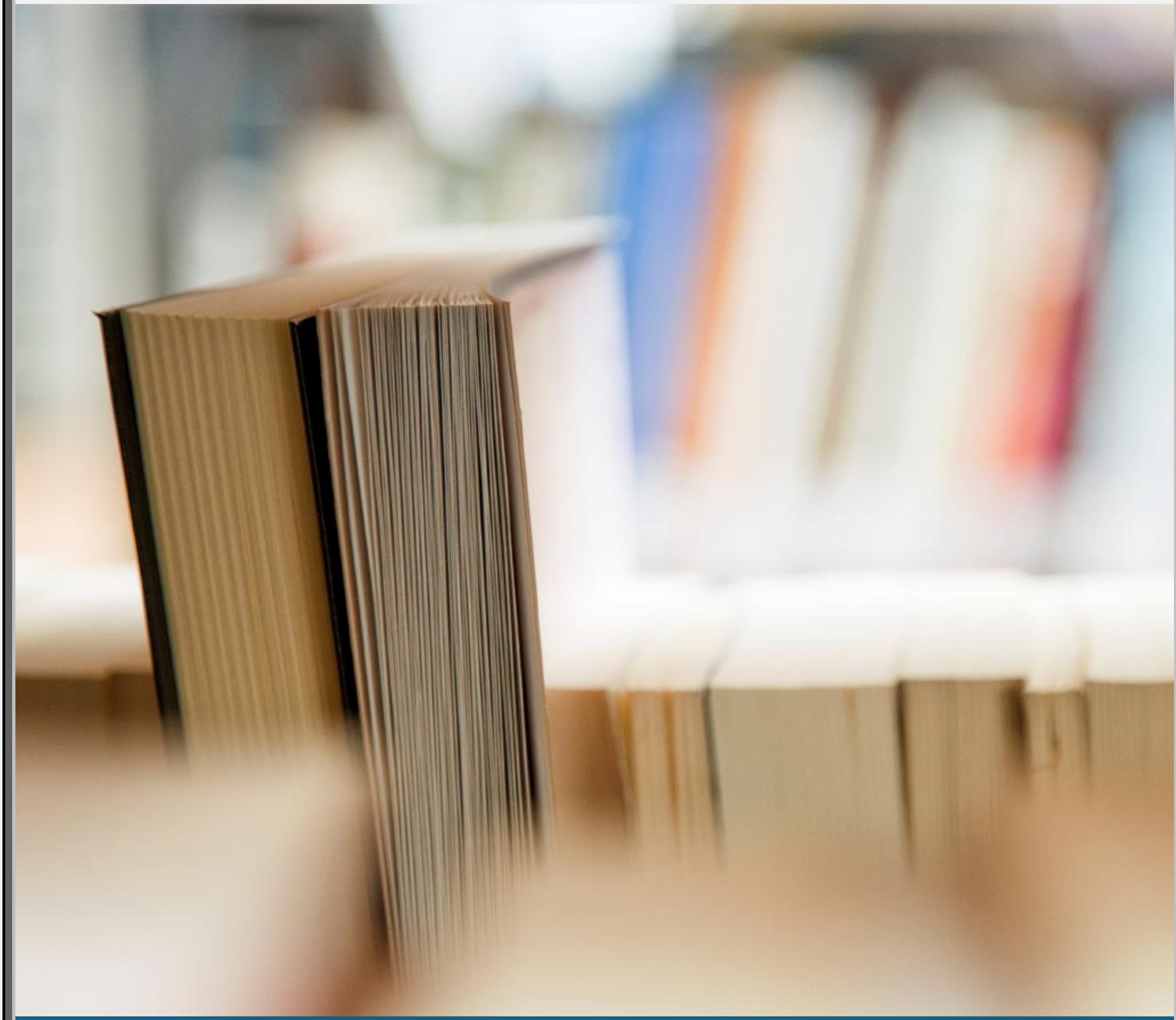

Algorithms in Python

vaibhaav waghmare 5/6/24 Python

Linear Search:

```
def linear_search(r, n):
    for i in range(len(r)):
        if r[i] == n:
            return i  # Element found, return index
    return -1  # Element not found

# Test the function
r = [10, 20, 80, 30, 60, 50, 110, 100, 130, 170]
n = 110

result = linear_search(r, n)
if result != -1:
    print("Element is present at index", result)
else:
    print("Element is not present in list")
```

Binary Search:

```
def binary_search(r, t):
    low = 0
    high = len(r) - 1

    while low <= high:
        mid = (high + low) // 2
        if r[mid] < t:
            low = mid + 1
        elif r[mid] > t:
            high = mid - 1
        else:
            return mid  # Element found, return index
```

```
    return -1  # Element not found

# Test the function
r = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
t = 70

result = binary_search(r, t)
if result != -1:
    print("Element is present at index", result)
else:
    print("Element is not present in list")
```

Max min Problem using Divide and Conquer:

```
def max_min(r, low, high):
    # If array has only one element
    if low == high:
        return (r[low], r[high])

    # If list has only two elements
    if high == low + 1:
        if r[low] > r[high]:
            return (r[low], r[high])
        else:
            return (r[high], r[low])

    # If list has more than 2 elements
    mid = (high + low) // 2
    max1, min1 = max_min(r, low, mid)
    max2, min2 = max_min(r, mid + 1, high)
```

```
    return (max(max1, max2), min(min1, min2))


# Test the function
r = [1000, 11, 445, 1, 330, 3000]
high = len(r) - 1
low = 0
maximum, minimum = max_min(r, low, high)
print(f"Minimum element is {minimum}")
print(f"Maximum element is {maximum}")
```

Knapsack problem using D&C:

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

# Test the function
val = [60, 100, 120]
wt = [10, 20, 30]
W = 50
n = len(val)

print(knapSack(W, wt, val, n))
```

Merge Sort:

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    # Find the middle point to divide the array into two halves
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]

    # Call merge_sort for first half
    left_half = merge_sort(left_half)

    # Call merge_sort for second half
    right_half = merge_sort(right_half)

    # Merge the two halves sorted
    return merge(left_half, right_half)

def merge(left_half, right_half):
    sorted_array = []
    left_index = 0
    right_index = 0

    # Copy data to temp arrays and then sort them
    while left_index < len(left_half) and right_index < len(right_half):
        if left_half[left_index] < right_half[right_index]:
            sorted_array.append(left_half[left_index])
            left_index += 1
        else:
            sorted_array.append(right_half[right_index])
            right_index += 1

    # Checking if any element was left
    while left_index < len(left_half):
        sorted_array.append(left_half[left_index])
        left_index += 1

    while right_index < len(right_half):
        sorted_array.append(right_half[right_index])
        right_index += 1

    return sorted_array

# Test the function
arr = [12, 11, 13, 5, 6, 7]
sorted_arr = merge_sort(arr)
print("Sorted array is:", sorted_arr)
```

Quick Sort:

```
def partition(arr, low, high):
    i = (low-1)  # index of smaller element
    pivot = arr[high]  # pivot

    for j in range(low, high):
        # If current element is smaller than or equal to pivot
        if arr[j] <= pivot:
            # increment index of smaller element
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)

def quickSort(arr, low, high):
    if len(arr) == 1:
        return arr
    if low < high:
        # pi is partitioning index, arr[p] is now at right place
        pi = partition(arr, low, high)

        # Separately sort elements before partition and after partition
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)

# Test the function
arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
quickSort(arr, 0, n-1)
print("Sorted array is:", arr)
```

Bubble Sort:

```
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n-1):
        # Last i elements are already in place
        for j in range(0, n-i-1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j+1] :
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

```
# Test the function
arr = [64, 34, 25, 12, 22, 11, 90]
bubbleSort(arr)
print("Sorted array is:", arr)
```

Matrix chain multiplication:

```
def MatrixChainOrder(p, n):
    # Create a table to store the cost of multiplying two matrices. Initialize it with zeros.
    m = [[0 for x in range(n)] for x in range(n)]

    # m[i, j] represents the cost of multiplying matrices from i to j. The cost is zero when
multiplying one matrix.
    for i in range(1, n):
        m[i][i] = 0

    # Compute the optimal multiplication order in a bottom-up manner.
    for L in range(2, n):
        for i in range(1, n - L + 1):
            j = i + L - 1
            m[i][j] = float('inf')
            for k in range(i, j):
                q = m[i][k] + m[k + 1][j] + p[i - 1] * p[k] * p[j]
                if q < m[i][j]:
                    m[i][j] = q

    # Return the cost of multiplying the entire chain.
    return m[1][n - 1]

# Test the function
arr = [1, 2, 3, 4]
size = len(arr)

print("Minimum number of multiplications is " + str(MatrixChainOrder(arr, size)))
```

Find minimum cost spanning tree:

```
import sys # For INT_MAX

def primMST(graph):
    V = len(graph)
    # Array to store constructed MST
    key = [sys.maxsize] * V
    parent = [None] * V # Array to store constructed MST
    key[0] = 0   # Make key 0 so that this vertex is picked as first vertex
    mstSet = [False] * V

    parent[0] = -1  # First node is always the root

    for cout in range(V):
        # Pick the minimum key vertex from the set of vertices not yet included in MST
        u = minKey(key, mstSet)

        # Add the picked vertex to the MST Set
        mstSet[u] = True

        # Update key value and parent index of the adjacent vertices of the picked vertex.
Consider only those vertices which are not yet included in MST
        for v in range(V):
            # graph[u][v] is non zero only for adjacent vertices of m
            # mstSet[v] is false for vertices not yet included in MST
            # Update the key only if graph[u][v] is smaller than key[v]
            if graph[u][v] > 0 and mstSet[v] == False and key[v] > graph[u][v]:
                    key[v] = graph[u][v]
                    parent[v] = u

    # print the constructed MST
    printMST(parent, graph)


def minKey(key, mstSet):
    # Initilaize minimum value
    min = sys.maxsize
    min_index = 0
    V = len(key)
    for v in range(V):
        if mstSet[v] == False and key[v] < min:
            min = key[v]
            min_index = v

    return min_index

# A utility function to print the constructed MST stored in parent[]
def printMST(parent, graph):
    print("Edge \tWeight")
```

```
    for i in range(1, len(graph)):
        print(parent[i], "-", i, "\t", graph[i][ parent[i] ])

# Test the function
graph = [[0, 2, 0, 6, 0],
         [2, 0, 3, 8, 5],
         [0, 3, 0, 0, 7],
         [6, 8, 0, 0, 9],
         [0, 5, 7, 9, 0]]

primMST(graph)
```

Prim’s algo:

```
import sys  # For INT_MAX

def primMST(graph):
    V = len(graph)
    # Array to store constructed MST
    key = [sys.maxsize] * V
    parent = [None] * V  # Array to store constructed MST
    key[0] = 0  # Make key 0 so that this vertex is picked as first vertex
    mstSet = [False] * V

    parent[0] = -1  # First node is always the root

    for cout in range(V):
        # Pick the minimum key vertex from the set of vertices not yet included in MST
        u = minKey(key, mstSet)

        # Add the picked vertex to the MST Set
        mstSet[u] = True

        # Update key value and parent index of the adjacent vertices of the picked vertex.
Consider only those vertices which are not yet included in MST
        for v in range(V):
            # graph[u][v] is non zero only for adjacent vertices of m
            # mstSet[v] is false for vertices not yet included in MST
            # Update the key only if graph[u][v] is smaller than key[v]
            if graph[u][v] > 0 and mstSet[v] == False and key[v] > graph[u][v]:
                key[v] = graph[u][v]
                parent[v] = u

    # print the constructed MST
    printMST(parent, graph)


def minKey(key, mstSet):
```

```
    # Initialize minimum value
    min = sys.maxsize
    min_index = 0
    V = len(key)
    for v in range(V):
        if mstSet[v] == False and key[v] < min:
            min = key[v]
            min_index = v

    return min_index


# A utility function to print the constructed MST stored in parent[]
def printMST(parent, graph):
    print("Edge \tWeight")
    for i in range(1, len(graph)):
        print(parent[i], "-", i, "\t", graph[i][parent[i]])


# Test the function
graph = [[0, 2, 0, 6, 0],
         [2, 0, 3, 8, 5],
         [0, 3, 0, 0, 7],
         [6, 8, 0, 0, 9],
         [0, 5, 7, 9, 0]]

primMST(graph)
```

Kruskal Algo:

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = []

    def add_edge(self, u, v, w):
        self.graph.append([u, v, w])

    # Utility function to find set of an element i
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    # Function that does union of two sets of x and y
    def union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
```

```
        # Attach smaller rank tree under root of high rank tree
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    # Function to construct MST using Kruskal's algorithm
    def kruskalMST(self):
        result = []
        i, e = 0, 0

        # Step 1: Sort all the edges in non-decreasing order of their weight.
        self.graph = sorted(self.graph, key=lambda item: item[2])

        parent = []
        rank = []

        # Create V subsets with single elements
        for node in range(self.V):
            parent.append(node)
            rank.append(0)

        # Number of edges to be taken is equal to V-1
        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 including this edge does not cause cycle, include it in result and increment the
index of result for next edge
            if x != y:
                e = e + 1
                result.append([u, v, w])
                self.union(parent, rank, x, y)
            # Else discard the edge

        # print the contents of result[] to display the built MST
        print("Following are the edges in the constructed MST")
        for u, v, weight in result:
            print("%d -- %d == %d" % (u, v, weight))

# Test the function
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.kruskalMST()
```

Dijkstra Algo for finding single source shortest path:

```
import sys

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

    def printSolution(self, dist):
        print("Vertex \tDistance from Source")
        for node in range(self.V):
            print(node, "\t", dist[node])

    def minDistance(self, dist, sptSet):
        min = sys.maxsize
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v
        return min_index

    def dijkstra(self, src):
        dist = [sys.maxsize] * self.V
        dist[src] = 0
        sptSet = [False] * self.V
        for cout in range(self.V):
            u = self.minDistance(dist, sptSet)
            sptSet[u] = True
            for v in range(self.V):
                if (self.graph[u][v] > 0 and
                   sptSet[v] == False and
                   dist[v] > dist[u] + self.graph[u][v]):
                    dist[v] = dist[u] + self.graph[u][v]
        self.printSolution(dist)

# Test the function
g = Graph(9)
g.graph = [[0, 4, 0, 0, 0, 0, 0, 8, 0],
           [4, 0, 8, 0, 0, 0, 0, 11, 0],
           [0, 8, 0, 7, 0, 4, 0, 0, 2],
           [0, 0, 7, 0, 9, 14, 0, 0, 0],
```

```
        [0, 0, 0, 9, 0, 10, 0, 0, 0],
        [0, 0, 4, 14, 10, 0, 2, 0, 0],
        [0, 0, 0, 0, 0, 2, 0, 1, 6],
        [8, 11, 0, 0, 0, 0, 1, 0, 7],
        [0, 0, 2, 0, 0, 0, 6, 7, 0]
        ]

g.dijkstra(0)
```

Travel Salesman Problem:

```
from itertools import permutations

def TSP(graph):
    # store all vertex apart from source vertex
    vertex = []
    for i in range(len(graph)):
        if i != 0:
            vertex.append(i)

    # store minimum weight Hamiltonian Cycle
    min_path = max(int, 'inf')

    next_permutation=permutations(vertex)
    for i in next_permutation:

        # store current Path weight(cost)
        current_pathweight = 0

        # compute current path weight
        k = 0
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][0]

        # update minimum
        min_path = min(min_path, current_pathweight)

    return min_path

# Test the function
graph = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]
print(TSP(graph))
```