M.Rohith
3122 21 5001 085

Sri Sivasubramaniya Nadar College of Engineering, Kalavakkam - 603 110
(An Autonomous Institution, Affiliated to Anna University, Chennai)

UCS2403: DESIGN & ANALYSIS OF ALGORITHMS

Assignment 10 - CAT 2 Revision

1. Write a Python code to implement the following sorting algorithms:
   (a) Insertion sort
   (b) Bubble sort
   (c) Selection sort
   (d) Merge sort
   (e) Quick sort

2. Give a dynamic programming algorithm to solve the following problem. There is a flight of stairs with n steps in total. A girl climbing these stairs can take either one step at a time or two steps at a time. In how many number of ways can she climb the stairs? Implement the algorithm using Python.

3. Write a Python code to implement the following Greedy algorithms:
   - Dijkstra's algorithm
   - Prim's algorithm
   - Kruskal's algorithm

4. In Domino Solitaire, you have a grid with two rows and many columns. Each square in the grid contains an integer. You are given a supply of rectangular 2×1 tiles, each of which exactly covers two adjacent squares of the grid. You have to place tiles to cover all the squares in the grid such that each tile covers two squares and no pair of tiles overlap. The score for a tile is the difference between the bigger and the smaller number that are covered by the tile. The aim of the game is to maximize the sum of the scores of all the tiles. See the figure for an example of a grid, along with two different tilings and their scores. The score for Tiling 1 is

M.Rohith
3122 21 5001 085

$12 = (9 - 8) + (6 - 2) + (7 - 1) + (3 - 2)$ while the score for Tiling 2 is $6 = (8 - 6) + (9 - 7) + (3 - 2) + (2 - 1)$. There are other tilings possible for this grid, but you can check that Tiling 1 has the maximum score among all tilings. Your task is to write a Python code that reads the grid of numbers and computes the maximum score that can be achieved by any tiling of the grid. (Source: Indian National Olympiad in Informatics, 2008)

Ex-no: 9
Date: 07-06-2023

M.Rohith
3122 21 5001 085

1. **Insertion sort**

```python
# Python program for implementation of Insertion Sort

# Function to do insertion sort
def insertionSort(arr):

    if (n := len(arr)) <= 1:
        return
    for i in range(1, n):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >=0 and key < arr[j] :
                arr[j+1] = arr[j]
                j -= 1
        arr[j+1] = key


#sorting the array [12, 11, 13, 5, 6] using insertionSort
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
print(arr)
```

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-9> python InsertionSort.py
[5, 6, 11, 12, 13]
```

M.Rohith
3122 21 5001 085

Bubble sort

```
# Python program for implementation of Bubble Sort

def bubbleSort(arr):
    n = len(arr)
    # optimize code, so if the array is already sorted, it doesn't
need
    # to go through the entire process
    swapped = False
    # Traverse through all array elements
    for i in range(n-1):
        # range(n) also work but outer loop will
        # repeat one time more than needed.
        # 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]:
                swapped = True
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

        if not swapped:
            # if we haven't needed to make a single swap, we
            # can just exit the main loop.
            return


# Driver code to test above
arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print("Sorted array is:")
for i in range(len(arr)):
    print("% d" % arr[i], end=" ")
```

M.Rohith
3122 21 5001 085

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-9> python BubbleSort.py
Sorted array is:
 11  12  22  25  34  64  90
```

Selection sort

```
# Selection sort in Python
# time complexity O(n*n)
#sorting by finding min_index
def selectionSort(array, size):

    for ind in range(size):
        min_index = ind

        for j in range(ind + 1, size):
            # select the minimum element in every iteration
            if array[j] < array[min_index]:
                min_index = j
        # swapping the elements to sort the array
        (array[ind], array[min_index]) = (array[min_index],
array[ind])

arr = [-2, 45, 0, 11, -9,88,-97,-202,747]
size = len(arr)
selectionSort(arr, size)
print('The array after sorting in Ascending Order by selection sort
is:')
print(arr)
```

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-9> python SelectionSort.py
The array after sorting in Ascending Order by selection sort is:
[-202, -97, -9, -2, 0, 11, 45, 88, 747]
```

M.Rohith
3122 21 5001 085

Merge sort

```
# Python program for implementation of MergeSort

# Merges two subarrays of arr[].
# First subarray is arr[l..m]
# Second subarray is arr[m+1..r]


def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    # create temp arrays
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0    # Initial index of first subarray
    j = 0    # Initial index of second subarray
    k = l    # Initial index of merged subarray

    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[], if there
    # are any
    while i < n1:
```

M.Rohith
3122 21 5001 085

```
        arr[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[], if there
    # are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

# l is for left index and r is right index of the
# sub-array of arr to be sorted


def mergeSort(arr, l, r):
    if l < r:

        # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l+(r-l)//2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)


# Driver code to test above
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print("Given array is")
for i in range(n):
    print("%d" % arr[i],end=" ")

mergeSort(arr, 0, n-1)
print("\n\nSorted array is")
for i in range(n):
    print("%d" % arr[i],end=" ")
```

M.Rohith
3122 21 5001 085

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-9> python MergeSort.py
Given array is
12 11 13 5 6 7

Sorted array is
5 6 7 11 12 13
```

Quick sort

```
# Python program for implementation of Quicksort Sort

# This implementation utilizes pivot as the last element in the nums
list
# It has a pointer to keep track of the elements smaller than the
pivot
# At the very end of partition() function, the pointer is swapped
with the pivot
# to come up with a "sorted" nums relative to the pivot


# Function to find the partition position
def partition(array, low, high):

    # choose the rightmost element as pivot
    pivot = array[high]

    # pointer for greater element
    i = low - 1

    # traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:

            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1

            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])
```

M.Rohith
3122 21 5001 085

```
    # Return the position from where partition is done
    return i + 1

# function to perform quicksort


def quickSort(array, low, high):
    if low < high:

        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)

        # Recursive call on the left of pivot
        quickSort(array, low, pi - 1)

        # Recursive call on the right of pivot
        quickSort(array, pi + 1, high)


data = [1, 7, 4, 1, 10, 9, -2]
print("Unsorted Array")
print(data)

size = len(data)

quickSort(data, 0, size - 1)

print('Sorted Array in Ascending Order:')
print(data)
```

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-9> python QuickSort.py
Unsorted Array
[1, 7, 4, 1, 10, 9, -2]
Sorted Array in Ascending Order:
[-2, 1, 1, 4, 7, 9, 10]
```

M.Rohith
3122 21 5001 085

2. Dynamic Program

```python
def count_ways(n):
    if n <= 1:
        return 1

    # Create a list to store the number of ways to reach each step
    ways = [0] * (n + 1)

    # Base cases: There is only one way to reach the first and second steps
    ways[1] = 1
    ways[2] = 2

    # Iterate over the remaining steps, building up the solution
    for i in range(3, n + 1):
        # The number of ways to reach the current step is the sum of the ways
        # to reach the previous two steps
        ways[i] = ways[i - 1] + ways[i - 2]

    return ways[n]

# Test the implementation
n = 5
result = count_ways(n)
print(f"The number of ways to climb {n} stairs: {result}")
```

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-9> python DynamicProg.py
The number of ways to climb 5 stairs: 8
```

M.Rohith
3122 21 5001 085

3. Dijkstra Algorithm

```
# Python program for Dijkstra's single
# source shortest path algorithm. The program is
# for adjacency matrix representation of the graph
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 \t Distance from Source")
        for node in range(self.V):
            print(node, "\t\t", dist[node])

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minDistance(self, dist, sptSet):

        # Initialize minimum distance for next node
        min = 1e7

        # Search not nearest vertex not in the
        # shortest path tree
        for v in range(self.V):
            if dist[v] < min and sptSet[v] == False:
                min = dist[v]
                min_index = v

        return min_index

    # Function that implements Dijkstra's single source
    # shortest path algorithm for a graph represented
    # using adjacency matrix representation
    def dijkstra(self, src):

        dist = [1e7] * self.V
```

M.Rohith
3122 21 5001 085

```
        dist[src] = 0
        sptSet = [False] * self.V

        for cout in range(self.V):

            # Pick the minimum distance vertex from
            # the set of vertices not yet processed.
            # u is always equal to src in first iteration
            u = self.minDistance(dist, sptSet)

            # Put the minimum distance vertex in the
            # shortest path tree
            sptSet[u] = True

            # Update dist value of the adjacent vertices
            # of the picked vertex only if the current
            # distance is greater than new distance and
            # the vertex in not in the shortest path tree
            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)

# Driver program
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)
```

M.Rohith
3122 21 5001 085

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-9> python Dijkstra.py
Vertex   Distance from Source
0               0
1               4
2               12
3               19
4               21
5               11
6               9
7               8
8               14
```

Prims Algorithm

```python
# A Python3 program for
# Prim's Minimum Spanning Tree (MST) algorithm.
# The program is for adjacency matrix
# representation of the graph

# Library for INT_MAX
import sys


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

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

    # A utility function to find the vertex with
    # minimum distance value, from the set of vertices
    # not yet included in shortest path tree
    def minKey(self, key, mstSet):

        # Initialize min value
        min = sys.maxsize
```

M.Rohith
3122 21 5001 085

```
        for v in range(self.V):
            if key[v] < min and mstSet[v] == False:
                min = key[v]
                min_index = v

        return min_index

    # Function to construct and print MST for a graph
    # represented using adjacency matrix representation
    def primMST(self):

        # Key values used to pick minimum weight edge in cut
        key = [sys.maxsize] * self.V
        parent = [None] * self.V # Array to store constructed MST
        # Make key 0 so that this vertex is picked as first vertex
        key[0] = 0
        mstSet = [False] * self.V

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

        for cout in range(self.V):

            # Pick the minimum distance vertex from
            # the set of vertices not yet processed.
            # u is always equal to src in first iteration
            u = self.minKey(key, mstSet)

            # Put the minimum distance vertex in
            # the shortest path tree
            mstSet[u] = True

            # Update dist value of the adjacent vertices
            # of the picked vertex only if the current
            # distance is greater than new distance and
            # the vertex in not in the shortest path tree
            for v in range(self.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 self.graph[u][v] > 0 and mstSet[v] == False \
                and key[v] > self.graph[u][v]:
                    key[v] = self.graph[u][v]
                    parent[v] = u

        self.printMST(parent)


# Driver's code
if __name__ == '__main__':
    g = Graph(5)
    g.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]]

    g.primMST()
```

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-9> python PrimsAlgo.py
Edge    Weight
0 - 1    2
1 - 2    3
0 - 3    6
1 - 4    5
```

Kruskals Algorithm

```
# Python program for Kruskal's algorithm to find
# Minimum Spanning Tree of a given connected,
# undirected and weighted graph


# Class to represent a graph
class Graph:

    def __init__(self, vertices):
```

M.Rohith
3122 21 5001 085

```
        self.V = vertices
        self.graph = []

    # Function to add an edge to graph
    def addEdge(self, u, v, w):
        self.graph.append([u, v, w])

    # A utility function to find set of an element i
    # (truly uses path compression technique)
    def find(self, parent, i):
        if parent[i] != i:

            # Reassignment of node's parent
            # to root node as
            # path compression requires
            parent[i] = self.find(parent, parent[i])
        return parent[i]

    # A function that does union of two sets of x and y
    # (uses union by rank)
    def union(self, parent, rank, x, y):

        # Attach smaller rank tree under root of
        # high rank tree (Union by Rank)
        if rank[x] < rank[y]:
            parent[x] = y
        elif rank[x] > rank[y]:
            parent[y] = x

        # If ranks are same, then make one as root
        # and increment its rank by one
        else:
            parent[y] = x
            rank[x] += 1

    # The main function to construct MST
    # using Kruskal's algorithm
    def KruskalMST(self):

        # This will store the resultant MST
        result = []
```

M.Rohith
3122 21 5001 085

```
        # An index variable, used for sorted edges
        i = 0

        # An index variable, used for result[]
        e = 0

        # 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 less than to V-1
        while e < self.V - 1:

            # Pick the smallest edge and increment
            # the index for next iteration
            u, v, w = self.graph[i]
            i = i + 1
            x = self.find(parent, u)
            y = self.find(parent, v)

            # If including this edge doesn't
            # cause cycle, then 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

        minimumCost = 0
        print("Edges in the constructed MST")
```

```
        for u, v, weight in result:
            minimumCost += weight
            print("%d -- %d == %d" % (u, v, weight))
        print("Minimum Spanning Tree", minimumCost)


# Driver code
if __name__ == '__main__':
    g = Graph(4)
    g.addEdge(0, 1, 10)
    g.addEdge(0, 2, 6)
    g.addEdge(0, 3, 5)
    g.addEdge(1, 3, 15)
    g.addEdge(2, 3, 4)

    # Function call
    g.KruskalMST()
```

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-9> python KruskalsAlgo.py
Edges in the constructed MST
2 -- 3 == 4
0 -- 3 == 5
0 -- 1 == 10
Minimum Spanning Tree 19
```

4. **Domino Solitaire**

```
def max_score(grid):
    n = len(grid[0])

    # Create a list to store the maximum scores for each column
    max_scores = [0] * (n + 1)

    # Base cases
    max_scores[0] = 0
    max_scores[1] = abs(grid[0][0] - grid[1][0])

    # Iterate over the columns, calculating the maximum score for
each
    for i in range(2, n + 1):
```

M.Rohith
3122 21 5001 085

```
        max_scores[i] = max(max_scores[i - 1], max_scores[i - 2] +
abs(grid[0][i - 1] - grid[0][i - 2]), max_scores[i - 2] +
abs(grid[1][i - 1] - grid[1][i - 2]))

    return max_scores[n]

# Test the implementation
grid = [[8, 6, 2, 3],
        [9, 7, 1, 2]]
result = max_score(grid)
print("Maximum score:", result)
```

Output:

```
PS C:\Rohith\Backup\Desktop\SEM 4\Design and Analysis of Algorithms lab\Assignment-9> python DominoSolitaire.py
Maximum score: 7
```