ASSIGNMENT (13.06.24)

1.Height of Binary Tree After Subtree Removal Queries

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def height\_of\_tree(node):

if not node:

return -1

left\_height = height\_of\_tree(node.left)

right\_height = height\_of\_tree(node.right)

return 1 + max(left\_height, right\_height)

def build\_tree\_from\_list(vals):

if not vals:

return None

nodes = [None if val is None else TreeNode(val) for val in vals]

kid\_idx = 1

for idx, node in enumerate(nodes):

if node is not None:

if kid\_idx < len(nodes) and nodes[kid\_idx] is not None:

node.left = nodes[kid\_idx]

kid\_idx += 1

if kid\_idx < len(nodes) and nodes[kid\_idx] is not None:

node.right = nodes[kid\_idx]

kid\_idx += 1

return nodes[0]

def find\_node(root, val):

if root is None:

return None

if root.val == val:

return root

left\_result = find\_node(root.left, val)

if left\_result is not None:

return left\_result

return find\_node(root.right, val)

def height\_after\_removal(root, query):

target\_node = find\_node(root, query)

if not target\_node:

return height\_of\_tree(root)

if target\_node.left:

original\_left = target\_node.left

target\_node.left = None

height\_with\_left\_removed = height\_of\_tree(root)

target\_node.left = original\_left

return height\_with\_left\_removed

if target\_node.right:

original\_right = target\_node.right

target\_node.right = None

height\_with\_right\_removed = height\_of\_tree(root)

target\_node.right = original\_right

return height\_with\_right\_removed

return height\_of\_tree(root)

def height\_after\_subtree\_removal(root, queries):

root\_tree\_height = height\_of\_tree(root)

results = []

for query in queries:

results.append(height\_after\_removal(root, query))

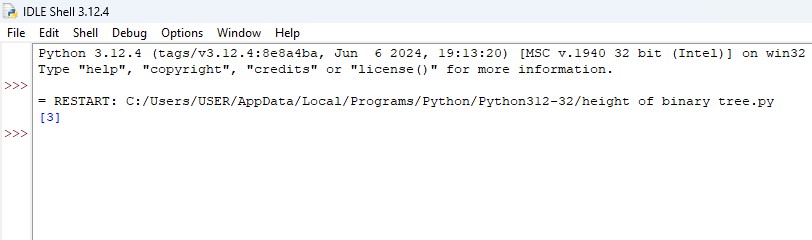
return results

root\_vals = [1, 3, 4, 2, None, 6, 5, None, None, None, None, None, 7]

queries = [4]

root = build\_tree\_from\_list(root\_vals)

print(height\_after\_subtree\_removal(root, queries))



2. Sort Array by Moving Items to Empty Space

def min\_operations\_to\_sort(nums):

n = len(nums)

visited = [False] \* n

zero\_index = nums.index(0)

swaps = 0

for i in range(n):

if not visited[i] and nums[i] != i:

cycle\_length = 0

x = i

while not visited[x]:

visited[x] = True

x = nums[x]

cycle\_length += 1

if cycle\_length > 0:

swaps += cycle\_length - 1

if zero\_index == 0 or zero\_index == n - 1:

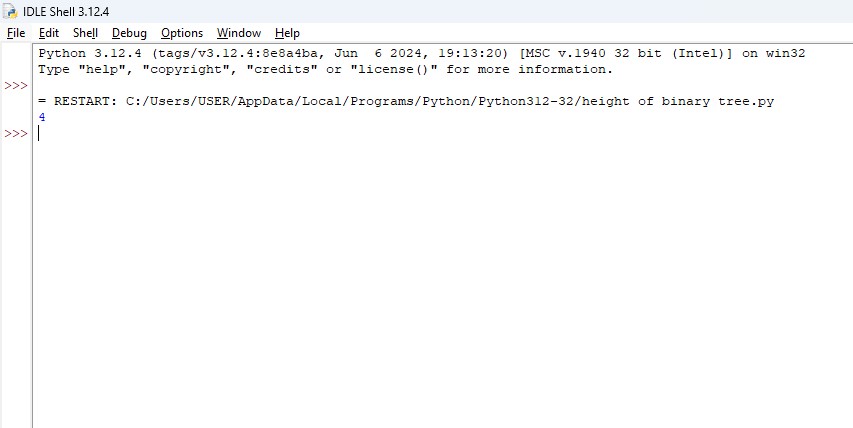
return swaps

else:

return swaps + 1

nums = [4, 2, 0, 3, 1]

print(min\_operations\_to\_sort(nums))



3. Apply Operations to an Array

def apply\_operations(nums):

n = len(nums)

for i in range(n - 1):

if nums[i] == nums[i + 1]:

nums[i] \*= 2

nums[i + 1] = 0

result = []

zero\_count = 0

for num in nums:

if num != 0:

result.append(num)

else:

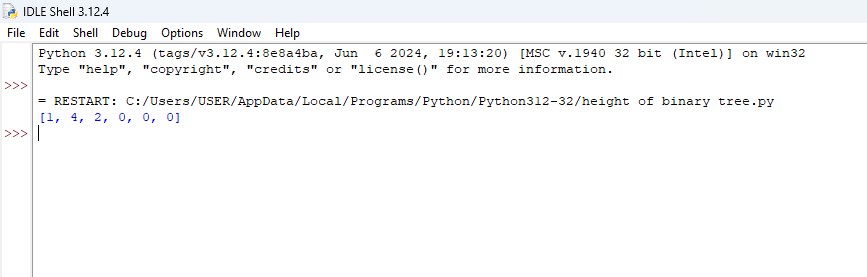
zero\_count += 1

result.extend([0] \* zero\_count)

return result

nums = [1, 2, 2, 1, 1, 0]

print(apply\_operations(nums))



4. Maximum Sum of Distinct Subarrays With Length K

def max\_sum\_of\_distinct\_subarrays(nums, k):

if len(nums) < k:

return 0

n = len(nums)

max\_sum = 0

current\_sum = 0

window\_start = 0

element\_set = set()

for window\_end in range(n):

while nums[window\_end] in element\_set:

element\_set.remove(nums[window\_start])

current\_sum -= nums[window\_start]

window\_start += 1

element\_set.add(nums[window\_end])

current\_sum += nums[window\_end]

if window\_end - window\_start + 1 == k:

max\_sum = max(max\_sum, current\_sum)

element\_set.remove(nums[window\_start])

current\_sum -= nums[window\_start]

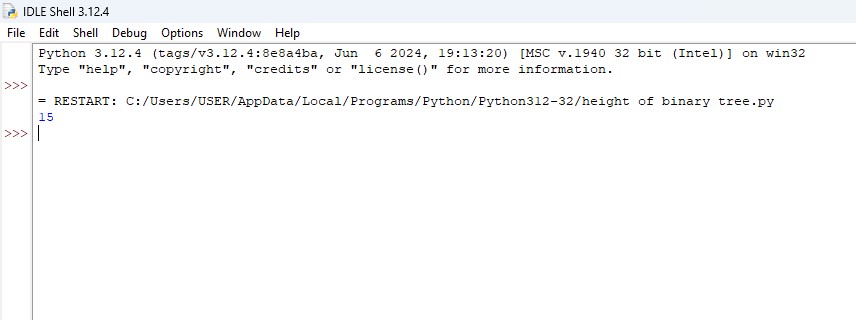
window\_start += 1

return max\_sum

nums1 = [1, 5, 4, 2, 9, 9, 9]

k1 = 3

print(max\_sum\_of\_distinct\_subarrays(nums1, k1))



5. Total Cost to Hire K Workers

from heapq import heappop, heappush

def total\_cost\_to\_hire\_workers(costs, k, candidates):

n = len(costs)

left\_heap = []

right\_heap = []

for i in range(min(candidates, n)):

heappush(left\_heap, (costs[i], i))

for i in range(max(candidates, n - candidates), n):

heappush(right\_heap, (costs[i], i))

total\_cost = 0

hired\_count = 0

left\_index = min(candidates, n)

right\_index = max(candidates, n - candidates)

while hired\_count < k:

if left\_heap and (not right\_heap or left\_heap[0][0] <= right\_heap[0][0]):

cost, index = heappop(left\_heap)

total\_cost += cost

if left\_index < right\_index:

heappush(left\_heap, (costs[left\_index], left\_index))

left\_index += 1

else:

cost, index = heappop(right\_heap)

total\_cost += cost

if right\_index > left\_index:

heappush(right\_heap, (costs[right\_index - 1], right\_index - 1))

right\_index -= 1

hired\_count += 1

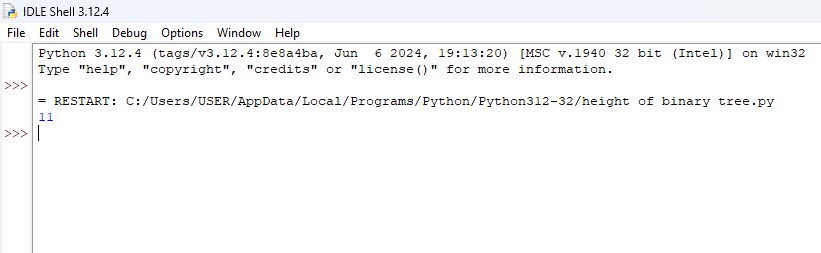
return total\_cost

costs = [17, 12, 10, 2, 7, 2, 11, 20, 8]

k = 3

candidates = 4

print(total\_cost\_to\_hire\_workers(costs, k, candidates))



6. Minimum Total Distance Traveled

def min\_total\_distance\_traveled(robots, factories):

robots.sort()

factories.sort()

r\_len = len(robots)

f\_len = len(factories)

dp = [[float('inf')] \* (f\_len + 1) for \_ in range(r\_len + 1)]

dp[0][0] = 0

for i in range(r\_len + 1):

dp[i][0] = 0

for j in range(1, f\_len + 1):

dp[0][j] = 0

for i in range(1, r\_len + 1):

for j in range(1, f\_len + 1):

dp[i][j] = dp[i][j-1]

limit\_j = factories[j-1][1]

total\_distance = 0

for t in range(1, min(limit\_j, i) + 1):

total\_distance += abs(robots[i-t] - factories[j-1][0])

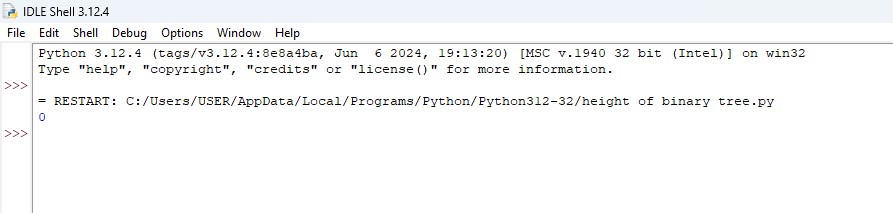
dp[i][j] = min(dp[i][j], dp[i-t][j-1] + total\_distance)

return dp[r\_len][f\_len]

robots = [0, 4, 6]

factories = [[2, 2], [6, 2]]

print(min\_total\_distance\_traveled(robots, factories))



7. Minimum Subarrays in a Valid Split

from math import gcd

from functools import lru\_cache

def min\_subarrays\_in\_valid\_split(nums):

n = len(nums)

dp = [float('inf')] \* (n + 1)

dp[0] = 0

@lru\_cache(None)

def gcd\_range(l, r):

result = nums[l]

for i in range(l + 1, r):

result = gcd(result, nums[i])

if result == 1:

return 1

return result

for j in range(1, n + 1):

for i in range(j):

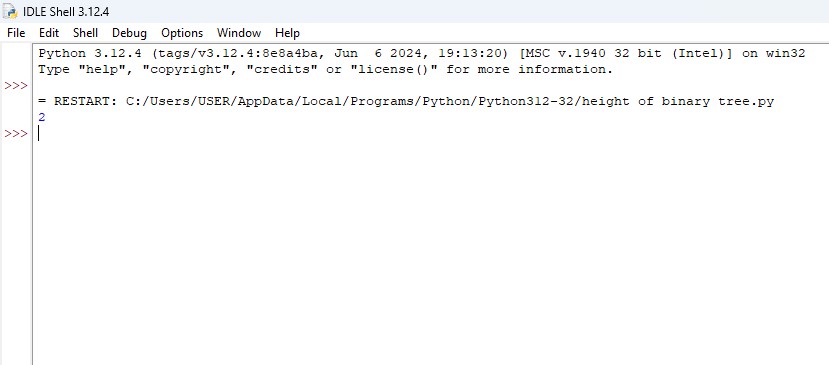
if gcd(nums[i], nums[j - 1]) > 1:

dp[j] = min(dp[j], dp[i] + 1)

return dp[n] if dp[n] != float('inf') else -1

nums1 = [2, 6, 3, 4, 3]

print(min\_subarrays\_in\_valid\_split(nums1))



8. Number of Distinct Averages

def distinct\_averages(nums):

nums.sort()

averages = set()

i, j = 0, len(nums) - 1

while i < j:

avg = (nums[i] + nums[j]) / 2

averages.add(avg)

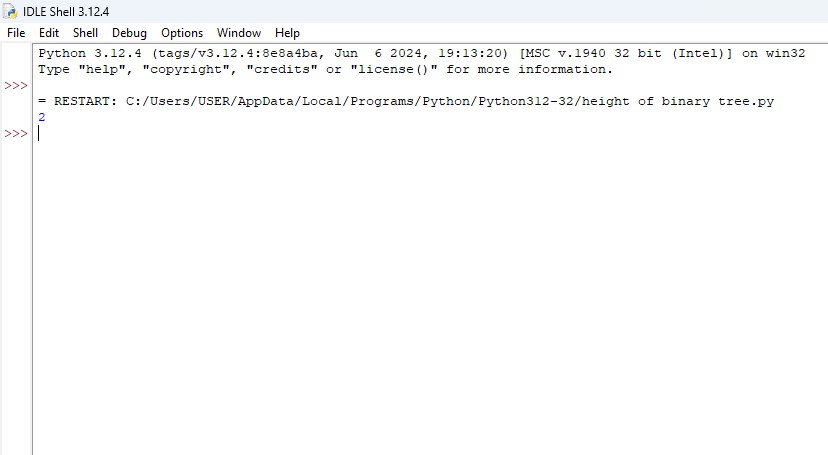
i += 1

j -= 1

return len(averages)

nums1 = [4, 1, 4, 0, 3, 5]

print(distinct\_averages(nums1))



9. Count Ways To Build Good Strings

def count\_good\_strings(low, high, zero, one):

MOD = 10\*\*9 + 7

dp = [0] \* (high + 1)

dp[0] = 1

for i in range(1, high + 1):

if i >= zero:

dp[i] += dp[i - zero]

if i >= one:

dp[i] += dp[i - one]

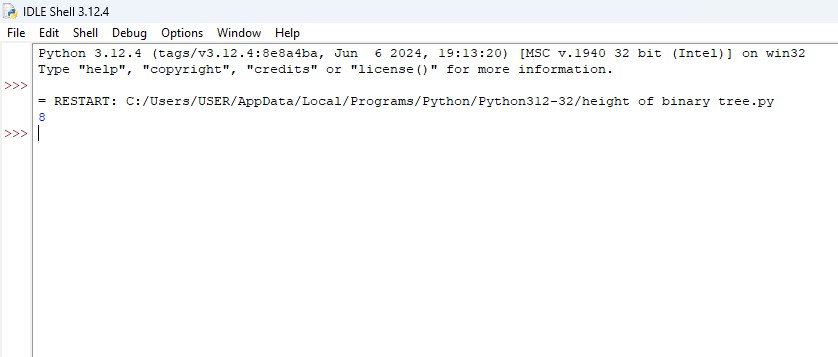
dp[i] %= MOD

result = sum(dp[low:high + 1]) % MOD

return result

low1, high1, zero1, one1 = 3, 3, 1, 1

print(count\_good\_strings(low1, high1, zero1, one1))



10. Most Profitable Path in a Tree

def most\_profitable\_path(edges, bob, amount):

from collections import defaultdict, deque

tree = defaultdict(list)

for u, v in edges:

tree[u].append(v)

tree[v].append(u)

def find\_bob\_path(node, parent):

if node == 0:

return [node]

for neighbor in tree[node]:

if neighbor != parent:

path = find\_bob\_path(neighbor, node)

if path:

return path + [node]

return []

bob\_path = find\_bob\_path(bob, -1)

bob\_time = {node: time for time, node in enumerate(bob\_path)}

def dfs(node, parent, time):

profit = amount[node]

if node in bob\_time:

if bob\_time[node] < time:

profit = 0

elif bob\_time[node] == time:

profit //= 2

max\_profit = float('-inf')

is\_leaf = True

for neighbor in tree[node]:

if neighbor != parent:

is\_leaf = False

max\_profit = max(max\_profit, dfs(neighbor, node, time + 1))

if is\_leaf:

return profit

return profit + max\_profit

return dfs(0, -1, 0)

edges1 = [[0, 1], [1, 2], [1, 3], [3, 4]]

bob1 = 3

amount1 = [-2, 4, 2, -4, 6]

print(most\_profitable\_path(edges1, bob1, amount1))

