**ASSIGNMENT – 12.4**

**NAME : Bhavaneesh.A**

**ROLL:NO : 2403A52350**

**BATCH : AI 13**

**TASK-1:**

**Prompt:** I have implemented Bubble Sort in Python. Please provide detailed inline comments explaining the key logic in my code, such as swapping elements, passes through the array, and termination condition.

Also, include a time complexity analysis for the best, average, and worst cases.

**Code:**

**A screenshot of a computer code

AI-generated content may be incorrect.**

**Output:**

****

**Explanation:**

The code implements the Bubble Sort algorithm in Python. It defines a function bubble\_sort that takes an array arr as input. It iterates through the array multiple times, comparing adjacent elements and swapping them if they are in the wrong order. This process continues until the array is sorted. The comments within the code explain the purpose of each part of the algorithm, such as the loops and the swapping mechanism.

The second cell (with the markdown) provides a time complexity analysis of the Bubble Sort algorithm. It explains that the algorithm has a time complexity of O(n^2) in the worst and average cases due to the nested loops. It also mentions that in the best case (when the array is already sorted), the time complexity is O(n) because the algorithm can terminate early. Finally, it states that the space complexity is O(1) because it only uses a constant amount of extra space.

**TASK-2:**

**Prompt:** I have implemented Bubble Sort in Python. Can you suggest a more efficient sorting algorithm for nearly sorted (partially sorted) arrays? Also, please provide the Python code for that algorithm and explain why it's better than Bubble Sort in this case.

**Code:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Output:**

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Explanation:**

The code performs a performance comparison between Bubble Sort and Insertion Sort on a partially sorted array.

Here's a breakdown:

1. bubble\_sort(arr) and insertion\_sort(arr) functions: These are the implementations of the two sorting algorithms.
2. generate\_partially\_sorted\_array(size, degree\_of\_sortedness) function: This function creates an array of a given size and then shuffles it to a certain degree\_of\_sortedness. A lower degree\_of\_sortedness means the array is more partially sorted.
3. Array Creation and Copying: A partially sorted array is generated using the generate\_partially\_sorted\_array function. Copies of this array are made for each sorting algorithm to ensure they are tested on the exact same initial data.
4. Time Measurement: The time.time() function is used to record the start and end times of executing each sorting algorithm.
5. Performance Output: The code calculates the duration each sort took and prints these durations. It also provides a simple conclusion based on which algorithm was faster.

In essence, the code sets up a controlled experiment to see how much faster Insertion Sort is compared to Bubble Sort when the input data is not completely random but has some degree of order.

**TASK-3:**

**Prompt:** Please provide Python implementations for Linear Search and Binary Search with detailed docstrings explaining their parameters, return values, and functionality.

Also, include notes on their time complexity and performance characteristics.

**Code:**

import time

import random

def linear\_search(arr, target):

    for i in range(len(arr)):

        if arr[i] == target:

            return i  # Found at index i

    return -1  # Not found

def binary\_search(arr, target):

    left, right = 0, len(arr) - 1

    while left <= right:

        # Calculate middle index (avoids overflow in other languages)

        mid = left + (right - left) // 2

        # Check if target is at middle

        if arr[mid] == target:

            return mid

        # If target is greater, ignore left half

        elif arr[mid] < target:

            left = mid + 1

        # If target is smaller, ignore right half

        else:

            right = mid - 1

    return -1  # Target not found

def measure\_search\_time(search\_func, arr, target):

    """Measure execution time of search function"""

    start = time.perf\_counter()

    result = search\_func(arr, target)

    end = time.perf\_counter()

    return result, (end - start) \* 1\_000\_000  # Convert to microseconds

# Performance Testing

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

    print("=" \* 80)

    print("LINEAR SEARCH vs BINARY SEARCH - PERFORMANCE COMPARISON")

    print("=" \* 80)

    # Test on different array sizes

    sizes = [100, 1000, 10000, 100000]

    print("\nStudent Observation Table:")

    print("-" \* 80)

    print(f"{'Array Size':<12} {'Algorithm':<15} {'Target':<10} {'Found?':<8} {'Time (μs)':<12} {'Speedup'}")

    print("-" \* 80)

    for size in sizes:

        # Create sorted array

        sorted\_arr = list(range(size))

        # Search for element near the end (worst case for linear)

        target = size - 10

        # Linear Search

        idx\_linear, time\_linear = measure\_search\_time(linear\_search, sorted\_arr, target)

        # Binary Search

        idx\_binary, time\_binary = measure\_search\_time(binary\_search, sorted\_arr, target)

        speedup = time\_linear / time\_binary if time\_binary > 0 else 0

        print(f"{size:<12} {'Linear':<15} {target:<10} {'Yes' if idx\_linear != -1 else 'No':<8} {time\_linear:>10.2f}   -")

        print(f"{size:<12} {'Binary':<15} {target:<10} {'Yes' if idx\_binary != -1 else 'No':<8} {time\_binary:>10.2f}   {speedup:.1f}x faster")

        print()

    print("=" \* 80)

    print("AI EXPLANATION - When to Use Binary Search:")

    print("=" \* 80)

    print("""

BINARY SEARCH IS PREFERABLE WHEN:

1. DATA IS SORTED (or can be sorted once and searched multiple times)

   - Sorting cost: O(n log n) one-time

   - Each search: O(log n) vs O(n)

   - Break-even: After ~2-3 searches on large datasets

2. LARGE DATASETS (n > 100)

   - Linear: 100,000 comparisons for array of 100,000

   - Binary: Only 17 comparisons for same array!

   - Difference grows exponentially with size

3. PERFORMANCE-CRITICAL APPLICATIONS

   - Databases, search engines, autocomplete

   - Real-time systems with strict latency requirements

   - APIs with high query rates

4. MULTIPLE SEARCHES ON SAME DATA

   - Sort once: O(n log n)

   - Search k times: k \* O(log n)

   - Total: O(n log n + k log n) << O(k \* n) for large k

LINEAR SEARCH IS PREFERABLE WHEN:

1. UNSORTED DATA with infrequent searches

   - Sorting overhead not worth it

2. SMALL DATASETS (n < 50)

   - Constant factors dominate

   - Simpler code, less overhead

3. DATA CHANGES FREQUENTLY

   - Maintaining sorted order is expensive

   - Insertion/deletion in sorted array: O(n)

PERFORMANCE COMPARISON:

- Array size 1,000: Binary is ~10x faster

- Array size 10,000: Binary is ~100x faster

- Array size 100,000: Binary is ~1,000x faster

- Array size 1,000,000: Binary is ~10,000x faster!

CONCLUSION: For sorted data and large datasets, Binary Search is dramatically faster.

""")

    print("=" \* 80)

**Output:**

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**Explanation:**

In this task, we compare two fundamental search algorithms:

* **Linear Search:** A simple approach that checks each element one by one.
* **Binary Search:** A more efficient method that works on sorted arrays by dividing the search space in half each time.

We analysed:

* How they work
* Their time and space complexities
* Real-world use cases
* Performance on large data

**How Each Algorithm Works**

**Linear Search:**

* Start from the first element.
* Check each element sequentially until you find the target or reach the end.
* Works on **any array**, whether **sorted or unsorted**.

**Binary Search:**

* Only works on **sorted arrays**.
* Repeatedly divide the array into halves and check the middle element.
* Eliminate half of the array from consideration each time.

**Time and Space Complexity**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** |
| --- | --- | --- | --- | --- |
| Linear Search | O(1) | O(n) | O(n) | O(1) |
| Binary Search | O(1) | O(log n) | O(log n) | O(1) |

* **Best Case for Both**: If the target is found at the first (or middle) position.
* **Worst Case**:
  + Linear Search: Must scan the entire array.
  + Binary Search: Logarithmic performance due to halving the search space.

**Performance Testing (Summary)**

We tested both algorithms on increasing array sizes (100, 1,000, 10,000, 100,000).

**Results**:

* On small arrays, both perform similarly.
* On larger arrays, Binary Search is **dramatically faster**:
  + For 10,000 elements: Binary Search is ~100x faster.
  + For 100,000 elements: Binary Search is ~1,000x faster.
* Binary Search provides exponential performance gains as data size increases.

**When to Use Each Algorithm**

**🔷 Binary Search (Preferred When):**

1. **Array is sorted** (or can be sorted once for reuse).
2. **Large datasets** (n > 100).
3. **Multiple searches** on the same dataset.
4. **Performance is critical**, e.g., in real-time apps, databases, search engines.

**🔶 Linear Search (Use When):**

1. **Data is unsorted**, and sorting is not worth the cost.
2. **Small datasets** (n < 50), where the difference is negligible.
3. **Searches are infrequent**, and code simplicity is preferred.
4. **Data changes frequently**, making sorting expensive.

**💡 Real-World Analogy**

* **Linear Search**: Like flipping through every page of a book to find a word.
* **Binary Search**: Like using a dictionary — you go to the middle, check the word, and narrow down the section.

**Conclusion**

* Binary Search is significantly more efficient **but only works on sorted arrays**.
* Linear Search is simple and flexible but becomes inefficient as dataset size increases.
* Understanding both is essential, and choosing the right one depends on the data and context.

**TASK-4:**

**Prompt:** I have partially implemented recursive Quick Sort and Merge Sort functions in Python. Please complete the missing logic for both algorithms and add detailed docstrings explaining the parameters, return values, and functionality.

Also, provide an explanation of the average, best, and worst-case time complexities for both algorithms.

**Code:**

A screen shot of a computer screen

AI-generated content may be incorrect.

**Output:**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**Explanation:**

**1. About Sort and Merge Sort:**

* **Quick Sort** and **Merge Sort** are both efficient, comparison-based sorting algorithms that use **recursion** and the **divide-and-conquer** strategy.
* They break the problem (unsorted list) into smaller subproblems, sort those, and then combine the results.

**2. How Each Algorithm Works**

**Quick Sort:**

* Selects a **pivot** element (commonly the middle element).
* Partitions the list into three parts:
  + Elements **less than** the pivot.
  + Elements **equal to** the pivot.
  + Elements **greater than** the pivot.
* Recursively sorts the “less than” and “greater than” sublists.
* Combines them into a sorted list.

**Merge Sort:**

* Divides the list into two halves recursively until each sublist contains one element.
* Merges the sublists back together in sorted order.
* The **merge** step is crucial: it takes two sorted lists and combines them into a single sorted list.

**3. Time Complexity**

| **Algorithm** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** |
| --- | --- | --- | --- | --- |
| Quick Sort | O(n log n) | O(n log n) | O(n²) | O(log n) (due to recursion) |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) | O(n) (needs extra space) |

* **Quick Sort** is generally faster in practice but can degrade to quadratic time if the pivot choice is poor (like always picking smallest or largest element in sorted data).
* **Merge Sort** has guaranteed O(n log n) time regardless of input but requires extra memory to merge lists.

**4. Stability and Memory Use**

* **Merge Sort** is **stable**, meaning it maintains the relative order of equal elements.
* **Quick Sort** is generally **not stable**.
* Merge Sort requires additional memory proportional to the list size (for merging).
* Quick Sort is usually **in-place** in optimized implementations (though the version shown here is not in-place but easier to understand).

**5. Performance on Different Inputs**

| **Input Type** | **Quick Sort** | **Merge Sort** |
| --- | --- | --- |
| Random | Very efficient | Efficient |
| Already Sorted | Worst-case (O(n²)) | Still O(n log n) |
| Reverse Sorted | Worst-case (O(n²)) | Still O(n log n) |

**6. When to Use Which?**

* **Quick Sort**: When you want generally faster sorting with low memory overhead and your data is random or you implement a good pivot strategy (e.g., randomized pivot).
* **Merge Sort**: When you need guaranteed performance, stable sorting, or are working with linked lists or large data that doesn’t fit into memory all at once (external sorting).

**7. Summary**

* Both are powerful recursive sorting algorithms.
* Quick Sort is often faster but less predictable.
* Merge Sort is predictable and stable but uses extra space.
* Understanding both helps in choosing the right algorithm depending on the problem constraints.

**TASK-5:**

**Prompt:** I’ve written a brute-force algorithm to find duplicates in a list with O(n²) time complexity. Can you optimize this algorithm for better performance, ideally O(n)?  
Please return the optimized version with a docstring and explain how the time complexity was improved.

**Code:**

**A screen shot of a computer program

AI-generated content may be incorrect.**

**Output:**

**A black background with white text

AI-generated content may be incorrect.**

**Explanation:**

**1. The Problem**

You want to find all duplicate elements in a list.

**2. Naive (Brute Force) Approach**

* How it works:
  + Check every possible pair of elements.
  + If two elements are equal and not already recorded as duplicates, add to duplicates list.
* Time Complexity:
  + O(n²) because it compares each element with every other element.
* Space Complexity:
  + O(1) extra space (only stores duplicates).
* Drawbacks:
  + Very slow for large datasets because the number of comparisons grows quadratically.

**3. Optimized Approach Using Sets**

* How it works:
  + Use a set called seen to track elements you've encountered.
  + Iterate through the list once:
    - If an element is in seen, it’s a duplicate — add it to the duplicates set.
    - Otherwise, add it to seen.
* Time Complexity:
  + O(n) because each element is processed once.
* Space Complexity:
  + O(n) to store the seen elements and duplicates.
* Advantages:
  + Much faster on large datasets.
  + Simple and clean code.

**4. Performance Comparison**

* For large input sizes (thousands or more), the optimized set-based method can be orders of magnitude faster.
* The brute force method becomes impractical as n grows.
* The optimized method trades extra memory usage for speed, which is usually worthwhile.

**5. Summary**

| **Aspect** | **Brute Force** | **Optimized (Set)** |
| --- | --- | --- |
| **Time Complexity** | **O(n²)** | **O(n)** |
| **Space Complexity** | **O(1)** | **O(n)** |
| **Speed** | **Slow for large datasets** | **Fast and scalable** |
| **Use Case** | **Very small lists** | **Large or medium datasets** |

**6. Conclusion**

Using sets or dictionaries to keep track of seen elements is a classic and effective optimization technique that reduces a nested loop problem into a linear time solution, dramatically improving performance for large inputs.