**DAY 3**

1. Write a program to perform the following o An empty list o A list with one element o A list with all identical elements o A list with negative numbers

Test Cases:

1. Input: [] o Expected Output: []

2. Input: [1] o Expected Output: [1]

3. Input: [7, 7, 7, 7] o Expected Output: [7, 7, 7, 7]

4. Input: [-5, -1, -3, -2, -4] o Expected Output: [-5, -4, -3, -2, -1]

**CODE:**

def process\_list(input\_list):

if len(input\_list) <= 1:

return input\_list

return sorted(input\_list)

test\_cases = [

([], []),

([1], [1]),

([7, 7, 7, 7], [7, 7, 7, 7]),

([-5, -1, -3, -2, -4], [-5, -4, -3, -2, -1])

]

for input\_list, expected\_output in test\_cases:

result = process\_list(input\_list)

print(f"Input: {input\_list}\nExpected Output: {expected\_output}\nActual Output: {result}\n")

**OUTPUT:**

Input: []

Expected Output: []

Actual Output: []

Input: [1]

Expected Output: [1]

Actual Output: [1]

Input: [7, 7, 7, 7]

Expected Output: [7, 7, 7, 7]

Actual Output: [7, 7, 7, 7]

Input: [-5, -1, -3, -2, -4]

Expected Output: [-5, -4, -3, -2, -1]

Actual Output: [-5, -4, -3, -2, -1]

1. Describe the Selection Sort algorithm's process of sorting an array. Selection Sort works by dividing the array into a sorted and an unsorted region. Initially, the sorted region is empty, and the unsorted region contains all elements. The algorithm repeatedly selects the smallest element from the unsorted region and swaps it with the leftmost unsorted element, then moves the boundary of the sorted region one element to the right. Explain why Selection Sort is simple to understand and implement but is inefficient for large datasets. Provide examples to illustrate step-by-step how Selection Sort rearranges the elements into ascending order, ensuring clarity in your explanation of the algorithm's mechanics and effectiveness.

Sorting a Random Array: Input: [5, 2, 9, 1, 5, 6] Output: [1, 2, 5, 5, 6, 9]

Sorting a Reverse Sorted Array: Input: [10, 8, 6, 4, 2] Output: [2, 4, 6, 8, 10]

Sorting an Already Sorted Array: Input: [1, 2, 3, 4, 5] Output: [1, 2, 3, 4, 5]

**CODE:**

def selection\_sort(arr):

for i in range(len(arr)):

min\_index = i

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

if arr[j] < arr[min\_index]:

min\_index = j

arr[i], arr[min\_index] = arr[min\_index], arr[i]

return arr

# Test cases

# Example 1: Sorting a random array

random\_array = [5, 2, 9, 1, 5, 6]

sorted\_random = selection\_sort(random\_array.copy())

print(f"Input: {random\_array}")

print(f"Sorted: {sorted\_random}")

# Example 2: Sorting a reverse sorted array

reverse\_sorted\_array = [10, 8, 6, 4, 2]

sorted\_reverse = selection\_sort(reverse\_sorted\_array.copy())

print(f"\nInput: {reverse\_sorted\_array}")

print(f"Sorted: {sorted\_reverse}")

# Example 3: Sorting an already sorted array

already\_sorted\_array = [1, 2, 3, 4, 5]

sorted\_already = selection\_sort(already\_sorted\_array.copy())

print(f"\nInput: {already\_sorted\_array}")

print(f"Sorted: {sorted\_already}")

1. Write code to modify bubble\_sort function to stop early if the list becomes sorted before all passes are completed.

Test Cases: • Test your optimized function with the following lists:

Input: [64, 25, 12, 22, 11]

Expected Output: [11, 12, 22, 25, 64]

**CODE:**

def bubble\_sort(arr):

n = len(arr)

for i in range(n):

swapped = False

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

swapped = True

if not swapped:

break

return arr

test\_case = [64, 25, 12, 22, 11]

sorted\_array = bubble\_sort(test\_case.copy())

print(sorted\_array)

**OUTPUT:**

[11, 12, 22, 25, 64]

1. Write code for Insertion Sort that manages arrays with duplicate elements during the sorting process. Ensure the algorithm's behavior when encountering duplicate values, including whether it preserves the relative order of duplicates and how it affects the overall sorting outcome.

Examples: 1. Array with Duplicates:

Input: [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

Output: [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

**CODE:**

def insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i]

j = i - 1

while j >= 0 and key < arr[j]:

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

return arr

# Test case

test\_case = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]

sorted\_array = insertion\_sort(test\_case.copy())

print(sorted\_array)

**OUTPUT:** [1, 1, 2, 3, 3, 4, 5, 5, 6, 9]

1. Given an array arr of positive integers sorted in a strictly increasing order, and an integer k. return the kth positive integer that is missing from this array.

Example 1: Input: arr = [2,3,4,7,11], k = 5

Output: 9

**CODE:**

def find\_kth\_missing(arr, k):

missing\_count = 0

current = 1

i = 0

while missing\_count < k:

if i < len(arr) and arr[i] == current:

i += 1

else:

missing\_count += 1

if missing\_count == k:

return current

current += 1

arr = [2, 3, 4, 7, 11]

k = 5

result = find\_kth\_missing(arr, k)

print(result)

OUTPUT: 9

1. A peak element is an element that is strictly greater than its neighbors. Given a 0-indexed integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks. You may imagine that nums[-1] = nums[n] = -∞. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array. You must write an algorithm that runs in O(log n) time.

Example 1:

Input: nums = [1,2,3,1]

Output: 2

**CODE:**

def find\_peak\_element(nums):

left, right = 0, len(nums) - 1

while left < right:

mid = (left + right) // 2

if nums[mid] < nums[mid + 1]:

left = mid + 1

else:

right = mid

return left

# Test case

nums = [1, 2, 3, 1]

result = find\_peak\_element(nums)

print(result)

OUTPUT: 2

1. Given two strings needle and haystack, return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

Example 1:

Input: haystack = "sadbutsad", needle = "sad"

Output: 0

Explanation: "sad" occurs at index 0 and 6. The first occurrence is at index 0, so we return 0.

**CODE:**

def strStr(haystack, needle):

if not needle:

return 0

return haystack.find(needle)

# Test case

haystack = "sadbutsad"

needle = "sad"

result = strStr(haystack, needle)

print(result)

OUTPUT: 0

1. Given an array of string words, return all strings in words that is a substring of another word. You can return the answer in any order. A substring is a contiguous sequence of characters within a string

Example 1: Input: words = ["mass","as","hero","superhero"]

Output: ["as","hero"]

Explanation: "as" is substring of "mass" and "hero" is substring of "superhero". ["hero","as"] is also a valid answer.

**CODE**

def string\_substrings(words):

substrings = set()

for i in range(len(words)):

for j in range(len(words)):

if i != j and words[i] in words[j]:

substrings.add(words[i])

break

return list(substrings)

words = ["mass", "as", "hero", "superhero"]

result = string\_substrings(words)

print(result)

OUTPUT: ['as', 'hero']