1. Aggregate:

stack = []

arr = [8,3,1,9,8,7,3,6]

operations = 0

total\_cost = 0

unit = []

def push(element):

    global stack,operations,total\_cost,unit

    stack.append(element)

    operations+=1

    total\_cost+=1

    unit.append(1)

    print("Stack after pushing ",element,": ",stack)

def pop():

    global stack,operations,total\_cost,unit

    if len(stack)==0: return

    stack.pop()

    operations+=1

    total\_cost+=1

    unit.append(1)

def multipop(n, length):

    for i in range(n):

        pop()

    print("Stack after poping ",n,"elements: ",stack,"\t\t(As",n,"is less than stack length",length,")")

for i in arr:

    print("For element: ",i)

    if i<=len(stack):

        multipop(i, len(stack))

    push(i)

print("T(n) = ",sum(unit)," and Total cost: ",total\_cost)

print("Amortized Cost: ",sum(unit)//operations)

1. Accounting

def accounting(n):

    size = 1

    total = 0

    doubling\_cost = 0

    insertion\_cost = 0

    balance = 0

    print("Doubling Cost\tInsertion Cost\tTotal Cost\tBalance\t\tSize\n")

    for i in range(1, n+1):

        insertion\_cost=1

        if i>size:

            size\*=2

            doubling\_cost = i-1

        total = doubling\_cost+insertion\_cost

        balance+=(3-total)

        print(doubling\_cost,"\t\t",insertion\_cost,"\t\t",total,"\t\t",balance,"\t\t",size)

        doubling\_cost = 0

        insertion\_cost = 0

accounting(10)

1. Potential

doubling\_costs = []

current\_length = 1

potential = []

for i in range(1, 11):

    if current\_length < i:

        current\_length \*= 2

        doubling\_costs.append(i-1)

    else:

        doubling\_costs.append(0)

    potential.append(2\*i - current\_length)

total\_cost = [x+1 for x in doubling\_costs]

print('Doubling Cost\t Iteration\t Total Cost\t Potential\t Amortized Cost')

print(f'{doubling\_costs[0]}\t\t {1}\t\t {total\_cost[0]}\t\t {potential[0]}\t\t {total\_cost[0] + potential[0]}')

for j in range(1, 10):

    amortized\_cost = total\_cost[j] + potential[j] - potential[j-1]

    print(f'{doubling\_costs[j]}\t\t {j+1}\t\t {total\_cost[j]}\t\t {potential[j]}\t\t {amortized\_cost}')

1. Hiring

candidates = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print("Candidates: ", candidates)

interviewed\_candidates = []

hired\_candidates = []

for candidate in candidates:

    interviewed\_candidates.append(candidate)

    if not hired\_candidates or candidate > max(hired\_candidates):

        hired\_candidates.append(candidate)

firing\_cost = len(hired\_candidates) - 1  # Since the last candidate is the best

print("Normal way : ")

print("Interviewed candidates:", interviewed\_candidates)

print("Hired candidates:", hired\_candidates)

print("Number of candidates hired:", len(hired\_candidates))

print("Firing cost:", firing\_cost)

import random

candidates = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

interviewed\_candidates = []

hired\_candidates = []

print("\nRandomized Approach")

# Randomly select and interview candidates

for i in range(len(candidates)):

    temp = random.choice(candidates)

    interviewed\_candidates.append(temp)

    candidates.remove(temp)

max = -1

for i in interviewed\_candidates:

    if i > max:

        max = i

        hired\_candidates.append(i)

# Calculate firing cost

firing\_cost = len(hired\_candidates) - 1  # Since the last candidate is the best

print("Interviewed candidates in randomized order:", interviewed\_candidates)

print("Hired candidates:", hired\_candidates)

print("Number of candidates hired:", len(hired\_candidates))

print("Firing cost:", firing\_cost)

1. Lumoto (QS)

import random

import time

def lumoto\_partitoning(low, high, data):

    i = low

    j = low

    pivot = data[high]

    while j<high:

        if data[j]<pivot:

            data[i], data[j] = data[j],data[i]

            i+=1

        j+=1

    data[i],data[high] = data[high], data[i]

    return i

def quick\_sort(low, high, data):

    if low<high:

        p = lumoto\_partitoning(low, high, data)

        quick\_sort(low, p-1, data)

        quick\_sort(p+1, high, data)

def randomized\_partitioning(low, high, data):

    rand\_index = random.randint(low, high)

    data[rand\_index], data[high] = data[high], data[rand\_index]

    return lumoto\_partitoning(low, high, data)

def quick\_sort\_rand(low, high, data):

    if low<high:

        p = randomized\_partitioning(low, high, data)

        quick\_sort(low, p-1, data)

        quick\_sort(p+1, high, data)

data = [i for i in range(900,0,-1)]

start = time.time()

quick\_sort(0,len(data)-1,data)

end = time.time()

print("Time taken (Lumoto): ",end-start)

data = [i for i in range(900,0,-1)]

start = time.time()

quick\_sort\_rand(0,len(data)-1,data)

end = time.time()

print("Time taken (Lumoto with Randomization): ",end-start)

1. Hoarse (QS):

import random

import time

def hoare\_partitioning(low, high, data):

    i = low-1

    j = high+1

    pivot = data[low]

    while True:

        i+=1

        if data[i]<pivot:

            i+=1

        j-=1

        if data[j] >pivot:

            j-=1

        if i>=j:

            return j

        data[i], data[j] = data[j], data[i]

def quick\_sort(low, high, data):

    if low<high:

        p = hoare\_partitioning(low,high,data)

        quick\_sort(low, p, data)

        quick\_sort(p+1, high, data)

def randomized\_partitioning(low, high, data):

    random\_index = random.randint(low, high)

    data[random\_index], data[low] = data[low], data[random\_index]

    return hoare\_partitioning(low, high, data)

def quick\_sort\_rand(low, high, data):

    if low<high:

        p = randomized\_partitioning(low,high,data)

        quick\_sort(low, p, data)

        quick\_sort(p+1, high, data)

data = [i for i in range(900, 0, -1)]

start = time.time()

quick\_sort(0, len(data)-1, data)

end = time.time()

print("Time Taken (Hoare): ",end-start)

data = [i for i in range(900, 0, -1)]

start = time.time()

quick\_sort\_rand(0, len(data)-1, data)

end = time.time()

print("Time Taken (Hoare with Randomization): ",end-start)

1. KD-Tree (Balanced):

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.depth = 0

        self.left = None

        self.right = None

def traverse\_in\_order(curr):

    if curr is None:

        return

    traverse\_in\_order(curr.left)

    print(f"({', '.join(map(str, curr.data))}) ", end="")

    traverse\_in\_order(curr.right)

def make\_kd\_tree(seq, depth=0):

    if len(seq) == 0:

        return None

    k = len(seq[0]) #no of dimensions

    dim = depth % k

    seq.sort(key=lambda x: x[dim])

    mid = len(seq) // 2

    mid\_elem = seq[mid]

    root = Node(mid\_elem)

    left\_sub\_arr = seq[:mid]

    right\_sub\_arr = seq[mid+1:]

    root.depth = depth

    root.left = make\_kd\_tree(left\_sub\_arr, depth+1)

    root.right = make\_kd\_tree(right\_sub\_arr, depth+1)

    return root

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

root = make\_kd\_tree(seq)

print("Inorder Traversal: ",end='')

traverse\_in\_order(root)

1. KD-Tree (Unbalanced)

class KDNode:

    def \_\_init\_\_(self, data):

        self.data= data

        self.depth = 0

        self.left = None

        self.right = None

def insert(node,point):

    if node is None:

        return KDNode(point)

    dim=node.depth%2

    if point[dim] < node.data[dim]:

        node.left=insert(node.left,point)

        node.left.depth=node.depth+1

    else:

        node.right=insert(node.right,point)

        node.right.depth=node.depth+1

    return node

def inorder(node):

    if node is None:

      return

    inorder(node.left)

    print(f"({','.join(map(str,node.data))}) ",end="")

    inorder(node.right)

unbalanced\_points = [[6, 2], [7, 1], [2, 9], [3, 6], [4, 8], [8, 4], [5, 3], [1, 5], [9, 5]]

unbalanced\_root = KDNode(unbalanced\_points[0])

for point in unbalanced\_points[1:]:

  insert(unbalanced\_root, point)

print("Initial tree:")

inorder(unbalanced\_root)

insert(unbalanced\_root, [3,5])

print("\nAfter insertion of point (3,5):")

inorder(unbalanced\_root)

1. Ford-Fulkerson:

class Graph:

    def \_\_init\_\_(self, graph):

        self.graph = graph

        self.ROW = len(graph)

    def bfs(self, s, t, parent):

        visited = [False] \* self.ROW

        queue = []

        queue.append(s)

        visited[s] = True

        while queue:

            u = queue.pop(0)

            for ind, val in enumerate(self.graph[u]):

                if not visited[ind] and val > 0:

                    queue.append(ind)

                    visited[ind] = True

                    parent[ind] = u

        return visited[t], parent

    def ford\_fulkerson(self, source, sink):

        max\_flow = 0

        parent = [-1] \* self.ROW

        while True:

            found\_path, parent = self.bfs(source, sink, parent)

            if not found\_path:

                break

            path\_flow = float("Inf")

            s = sink

            while s != source:

                path\_flow = min(path\_flow, self.graph[parent[s]][s])

                s = parent[s]

            max\_flow += path\_flow

            # Print the augmented path and its minimum value

            path = [sink]

            v = sink

            while v != source:

                u = parent[v]

                path.insert(0, u)

                v = u

            print("Augmented path: ", " -> ".join(str(x) for x in path), " Minimum flow: ", path\_flow)

            v = sink

            while v != source:

                u = parent[v]

                self.graph[u][v] -= path\_flow

                self.graph[v][u] += path\_flow

                v = u

        return max\_flow

#            0   1   2   3   4   5

#            S,  B,  P,  M,  K,  D

graph = [   [0,  0,  0,  0,  0,  0],  #S

            [17, 0,  0,  0,  10, 0],  #B

            [6,  7,  0,  0,  0,  0],  #P

            [0,  12, 0,  0,  14, 0],  #M

            [0,  0, 10,  6,  0,  0],  #K

            [0,  0,  0,  8,  14, 0],  #D

        ]

g = Graph(graph)

source = 5

sink = 0

print("Max Flow: %d " % g.ford\_fulkerson(source, sink))

1. Convex Hull

import math

def structure(p, q, r):

    v = (q[1] - p[1]) \* (r[0] - q[0]) - (q[0] - p[0]) \* (r[1] - q[1])

    if v == 0:

        return 0 # 0 hai to collinear

    return 1 if v > 0 else -1 # 1=clockwise, -1=counterclockwise

def gs(points):

    n = len(points)

    if n < 3:

        return []

    min\_pt = min(points, key=lambda x: (x[1], x[0]))

    angle\_key = lambda x: math.atan2(x[1] - min\_pt[1], x[0] - min\_pt[0])

    sort\_pts = sorted(points, key=lambda x: (angle\_key(x), x))

    print(sort\_pts)

    stack = [sort\_pts[0], sort\_pts[1], sort\_pts[2]]

    print(f"After addition 3 points , stack : {stack}")

    for i in range(3, n):

        while len(stack) > 1 and structure(stack[-2], stack[-1], sort\_pts[i]) == 1:

            stack.pop()

        stack.append(sort\_pts[i])

        print(f"adding {sort\_pts[i]} --> stack : {stack}")

    return stack

points =[(0, 0), (3, 0), (1, 4), (3, 3), (5, 2), (5, 5), (9, 6), (7, 0),(10,0)]

print(f"Convex hull will be : {gs(points)}")