**LAB 04: Divide-and-Conquer**

**II. Exercise**

**Warn-up exercise**

1. Find an element in an ascending sorted array. If the element is found, return the index of the element in the array, otherwise, return -1.

# Find an element in an ascending sorted array. If the element is found, return the index of the element in the array, otherwise, return -1.

def find\_element( array, element ):

    if len(array) == 0:

        return -1

    else:

        mid = len(array) // 2

        if array[mid] == element:

            return mid

        elif array[mid] > element:

            return find\_element( array[:mid], element )

        else:

            return find\_element( array[mid+1:], element )

if \_\_name\_\_ == '\_\_main\_\_':

    print(find\_element([1,2,3,4,5,6,7,8,9], 5))

    print(find\_element([1,2,3,4,5,6,7,8,9], 10))

# Basic OP: commparision in line 9

# Worst case: O(log n)

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

# Time complexity: O(log n)

**Intermediate exercises**

1. Sort an array with Quicksort

def HoarePartition(A, l, r):

    x = A[l]

    i = l - 1

    j = r + 1

    while True:

        while True:

            j = j - 1

            if A[j] <= x:

                break

        while True:

            i = i + 1

            if A[i] >= x:

                break

        if i < j:

            A[i], A[j] = A[j], A[i]

        else:

            return j

def quicksort(A, l, r):

    if l < r:

        s = HoarePartition(A, l, r)

        quicksort(A, l, s)

        quicksort(A, s + 1, r)

if \_\_name\_\_ == '\_\_main\_\_':

    A = [3,6,5,2,7,1,9,8,4]

    quicksort(A, 0, len(A) - 1)

    print(A)

# Basic OP: commparision in line 21

# Worst case: O(n^2)

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

# Time complexity: O(n^2)

1. Height of tree

class Node:

    def \_\_init\_\_(self, value):

        self.value = value

        self.left = None

        self.right = None

def height(root):

    if root is None:

        return -1

    else:

        return 1 + max(height(root.left), height(root.right))

if \_\_name\_\_ == '\_\_main\_\_':

    root = Node(1)

    root.left = Node(2)

    root.right = Node(3)

    root.left.left = Node(4)

    root.left.right = Node(5)

    print(height(root))

# Basic OP: addition in line 11

# Worst case: O(n)

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

# Time complexity: O(n)

1. Pre-order, post-order, in-order

class Node:

    def \_\_init\_\_(self, value):

        self.value = value

        self.left = None

        self.right = None

def pre\_order(root):

    if root is None:

        return

    else:

        print(root.value)

        pre\_order(root.left)

        pre\_order(root.right)

def post\_order(root):

    if root is None:

        return

    else:

        post\_order(root.left)

        post\_order(root.right)

        print(root.value)

def in\_order(root):

    if root is None:

        return

    else:

        in\_order(root.left)

        print(root.value)

        in\_order(root.right)

# Driver code

root = Node(1)

root.left = Node(2)

root.right = Node(3)

root.left.left = Node(4)

root.left.right = Node(5)

root.right.left = Node(6)

root.right.right = Node(7)

print("Pre-order traversal of binary tree is")

pre\_order(root)

print("Post-order traversal of binary tree is")

post\_order(root)

print("In-order traversal of binary tree is")

in\_order(root)

# Basic OP: assignment in line 3

# Worst case: O(n)

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

# Time complexity: O(n)

**Upper-intermediate exercise**

1. Find the closest pair of points on a plane (pseudocode on the next page)

import math

def EuclideanDistance(p1, p2):

    return math.sqrt((p1.x - p2.x)\*\*2 + (p1.y - p2.y)\*\*2)

def BruteForceClosestPair(P):

    n = len(P)

    best = EuclideanDistance(P[0], P[1])

    for i in range(n-1):

        for j in range(i+1, n):

            d = EuclideanDistance(P[i], P[j])

            if d < best:

                best = d

                pair = (P[i], P[j])

    return (best, pair)

def StripClosestPair(P, Q, d):

    mid = len(P) // 2

    xmid = P[mid].x

    S = []

    for point in Q:

        if abs(point.x - xmid) < d:

            S.append(point)

    best = d

    n = len(S)

    for i in range(n-1):

        k = i + 1

        while k <= n-1 and S[k].y - S[i].y < best:

            d = EuclideanDistance(S[i], S[k])

            if d < best:

                best = d

                pair = (S[i], S[k])

            k += 1

    return (best, pair)

def EfficientClosestPair(P, Q):

    n = len(P)

    if n <= 3:

        return BruteForceClosestPair(P)

    else:

        mid = n // 2

        Qleft = []

        Qright = []

        for point in Q:

            if point in P[:mid]:

                Qleft.append(point)

            else:

                Qright.append(point)

        (dleft, pairleft) = EfficientClosestPair(P[:mid], Qleft)

        (dright, pairright) = EfficientClosestPair(P[mid:], Qright)

        if dleft < dright:

            d = dleft

            pair = pairleft

        else:

            d = dright

            pair = pairright

        (dstrip, pairstrip) = StripClosestPair(P, Q, d)

        if dstrip < d:

            d = dstrip

            pair = pairstrip

        return (d, pair)

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def \_\_repr\_\_(self):

        return '(%d, %d)' % (self.x, self.y)

    def \_\_eq\_\_(self, other):

        return self.x == other.x and self.y == other.y

# Driver code

P = [Point(2,3), Point(12,30), Point(40,50), Point(5,1), Point(12,10), Point(3,4)]

Q = [Point(2,3), Point(3,4), Point(5,1), Point(12,10), Point(12,30), Point(40,50)]

print(EfficientClosestPair(P, Q))

# Basic OP: addition in line 11

# Worst case: O(n^2)

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

# Time complexity: O(n^2)