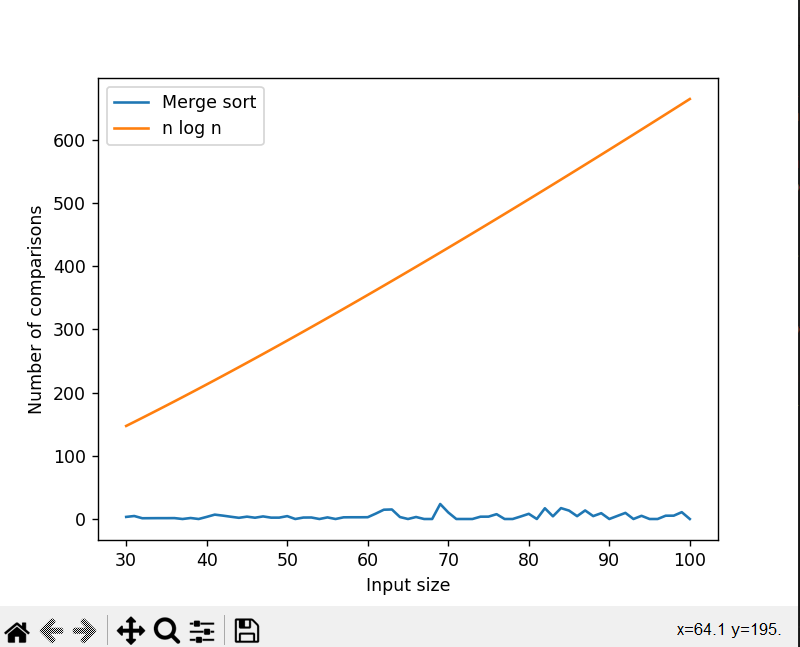
plt.xlabel("Input size")

plt.ylabel("Number of comparisons")

plt.legend()

plt.show()

**Output:-**

****

**2. Heap Sort(The Program should report the number of comparison)**

import random

import matplotlib.pyplot as plt

import numpy as np

def partition(arr, low, high):

pivot = arr[high]

i = low - 1

comparisons = 0

for j in range(low, high):

comparisons += 1

if arr[j] <= pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

comparisons += 1

arr[i+1], arr[high] = arr[high], arr[i+1]

comparisons += 1

return i+1, comparisons

def randomized\_partition(arr, low, high):

i = random.randint(low, high)

arr[i], arr[high] = arr[high], arr[i]

return partition(arr, low, high)

def quick\_sort(arr, low, high):

comparisons = 0

if low < high:

pi, comp1 = randomized\_partition(arr, low, high)

comparisons += comp1

comp2 = quick\_sort(arr, low, pi-1)

comparisons += comp2

comp3 = quick\_sort(arr, pi+1, high)

comparisons += comp3

return comparisons

# Generate arrays of varying sizes and sort them with Quick Sort

n\_values = np.arange(30, 1001, 10)

quick\_sort\_comparisons = []

for n in n\_values:

my\_arr = [random.randint(1, 1000) for i in range(n)]

num\_comparisons = quick\_sort(my\_arr, 0, n-1)

quick\_sort\_comparisons.append(num\_comparisons)

# Generate a simple line graph of the number of comparisons made by Quick Sort versus n

fig, ax = plt.subplots()

ax.plot(n\_values, quick\_sort\_comparisons, label="Quick Sort")

# Generate a graph of nlogn for comparison

nlogn\_values = n\_values \* np.log(n\_values)

ax.plot(n\_values, nlogn\_values, label="nlogn")

# Add axis labels and a legend

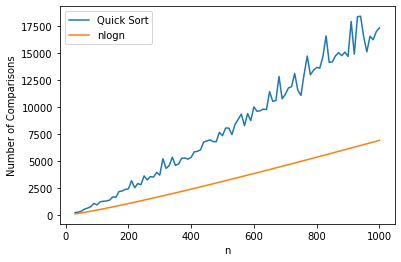
ax.set\_xlabel("n")

ax.set\_ylabel("Number of Comparisons")

ax.legend()

# Display the graph

plt.show()



**3.** **Implement Randomized Quick sort (The program should report the number of comparisons)**

#

import random

import math

import matplotlib.pyplot as plt

def randomized\_quicksort(arr, lo, hi):

if lo < hi:

comparisons = 0

p, comparisons = randomized\_partition(arr, lo, hi)

comparisons += randomized\_quicksort(arr, lo, p-1)

comparisons += randomized\_quicksort(arr, p+1, hi)

return comparisons

else:

return 0

def randomized\_partition(arr, lo, hi):

i = random.randint(lo, hi)

arr[i], arr[hi] = arr[hi], arr[i]

return partition(arr, lo, hi)

def partition(arr, lo, hi):

pivot = arr[hi]

i = lo - 1

comparisons = 0

for j in range(lo, hi):

comparisons += 1

if arr[j] <= pivot:

i += 1

arr[i], arr[j] = arr[j], arr[i]

arr[i+1], arr[hi] = arr[hi], arr[i+1]

return i+1, comparisons

def generate\_random\_array(size):

return [random.randint(-1000, 1000) for \_ in range(size)]

def theoretical\_comparisons(n):

return n \* math.log(n, 2)

def plot\_results(sizes, actual\_comparisons):

theoretical = [theoretical\_comparisons(size) for size in sizes]

plt.plot(sizes, actual\_comparisons, label="Randomized Quicksort")

plt.plot(sizes, theoretical, label="Theoretical")

plt.xlabel("Input Size (n)")

plt.ylabel("Number of Comparisons")

plt.legend()

plt.show()

sizes = list(range(30, 101))

actual\_comparisons = []

for size in sizes:

total\_comparisons = 0

for i in range(100):

arr = generate\_random\_array(size)

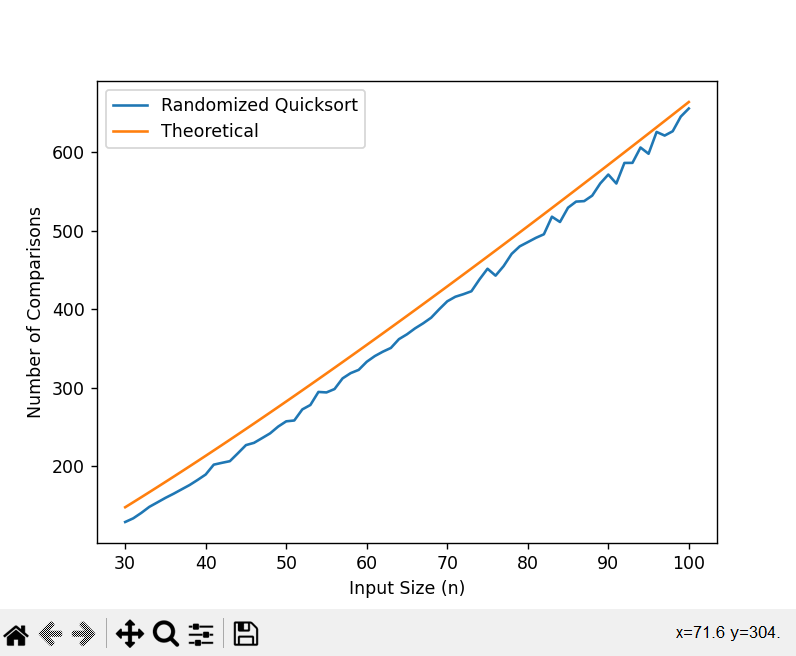
comparisons = randomized\_quicksort(arr, 0, len(arr)-1)

total\_comparisons += comparisons

actual\_comparisons.append(total\_comparisons / 100)

plot\_results(sizes, actual\_comparisons)

**Output:-**



**4. Implement Radix Sort**

#

import random

def radix\_sort(arr):

max\_num = max(arr)

exp = 1

while max\_num // exp > 0:

counting\_sort(arr, exp)

exp \*= 10

def counting\_sort(arr, exp):

n = len(arr)

output = [0] \* n

count = [0] \* 10

for i in range(n):

index = arr[i] // exp

count[index % 10] += 1

for i in range(1, 10):

count[i] += count[i - 1]

for i in range(n - 1, -1, -1):

index = arr[i] // exp

output[count[index % 10] - 1] = arr[i]

count[index % 10] -= 1

for i in range(n):

arr[i] = output[i]

# generate an array of size 30 with random integers

arr = [random.randint(0, 100) for \_ in range(30)]

# print the unsorted array

print("Unsorted array:", arr)

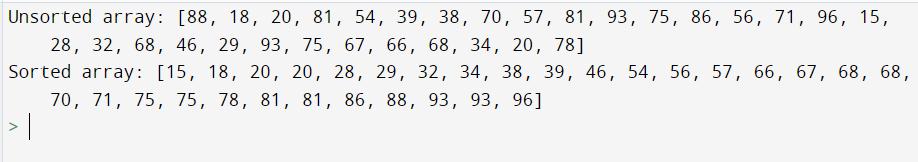
# sort the array using radix sort

radix\_sort(arr)

# print the sorted array

print("Sorted array:", arr)

**Output:-**



**5. Implement Bucket sort**

#

def bucket\_sort(arr):

bucket = []

for i in range(len(arr)):

bucket.append([])

for j in arr:

index = int(10 \* j)

bucket[index].append(j)

for i in range(len(arr)):

bucket[i] = sorted(bucket[i])

k = 0

for i in range(len(arr)):

for j in range(len(bucket[i])):

arr[k] = bucket[i][j]

k += 1

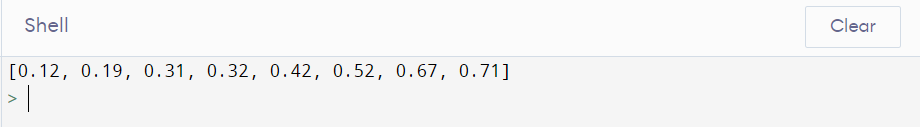
return arr

arr = [0.42, 0.32, 0.12, 0.52, 0.67, 0.31, 0.19, 0.71]

sorted\_arr = bucket\_sort(arr)

print(sorted\_arr)

**Output:-**

****

**6. Implement Randomized Select**

import random

def randomized\_select(arr, p, r, i):

if p == r:

return arr[p]

q = randomized\_partition(arr, p, r)

k = q - p + 1

if i == k:

return arr[q]

elif i < k:

return randomized\_select(arr, p, q - 1, i)

else:

return randomized\_select(arr, q + 1, r, i - k)

def randomized\_partition(arr, p, r):

i = random.randint(p, r)

arr[r], arr[i] = arr[i], arr[r]

return partition(arr, p, r)

def partition(arr, p, r):

x = arr[r]

i = p - 1

for j in range(p, r):

if arr[j] <= x:

i += 1

arr[i], arr[j] = arr[j], arr[i]

arr[i+1], arr[r] = arr[r], arr[i+1]

return i + 1

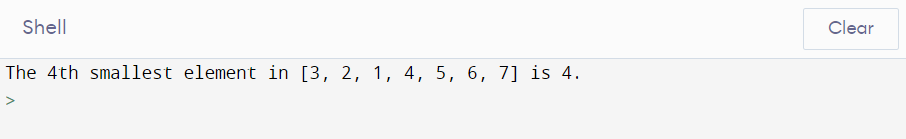
arr = [3, 7, 2, 1, 4, 6, 5]

i = 4

result = randomized\_select(arr, 0, len(arr)-1, i)

print(f"The {i}th smallest element in {arr} is {result}.")

**Output:-**



**7. Implement Breadth-First Search in Graph**

#

from collections import deque

# Define the graph

graph = {

'A': ['B', 'C', 'D'],

'B': ['E', 'F'],

'C': ['G'],

'D': ['H'],

'E': [],

'F': ['I', 'J'],

'G': ['K'],

'H': ['L'],

'I': [],

'J': ['M'],

'K': [],

'L': [],

'M': ['N'],

'N': []

}

def bfs(graph, start):

visited = set() # Create an empty set to keep track of visited nodes

queue = deque([start]) # Create a queue and add the starting node

visited.add(start) # Mark the starting node as visited

while queue: # Loop until the queue is empty

node = queue.popleft() # Get the next node from the queue

print(node) # Do something with the node (e.g. print it)

for neighbor in graph[node]: # Iterate over the neighbors of the node

if neighbor not in visited: # If the neighbor hasn't been visited yet

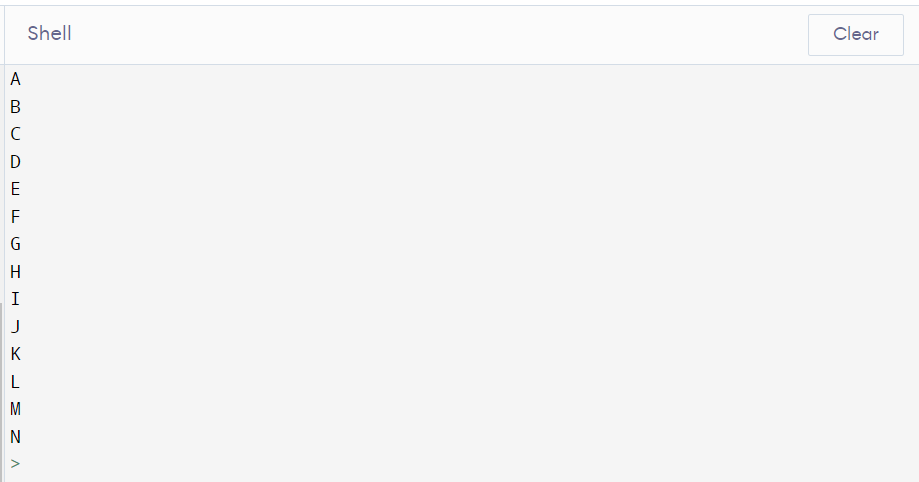
visited.add(neighbor) # Mark it as visited

queue.append(neighbor) # Add it to the queue

# Call the BFS function with the graph and starting node

bfs(graph, 'A')

**Output:-**



**8. Implement Depth-First Search in Graph**

#

# Define the graph

graph = {

'A': ['B', 'C', 'D'],

'B': ['E', 'F'],

'C': ['G'],

'D': ['H'],

'E': [],

'F': ['I', 'J'],

'G': ['K'],

'H': ['L'],

'I': [],

'J': ['M'],

'K': [],

'L': [],

'M': ['N'],

'N': []

}

def dfs(graph, start, visited=set()):

visited.add(start) # Mark the starting node as visited

print(start) # Do something with the node (e.g. print it)

for neighbor in graph[start]: # Iterate over the neighbors of the node

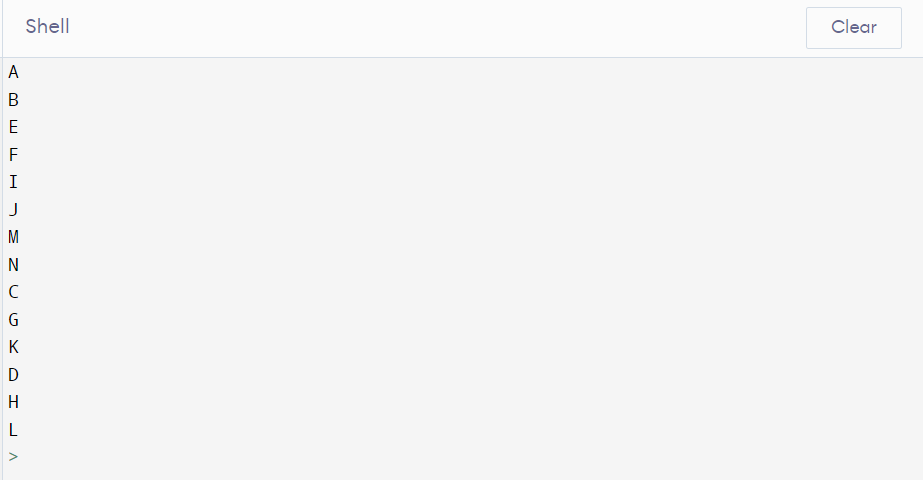
if neighbor not in visited: # If the neighbor hasn't been visited yet

dfs(graph, neighbor, visited) # Recursively call DFS on the neighbor

# Call the DFS function with the graph and starting node

dfs(graph, 'A')

**Output:-**



**9. Write a program to determine the minimum spanning tree of a graph using both prisms and kruskals algorithm.**

#PRISMS

import heapq

# Define the graph

graph = {

'A': [('B', 2), ('C', 3), ('D', 4)],

'B': [('A', 2), ('E', 3), ('F', 4)],

'C': [('A', 3), ('G', 4)],

'D': [('A', 4), ('H', 5)],

'E': [('B', 3)],

'F': [('B', 4), ('I', 5), ('J', 6)],

'G': [('C', 4), ('K', 5)],

'H': [('D', 5), ('L', 6)],

'I': [('F', 5)],

'J': [('F', 6), ('M', 7)],

'K': [('G', 5)],

'L': [('H', 6)],

'M': [('J', 7), ('N', 8)],

'N': [('M', 8)]

}

def prim(graph, start):

# Initialize the heap with the edges connected to the starting node

heap = [(weight, start, neighbor) for neighbor, weight in graph[start]]

heapq.heapify(heap)

visited = set() # Create an empty set to keep track of visited nodes

visited.add(start) # Mark the starting node as visited

mst = [] # Create an empty list to store the edges in the MST

while heap: # Loop until the heap is empty

weight, node1, node2 = heapq.heappop(heap) # Get the edge with the smallest weight

if node2 not in visited: # If the second node is not visited

visited.add(node2) # Mark it as visited

mst.append((node1, node2, weight)) # Add the edge to the MST

# Add the edges connected to the new node to the heap

for neighbor, weight in graph[node2]:

if neighbor not in visited:

heapq.heappush(heap, (weight, node2, neighbor))

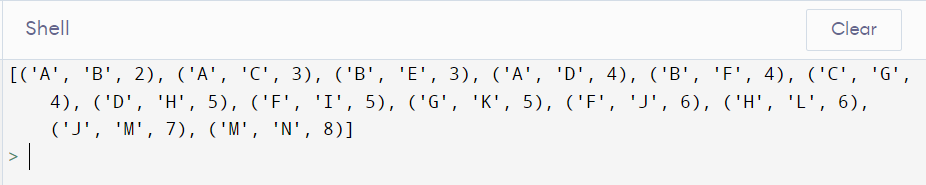
return mst

# Call the prim function with the graph and starting node

mst = prim(graph, 'A')

print(mst)

**Output:-**

****

#KRUSKALS

# Define the graph

graph = {

'A': [('B', 2), ('C', 3), ('D', 4)],

'B': [('A', 2), ('E', 3), ('F', 4)],

'C': [('A', 3), ('G', 4)],

'D': [('A', 4), ('H', 5)],

'E': [('B', 3)],

'F': [('B', 4), ('I', 5), ('J', 6)],

'G': [('C', 4), ('K', 5)],

'H': [('D', 5), ('L', 6)],

'I': [('F', 5)],

'J': [('F', 6), ('M', 7)],

'K': [('G', 5)],

'L': [('H', 6)],

'M': [('J', 7), ('N', 8)],

'N': [('M', 8)]

}

def find(parent, node):

# Find the root of the node's component using path compression

if parent[node] != node:

parent[node] = find(parent, parent[node])

return parent[node]

def union(parent, rank, node1, node2):

# Merge the components of the two nodes using union by rank

root1 = find(parent, node1)

root2 = find(parent, node2)

if rank[root1] < rank[root2]:

parent[root1] = root2

elif rank[root1] > rank[root2]:

parent[root2] = root1

else:

parent[root2] = root1

rank[root1] += 1

def kruskal(graph):

edges = [] # Create an empty list to store the edges

for node in graph:

for neighbor, weight in graph[node]:

edges.append((weight, node, neighbor)) # Add each edge to the list

edges.sort() # Sort the edges by weight in ascending order

parent = {node: node for node in graph} # Initialize each node as a separate component

rank = {node: 0 for node in graph} # Initialize the rank of each component as 0

mst = [] # Create an empty list to store the edges in the MST

for weight, node1, node2 in edges: # Iterate over the edges in ascending order of weight

if find(parent, node1) != find(parent, node2): # If the nodes are not in the same component

union(parent, rank, node1, node2) # Merge the components

mst.append((node1, node2, weight)) # Add the edge to the MST

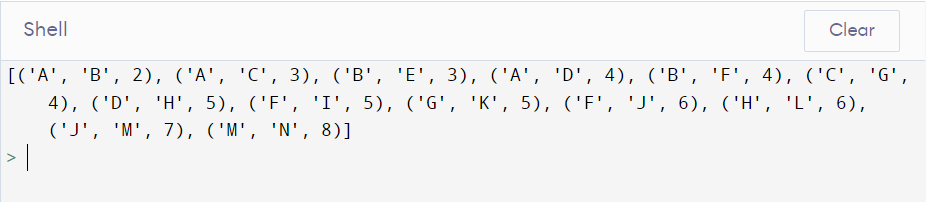
return mst

# Call the kruskal function with the graph

mst = kruskal(graph)

print(mst)

**Output:-**

****

**10. Write a program to solve the weighted interval scheduling problem**

#

def weighted\_interval\_scheduling(intervals, weights):

# Sort the intervals by end time

intervals = sorted(intervals, key=lambda x: x[1])

# Initialize the DP table with zeros

dp = [0] \* (len(intervals) + 1)

# Iterate over the intervals in reverse order

for i in range(len(intervals) - 1, -1, -1):

# Find the latest interval that doesn't overlap with the current one

j = i + 1

while j < len(intervals) and intervals[j][0] < intervals[i][1]:

j += 1

# Compute the maximum weight that can be obtained by considering the current interval

# and the maximum weight that can be obtained by skipping it

weight\_with\_interval = weights[i] + dp[j]

weight\_without\_interval = dp[i + 1]

# Choose the maximum weight and store it in the DP table

dp[i] = max(weight\_with\_interval, weight\_without\_interval)

# Return the maximum weight

return dp[0]

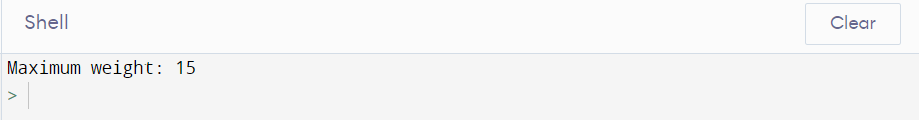
intervals = [(0, 3), (2, 4), (3, 5), (4, 7), (5, 8), (7, 9)]

weights = [2, 3, 4, 5, 6, 7]

max\_weight = weighted\_interval\_scheduling(intervals, weights)

print("Maximum weight:", max\_weight)

**Output:-**

****

**11. Write a program to solve the 0-1 knapsack problem**

def knapsack\_01(weights, values, capacity):

n = len(weights)

# Initialize the DP table with zeros

dp = [[0] \* (capacity + 1) for \_ in range(n + 1)]

# Fill the DP table

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

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

# If the item can fit in the knapsack, choose the maximum value between

# taking the item and not taking it

if weights[i-1] <= j:

dp[i][j] = max(dp[i-1][j], values[i-1] + dp[i-1][j-weights[i-1]])

# If the item cannot fit in the knapsack, do not take it

else:

dp[i][j] = dp[i-1][j]

# Return the maximum value

return dp[n][capacity]

weights = [1, 2, 3, 4, 5]

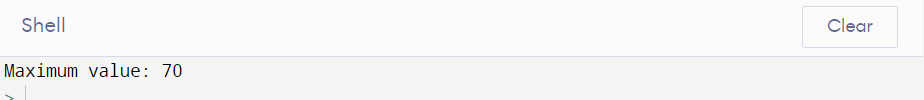
values = [10, 20, 30, 40, 50]

capacity = 7

max\_value = knapsack\_01(weights, values, capacity)

print("Maximum value:", max\_value)

**Output:-**

****