**LAB 03: Brute-force Algorithms**

**IV. Exercise**

Warn-up problems

1. Exponential power

def Exponential\_power(a,n):

    if n==0:

        return 1

    else:

        return a\*Exponential\_power(a,n-1)

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

    print(Exponential\_power(2,3))

# Basic OP: multiplication on line 8

# Worst case: other cases

# Count: T(n) = n

# Time complexity: O(n)

1. Combination

def combination(n, k):

    if k == 0 or k == n:

        return 1

    else:

        return combination(n-1, k-1) + combination(n-1, k)

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

    print(combination(6, 2))

# Basic OP: multiplication on line 5

# Worst case: other cases

# Count: T(n) = n

# Time complexity: O(n)

1. Matrix multiplication

def multiply(A, B, n):

    C = [[0 for i in range(n)] for j in range(n)]

    for i in range(n):

        for j in range(n):

            for k in range(n):

                C[i][j] += A[i][k] \* B[k][j]

    return C

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

    A = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

    B = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

    print(multiply(A, B, 3))

# Basic OP: addition on line 7

# Worst case: other cases

# Count: T(n) = n\*k

# Time complexity: O(n\*k)

More challenging problem

1. Nearest pair (closest pair)

class Point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def \_\_str\_\_(self):

        return "({}, {})".format(self.x, self.y)

def distance(p1, p2):

    return ((p1.x - p2.x)\*\*2 + (p1.y - p2.y)\*\*2)\*\*0.5

def closest\_pair(points):

    min\_dist = float("inf")

    for i in range(len(points)):

        for j in range(i+1, len(points)):

            dist = distance(points[i], points[j])

            if dist < min\_dist:

                min\_dist = dist

                p1 = points[i]

                p2 = points[j]

    return p1, p2, min\_dist

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

    points = [Point(1, 1), Point(2, 2), Point(3, 3), Point(4, 4), Point(5, 5)]

    p1, p2, min\_dist = closest\_pair(points)

    print("The closest pair is {} and {}, with distance {}".format(p1, p2, min\_dist))

1. Convex hull

class point:

    def \_\_init\_\_(self, x, y):

        self.x = x

        self.y = y

    def \_\_str\_\_(self):

        return "(" + str(self.x) + ", " + str(self.y) + ")"

def convex\_hull(points, n):

    if n < 3:

        return

    # Find the leftmost point

    l = 0

    for i in range(1, n):

        if points[i].x < points[l].x:

            l = i

    # Start from leftmost point, keep moving counterclockwise until reach the start point again

    p = l

    q = 0

    while True:

        # Search for a point 'q' such that orientation(p, i, q) is counterclockwise for all points 'i'

        for i in range(n):

            if orientation(points[p], points[i], points[q]) == 2:

                q = i

        # Add q to result as a next point of p in the convex hull

        print(points[q])

        # Now q is a previous point of p, so set p as q for next iteration, so that q is added to result

        p = q

        # While we don't come to first point

        if p == l:

            break

def orientation(p, q, r):

    val = (q.y - p.y) \* (r.x - q.x) - (q.x - p.x) \* (r.y - q.y)

    if val == 0:

        return 0 # colinear

    elif val > 0:

        return 1 # clockwise

    else:

        return 2 # counterclockwise

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

    points = [point(0, 3), point(2, 2), point(1, 1), point(2, 1), point(3, 0), point(0, 0), point(3, 3)]

    convex\_hull(points, len(points))

# Basic OP: assignment in line 23

# Worst case: when all the points are on the hull

# Time complexity: O(n\*h) where h is the number of points on the hull

1. Travelling Salesman Problem

import itertools

class Graph:

    def \_\_init\_\_(self, vertices):

        self.V = vertices

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

def BruteForceTSP(graph):

    vertex = []

    for i in range(graph.V):

        if i != 0:

            vertex.append(i)

    min\_path = 999999999

    next\_permutation = itertools.permutations(vertex)

    for i in next\_permutation:

        current\_pathweight = 0

        k = i[0]

        for j in i:

            current\_pathweight += graph.graph[k][j]

            k = j

        current\_pathweight += graph.graph[k][0]

        min\_path = min(min\_path, current\_pathweight)

    return min\_path

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

    v = 4

    g = Graph(v)

    g.graph = [[0, 10, 15, 20], [10, 0, 35, 25], [15, 35, 0, 30], [20, 25, 30, 0]]

    print(BruteForceTSP(g))

# Basic OP: addition on line 29

# Worst case: other cases

# Count: T(n) = n!

# Time complexity: O(n!)

1. Knapsack Problem

def knapsack(W, V, K):

    n = len(W)

    max\_value = 0

    for i in range(2 \*\* n):

        weight = 0

        value = 0

        for j in range(n):

            if (i >> j) & 1:

                weight += W[j]

                value += V[j]

        if weight <= K:

            max\_value = max(max\_value, value)

    return max\_value

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

    W = [10, 20, 30]

    V = [60, 100, 120]

    K = 50

    print(knapsack(W, V, K))

# Basic OP: addition on line 9

# Worst case: other cases

# T(n) = 2^n

# Time complexity: O(n^2)