**LAB 05: Decrease-and-Conquer**

**III. Exercise**

**Warn-up**

1. Recursive insertion sort.

def insertionSortRecursive(arr,n):

   # base case

   if n<=1:

      return

   # Sort

   insertionSortRecursive(arr,n-1)

   last = arr[n-1]

   j = n-2

   # move ahead

   while (j>=0 and arr[j]>last):

      arr[j+1] = arr[j]

      j = j-1

   arr[j+1]=last

# main

arr = [1,5,3,4,8,6,3,4,5]

n = len(arr)

insertionSortRecursive(arr, n)

print("Sorted array is:")

for i in range(n):

   print(arr[i],end=" ")

# Basic OP: addition in line 6

# Worst case: O(n^2)

# T(n) = T(n-1) + n

# Time complexity: O(n^2)

1. Recursive Exponentiation by squaring

# Recursive Exponentiation by squaring

def exp(x, n):

    if n == 0:

        return 1

    elif n == 1:

        return x

    elif n % 2 == 0:

        return exp(x, n // 2) \*\* 2

    else:

        return x \* exp(x, n // 2) \*\* 2

print(exp(2, 10))

# Basic OP: multiplication in line 10

# Worst case: O(n)

# T(n) = T(n/2) + 1

# Time complexity: O(n)

1. Recursive Euclid’s algorithm for greatest common divisor

# Recursive Euclid’s algorithm for greatest common divisor

def gcd(a, b):

    if b == 0:

        return a

    else:

        return gcd(b, a % b)

print(gcd(24, 16))

# Basic OP: modulo in line 7

# Worst case: O(n)

# T(n) = T(n/2) + 1

# Time complexity: O(n)

**Intermediate exercises**

1. Depth-first search (DFS)

class Graph:

    def \_\_init\_\_(self):

        self.vertices = {}

        self.edges = {}

        self.marked = {}

        self.count = 0

    def add\_vertex(self, vertex):

        if vertex not in self.vertices:

            self.vertices[vertex] = vertex

            self.marked[vertex] = 0

    def add\_edge(self, vertex1, vertex2):

        if vertex1 not in self.vertices:

            self.add\_vertex(vertex1)

        if vertex2 not in self.vertices:

            self.add\_vertex(vertex2)

        if vertex1 not in self.edges:

            self.edges[vertex1] = []

        if vertex2 not in self.edges:

            self.edges[vertex2] = []

        self.edges[vertex1].append(vertex2)

        self.edges[vertex2].append(vertex1)

    def dfs(self, vertex):

        self.marked[vertex] = 1

        self.count += 1

        print(vertex, self.count)

        for w in self.edges[vertex]:

            if self.marked[w] == 0:

                self.dfs(w)

    def dfs\_traversal(self):

        for v in self.vertices:

            if self.marked[v] == 0:

                self.dfs(v)

# Driver code

g = Graph()

g.add\_edge(1, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 4)

g.add\_edge(2, 5)

g.add\_edge(3, 6)

g.add\_edge(3, 7)

g.add\_edge(4, 8)

g.add\_edge(4, 9)

g.dfs\_traversal()

1. Breadth-first search (BFS)

# Breadth first search

class Graph:

    def \_\_init\_\_(self):

        self.vertices = {}

        self.edges = {}

        self.marked = {}

        self.count = 0

    def add\_vertex(self, vertex):

        if vertex not in self.vertices:

            self.vertices[vertex] = vertex

            self.marked[vertex] = 0

    def add\_edge(self, vertex1, vertex2):

        if vertex1 not in self.vertices:

            self.add\_vertex(vertex1)

        if vertex2 not in self.vertices:

            self.add\_vertex(vertex2)

        if vertex1 not in self.edges:

            self.edges[vertex1] = []

        if vertex2 not in self.edges:

            self.edges[vertex2] = []

        self.edges[vertex1].append(vertex2)

        self.edges[vertex2].append(vertex1)

    def bfs(self, vertex):

        queue = []

        self.marked[vertex] = 1

        self.count += 1

        print(vertex, self.count)

        queue.append(vertex)

        while len(queue) > 0:

            v = queue.pop(0)

            for w in self.edges[v]:

                if self.marked[w] == 0:

                    self.marked[w] = 1

                    self.count += 1

                    print(w, self.count)

                    queue.append(w)

    def bfs\_traversal(self):

        for v in self.vertices:

            if self.marked[v] == 0:

                self.bfs(v)

# Driver code

g = Graph()

g.add\_edge(1, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 4)

g.add\_edge(2, 5)

g.add\_edge(3, 6)

g.add\_edge(3, 7)

g.add\_edge(4, 8)

g.add\_edge(4, 9)

g.bfs\_traversal()

1. Topological Sorting

class Graph:

    def \_\_init\_\_(self):

        self.vertices = {}

        self.edges = {}

        self.marked = {}

        self.count = 0

        self.stack = []

    def add\_vertex(self, vertex):

        if vertex not in self.vertices:

            self.vertices[vertex] = vertex

            self.marked[vertex] = 0

    def add\_edge(self, vertex1, vertex2):

        if vertex1 not in self.vertices:

            self.add\_vertex(vertex1)

        if vertex2 not in self.vertices:

            self.add\_vertex(vertex2)

        if vertex1 not in self.edges:

            self.edges[vertex1] = []

        if vertex2 not in self.edges:

            self.edges[vertex2] = []

        self.edges[vertex1].append(vertex2)

        self.edges[vertex2].append(vertex1)

    def dfs(self, vertex):

        self.marked[vertex] = 1

        self.count += 1

        print(vertex, self.count)

        for w in self.edges[vertex]:

            if self.marked[w] == 0:

                self.dfs(w)

        self.stack.append(vertex)

    def dfs\_traversal(self):

        for v in self.vertices:

            if self.marked[v] == 0:

                self.dfs(v)

    def topological\_sort(self):

        self.dfs\_traversal()

        print(self.stack)

        self.stack.reverse()

        print(self.stack)

# Driver code

g = Graph()

g.add\_edge(1, 2)

g.add\_edge(1, 3)

g.add\_edge(2, 4)

g.add\_edge(2, 5)

g.add\_edge(3, 6)

g.add\_edge(3, 7)

g.add\_edge(4, 8)

g.add\_edge(4, 9)

g.topological\_sort()

**Upper-intermediate exercise**

1. Partition-based algorithm for selection problem

def LomutoPartition(A, l, r):

    p = A[l]

    s = l

    for i in range(l+1, r+1):

        if A[i] < p:

            s += 1

            A[i], A[s] = A[s], A[i]

    A[l], A[s] = A[s], A[l]

    return s

def quickselect(A, l, r, k):

    if l == r:

        return A[l]

    s = LomutoPartition(A, l, r)

    if k == s:

        return A[s]

    elif k < s:

        return quickselect(A, l, s-1, k)

    else:

        return quickselect(A, s+1, r, k)

# Driver code

A = [2, 8, 7, 1, 3, 5, 6, 4]

print(quickselect(A, 0, len(A)-1, 3))

print(quickselect(A, 0, len(A)-1, 5))

print(quickselect(A, 0, len(A)-1, 7))

# Basic OP: addition in line 20

# Worst case: O(n^2)

# T(n) = T(n/2) + n

# Time complexity: O(n^2)

1. Binary Search Tree Algorithms

1) Searching

2) Insertion of a new key

3) Finding the smallest (or the largest) key

class Node:

    def \_\_init\_\_(self, value):

        self.value = value

        self.left = None

        self.right = None

def search(root, key):

    if root is None or root.value == key:

        return root

    if root.value < key:

        return search(root.right, key)

    return search(root.left, key)

def insert(root, key):

    if root is None:

        return Node(key)

    if root.value == key:

        return root

    if root.value < key:

        root.right = insert(root.right, key)

    else:

        root.left = insert(root.left, key)

    return root

def findSmallest(root):

    if root is None:

        return None

    if root.left is None:

        return root

    return findSmallest(root.left)

# Driver code

root = None

root = insert(root, 8)

root = insert(root, 3)

root = insert(root, 10)

root = insert(root, 1)

root = insert(root, 6)

print("Smallest value in BST is", findSmallest(root).value)

print("Search for 10:", search(root, 10).value)