Ai-assisted lab assessment – 12.4

* Task1: **Implementing Bubble Sort with AI Comments**
* Prompt: Write a Python implementation of Bubble Sort.  
  Then, add detailed inline comments explaining the key logic, including:

and why elements are swapped,How passes through the list work

What condition causes the algorithm to terminate,Finally, provide a time complexity analysis of Bubble Sort in best, average, and worst-case scenarios.

* Code:
* def bubble\_sort(arr):
* """
* Sorts a list of numbers using the Bubble Sort algorithm.
* Args:
* arr: The list of numbers to be sorted.
* Returns:
* The sorted list.
* """
* n = len(arr)
* # Outer loop for passes through the list
* # This loop iterates n-1 times, where n is the length of the list.
* # In each pass, the largest unsorted element 'bubbles up' to its correct position.
* for i in range(n - 1):
* # Inner loop for comparisons and swaps within each pass
* # This loop compares adjacent elements and swaps them if they are in the wrong order.
* # The range decreases with each outer loop iteration because the last i elements are already sorted.
* for j in range(n - 1 - i):
* # Compare adjacent elements
* if arr[j] > arr[j + 1]:
* # Swap elements if they are in the wrong order
* # This is done using tuple packing/unpacking in Python.
* # The larger element is moved to the right.
* arr[j], arr[j + 1] = arr[j + 1], arr[j]
* # The algorithm terminates when the outer loop completes n-1 passes.
* # After n-1 passes, the list is guaranteed to be sorted.
* return arr
* # Example usage:
* my\_list = [64, 34, 25, 12, 22, 11, 90]
* sorted\_list = bubble\_sort(my\_list)
* print("Sorted array is:", sorted\_list)
* output: Sorted array is: [11, 12, 22, 25, 34, 64, 90]
* TASK2: **Optimizing Bubble Sort → Insertion Sort**
* PROMPT: I have implemented Bubble Sort.  
  Suggest a more efficient algorithm for partially sorted arrays.  
  Provide a Python implementation of Insertion Sort.  
  Explain why Insertion Sort performs better than Bubble Sort on nearly sorted data.  
  Compare their time complexities and behavior on nearly sorted input.
* CODE:
* def insertion\_sort(arr):
* """
* Sorts a list of numbers using the Insertion Sort algorithm.
* Args:
* arr: The list of numbers to be sorted.
* Returns:
* The sorted list.
* """
* # Traverse through 1 to len(arr)
* # The outer loop iterates from the second element to the end of the list.
* # Each element from the second one onwards is considered for insertion into the sorted portion of the list.
* for i in range(1, len(arr)):
* # The current element to be inserted
* key = arr[i]
* # Initialize j to the index of the element just before the current element
* # This is where we start comparing the key with elements in the sorted portion.
* j = i - 1
* # Move elements of arr[0..i-1], that are greater than key,
* # to one position ahead of their current position
* # This inner loop shifts elements to the right to make space for the key
* # if they are larger than the key. It stops when it finds an element
* # smaller than or equal to the key, or when it reaches the beginning of the list.
* while j >= 0 and key < arr[j]:
* arr[j + 1] = arr[j]
* j -= 1
* # Place the key at its correct position in the sorted subarray
* # Once the inner loop finishes, j+1 is the index where the key should be inserted.
* arr[j + 1] = key
* return arr
* # Example usage:
* my\_list = [12, 11, 13, 5, 6]
* sorted\_list = insertion\_sort(my\_list)
* print("Sorted array is:", sorted\_list)
* my\_nearly\_sorted\_list = [2, 3, 4, 1, 5, 6, 7]
* sorted\_nearly\_sorted\_list = insertion\_sort(my\_nearly\_sorted\_list)
* print("Sorted nearly sorted array is:", sorted\_nearly\_sorted\_list)
* OUTPUT: Sorted array is: [5, 6, 11, 12, 13]
* Sorted nearly sorted array is: [1, 2, 3, 4, 5, 6, 7]
* TASK3: Binary Search vs Linear Search
* PROMPT: Write Python implementations of both Linear Search and Binary Search.  
  Add docstrings to each function explaining its purpose and usage.  
  Include performance notes for each algorithm.  
  Explain when Binary Search is preferable over Linear Search.  
  Also, provide a comparison table showing their performance on sorted vs unsorted data.
* LINEAR CODE AND OUTPUT:
* def linear\_search(arr, target):
* """
* Performs a linear search for a target value in a list.
* Args:
* arr: The list to search within.
* target: The value to search for.
* Returns:
* The index of the target value if found, otherwise -1.
* """
* # Iterate through each element of the list
* for i in range(len(arr)):
* # If the current element matches the target, return its index
* if arr[i] == target:
* return i
* # If the target is not found after iterating through the entire list, return -1
* return -1
* # Performance Notes for Linear Search:
* # - Time Complexity:
* #   - Best Case: O(1) (when the target is the first element)
* #   - Average Case: O(n) (when the target is somewhere in the middle)
* #   - Worst Case: O(n) (when the target is the last element or not in the list)
* # - Space Complexity: O(1) (constant extra space)
* # - Linear search is simple to implement and works on both sorted and unsorted lists.
* # - However, it can be inefficient for large lists, especially in the average and worst cases.
* # Example usage:
* my\_list = [10, 5, 20, 15, 25, 30]
* target\_value = 20
* index = linear\_search(my\_list, target\_value)
* if index != -1:
* print(f"Linear Search: Target {target\_value} found at index {index}")
* else:
* print(f"Linear Search: Target {target\_value} not found")
* target\_value = 35
* index = linear\_search(my\_list, target\_value)
* if index != -1:
* print(f"Linear Search: Target {target\_value} found at index {index}")
* else:
* print(f"Linear Search: Target {target\_value} not found")
* Linear Search: Target 20 found at index 2
* Linear Search: Target 35 not found
* BINARY CIDE AND OUTPUT:
* def binary\_search(arr, target):
* """
* Performs a binary search for a target value in a sorted list.
* Args:
* arr: The sorted list to search within.
* target: The value to search for.
* Returns:
* The index of the target value if found, otherwise -1.
* """
* low = 0
* high = len(arr) - 1
* # Continue searching while the search space is valid
* while low <= high:
* # Calculate the middle index
* mid = (low + high) // 2
* # Get the value at the middle index
* mid\_val = arr[mid]
* # If the middle value is the target, return the index
* if mid\_val == target:
* return mid
* # If the target is less than the middle value, search in the left half
* elif target < mid\_val:
* high = mid - 1
* # If the target is greater than the middle value, search in the right half
* else:
* low = mid + 1
* # If the loop finishes without finding the target, return -1
* return -1
* # Performance Notes for Binary Search:
* # - Time Complexity:
* #   - Best Case: O(1) (when the target is the middle element in the first comparison)
* #   - Average Case: O(log n) (due to the halving of the search space in each step)
* #   - Worst Case: O(log n) (when the target is not in the list or at the edges of the search space)
* # - Space Complexity: O(1) (constant extra space for iterative approach) or O(log n) (for recursive approach due to function call stack)
* # - Binary search is significantly more efficient than linear search for large sorted lists.
* # - It requires the input list to be sorted.
* # Example usage:
* my\_sorted\_list = [10, 15, 20, 25, 30, 35]
* target\_value = 25
* index = binary\_search(my\_sorted\_list, target\_value)
* if index != -1:
* print(f"Binary Search: Target {target\_value} found at index {index}")
* else:
* print(f"Binary Search: Target {target\_value} not found")
* target\_value = 12
* index = binary\_search(my\_sorted\_list, target\_value)
* if index != -1:
* print(f"Binary Search: Target {target\_value} found at index {index}")
* else:
* print(f"Binary Search: Target {target\_value} not found")
* Binary Search: Target 25 found at index 3
* Binary Search: Target 12 not found
* TASK4: Quick Sort and Merge Sort Comparison
* PROMPT: I have partially completed recursive functions for Quick Sort and Merge Sort.  
  Complete the missing logic in both functions.  
  Add docstrings explaining each function.  
  Compare Quick Sort and Merge Sort on random, sorted, and reverse-sorted lists.  
  Provide a complexity analysis for best, average, and worst-case scenarios.
* CODE:
* import time
* import random
* def quick\_sort(arr):
* """
* Sorts a list of numbers using the Quick Sort algorithm.
* Quick Sort is a divide-and-conquer algorithm. It works by selecting a
* 'pivot' element from the array and partitioning the other elements into
* two sub-arrays, according to whether they are less than or greater than
* the pivot. The sub-arrays are then recursively sorted.
* Args:
* arr: The list of numbers to be sorted.
* Returns:
* The sorted list.
* """
* # Base case for recursion: if the list has 0 or 1 element, it's already sorted.
* if len(arr) <= 1:
* return arr
* # Choose a pivot element. Here, we choose the last element.
* pivot = arr[-1]
* # Partition the remaining elements into three lists: less, equal, and greater.
* less = []
* equal = []
* greater = []
* for x in arr:
* if x < pivot:
* less.append(x)
* elif x == pivot:
* equal.append(x)
* else:
* greater.append(x)
* # Recursively sort the 'less' and 'greater' sub-lists and combine them
* # with the 'equal' list to get the final sorted list.
* return quick\_sort(less) + equal + quick\_sort(greater)
* def merge\_sort(arr):
* """
* Sorts a list of numbers using the Merge Sort algorithm.
* Merge Sort is a divide-and-conquer algorithm that recursively divides
* the input list into smaller sublists until they are trivially sorted,
* and then merges these sublists in a sorted manner.
* Args:
* arr: The list of numbers to be sorted.
* Returns:
* The sorted list.
* """
* # Base case: if the list has 0 or 1 element, it's already sorted.
* if len(arr) <= 1:
* return arr
* # Find the middle point to divide the list into two halves
* mid = len(arr) // 2
* left\_half = arr[:mid]
* right\_half = arr[mid:]
* # Recursively sort the two halves
* left\_half = merge\_sort(left\_half)
* right\_half = merge\_sort(right\_half)
* # Merge the sorted halves
* return merge(left\_half, right\_half)
* def merge(left, right):
* """
* Merges two sorted lists into a single sorted list.
* Args:
* left: The left sorted list.
* right: The right sorted list.
* Returns:
* A single sorted list containing elements from both input lists.
* """
* merged\_list = []
* i = j = 0
* # Compare elements from both lists and append the smaller one to the merged list
* while i < len(left) and j < len(right):
* if left[i] < right[j]:
* merged\_list.append(left[i])
* i += 1
* else:
* merged\_list.append(right[j])
* j += 1
* # Append any remaining elements from the left list
* while i < len(left):
* merged\_list.append(left[i])
* i += 1
* # Append any remaining elements from the right list
* while j < len(right):
* merged\_list.append(right[j])
* j += 1
* return merged\_list
* # Function to measure execution time
* def measure\_time(sort\_function, arr):
* start\_time = time.time()
* sort\_function(arr.copy()) # Use a copy to avoid modifying the original list
* end\_time = time.time()
* return end\_time - start\_time
* # Generate different types of lists
* list\_size = 10000  # You can adjust the size for testing
* # Random list
* random\_list = [random.randint(0, list\_size) for \_ in range(list\_size)]
* # Sorted list
* sorted\_list = list(range(list\_size))
* # Reverse-sorted list
* reverse\_sorted\_list = list(range(list\_size, 0, -1))
* # Compare performance on random list
* time\_quick\_random = measure\_time(quick\_sort, random\_list)
* time\_merge\_random = measure\_time(merge\_sort, random\_list)
* print(f"Performance on Random List (size {list\_size}):")
* print(f"Quick Sort: {time\_quick\_random:.6f} seconds")
* print(f"Merge Sort: {time\_merge\_random:.6f} seconds")
* print("-" \* 30)
* # Compare performance on sorted list
* time\_quick\_sorted = measure\_time(quick\_sort, sorted\_list)
* time\_merge\_sorted = measure\_time(merge\_sort, sorted\_list)
* print(f"Performance on Sorted List (size {list\_size}):")
* print(f"Quick Sort: {time\_quick\_sorted:.6f} seconds")
* print(f"Merge Sort: {time\_merge\_sorted:.6f} seconds")
* print("-" \* 30)
* # Compare performance on reverse-sorted list
* time\_quick\_reverse\_sorted = measure\_time(quick\_sort, reverse\_sorted\_list)
* time\_merge\_reverse\_sorted = measure\_time(merge\_sort, reverse\_sorted\_list)
* print(f"Performance on Reverse-Sorted List (size {list\_size}):")
* print(f"Quick Sort: {time\_quick\_reverse\_sorted:.6f} seconds")
* print(f"Merge Sort: {time\_merge\_reverse\_sorted:.6f} seconds")
* print("-" \* 30)
* OUTPUT:
* Performance on Random List (size 10000):
* Quick Sort: 0.011865 seconds
* Merge Sort: 0.024295 seconds
* TASK5: AI-Suggested Algorithm Optimization
* PROMPT: Here's a naive Python algorithm to find duplicates in a list using brute force (O(n²) time).  
  Optimize it using a more efficient approach (e.g., sets or dictionaries) to achieve O(n) time.  
  Provide both versions of the code.  
  Explain how the optimized version improves time complexity.  
  Compare execution times on large input sizes.
* CODE:
* def find\_duplicates\_naive(arr):
* """
* Finds duplicate elements in a list using a naive brute-force approach (O(n^2)).
* Args:
* arr: The list to search for duplicates.
* Returns:
* A list of unique duplicate elements found in the input list.
* """
* duplicates = []
* n = len(arr)
* # Iterate through each element in the list
* for i in range(n):
* # Compare the current element with all subsequent elements
* for j in range(i + 1, n):
* # If a duplicate is found and it's not already in the duplicates list, add it
* if arr[i] == arr[j] and arr[i] not in duplicates:
* duplicates.append(arr[i])
* return duplicates
* # Example usage:
* my\_list = [1, 2, 3, 4, 2, 5, 6, 3, 7, 1]
* duplicate\_items = find\_duplicates\_naive(my\_list)
* print("Naive approach - Duplicates:", duplicate\_items)
* my\_list\_2 = [10, 20, 30, 40, 50]
* duplicate\_items\_2 = find\_duplicates\_naive(my\_list\_2)
* print("Naive approach - Duplicates:", duplicate\_items\_2)
* OUTPUT:
* Naive approach - Duplicates: [1, 2, 3]
* Naive approach - uplicates: []