**LAB 06: Greedy Technique**

**III. Exercise**

**Warm up**

1. Activity Selection Problem

def activity\_selection(activities):

    activities.sort(key=lambda x: x[1])

    selected = [activities[0]]

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

        if activities[i][0] >= selected[-1][1]:

            selected.append(activities[i])

    return selected

activities = [(1, 4), (3, 5), (0, 6), (5, 7), (3, 8), (5, 9), (6, 10), (8, 11), (8, 12), (2, 13), (12, 14)]

print(activity\_selection(activities))

# Basic OP: addition in line 6

# Worst case: O(n log n) + O(n) = O(n log n)

# T(n)  = T(n/2) + O(n) = O(n log n)

# Time complexity: O(n log n)

1. Job Sequencing Problem with Deadlines

def job\_sequencing(tasks):

    # Sort the tasks in decreasing order of their profits

    tasks.sort(key=lambda x: x[2], reverse=True)

    # Get the maximum deadline

    max\_deadline = max(tasks, key=lambda x: x[1])[1]

    # Create a list of free slots

    free\_slots = [True] \* max\_deadline

    # Create a list to store the scheduled tasks

    scheduled\_tasks = [None] \* max\_deadline

    # Iterate through the tasks

    for task in tasks:

        # Find the latest free slot for the task

        for i in range(task[1] - 1, -1, -1):

            if free\_slots[i]:

                # Schedule the task

                scheduled\_tasks[i] = task

                # Mark the slot as occupied

                free\_slots[i] = False

                break

    # Return the list of scheduled tasks

    return scheduled\_tasks

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

    tasks = [(1, 9, 15), (2, 2, 2), (3, 5, 18), (4, 7, 1), (5, 4, 25), (6, 2, 20), (7, 5, 8), (8, 7, 10), (9, 4, 12), (10, 3, 5)]

    print(job\_sequencing(tasks))

# Basic OP: Comparision in line 16

# Worst case: O(n^2)

# T(n)  = T(n-1) + O(n) = O(n^2)

# Time complexity: O(n^2)

**Intermediate exercises**

1. Prim’s Algorithm

class Graph:

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

        self.V = vertices

        self.E = edges

        self.graph = []

        self.graph\_matrix = [[0 for column in range(vertices)] for row in range(vertices)]

    def add\_edge(self, u, v, w):

        self.graph.append([u, v, w])

        self.graph\_matrix[u][v] = w

        self.graph\_matrix[v][u] = w

    def printMST(self, parent):

        print("Edge \tWeight")

        for i in range(1, self.V):

            print(parent[i], "-", i, "\t", self.graph\_matrix[i][ parent[i] ])

    def minKey(self, key, mstSet):

        min = float('inf')

        for v in range(self.V):

            if key[v] < min and mstSet[v] == False:

                min = key[v]

                min\_index = v

        return min\_index

    def primMST(self):

        key = [float('inf')] \* self.V

        parent = [None] \* self.V

        key[0] = 0

        mstSet = [False] \* self.V

        parent[0] = -1

        for cout in range(self.V):

            u = self.minKey(key, mstSet)

            mstSet[u] = True

            for v in range(self.V):

                if self.graph\_matrix[u][v] > 0 and mstSet[v] == False and key[v] > self.graph\_matrix[u][v]:

                    key[v] = self.graph\_matrix[u][v]

                    parent[v] = u

        self.printMST(parent)

g = Graph(5, 7)

g.add\_edge(0, 1, 2)

g.add\_edge(0, 3, 6)

g.add\_edge(1, 2, 3)

g.add\_edge(1, 3, 8)

g.add\_edge(1, 4, 5)

g.add\_edge(2, 4, 7)

g.add\_edge(3, 4, 9)

g.primMST()

# Basic OP: assignment in line 37

# Worst case: O(n^2)

# T(n)  = T(n-1) + O(n) = O(n^2)

# Time complexity: O(n^2)

1. Kruskal’s Algorithm

class Graph:

    def \_\_init\_\_(self, vertices):

        self.V = vertices

        self.graph = []

    def add\_edge(self, u, v, w):

        self.graph.append([u, v, w])

    def find(self, parent, i):

        if parent[i] == i:

            return i

        return self.find(parent, parent[i])

    def union(self, parent, rank, x, y):

        xroot = self.find(parent, x)

        yroot = self.find(parent, y)

        if rank[xroot] < rank[yroot]:

            parent[xroot] = yroot

        elif rank[xroot] > rank[yroot]:

            parent[yroot] = xroot

        else:

            parent[yroot] = xroot

            rank[xroot] += 1

    def kruskal(self):

        result = []

        i = 0

        e = 0

        self.graph = sorted(self.graph, key=lambda item: item[2])

        parent = []

        rank = []

        for node in range(self.V):

            parent.append(node)

            rank.append(0)

        while e < self.V - 1:

            u, v, w = self.graph[i]

            i = i + 1

            x = self.find(parent, u)

            y = self.find(parent, v)

            if x != y:

                e = e + 1

                result.append([u, v, w])

                self.union(parent, rank, x, y)

        minimum\_cost = 0

        print("Edges in the constructed MST")

        for u, v, weight in result:

            minimum\_cost += weight

            print("%d -- %d == %d" % (u, v, weight))

        print("Minimum Spanning Tree" , minimum\_cost)

g = Graph(4)

g.add\_edge(0, 1, 10)

g.add\_edge(0, 2, 6)

g.add\_edge(0, 3, 5)

g.add\_edge(1, 3, 15)

g.add\_edge(2, 3, 4)

g.kruskal()

# Basic OP: addition in line 42

# Worst case: O(n^2)

# T(n)  = T(n-1) + O(n) = O(n^2)

# Time complexity: O(n^2)