SORTING AND SEARCHING ALGORITHMS

Assignment 12

Student Name:	Mahanth Arya
Hall Ticket Number:	2303A54028
Batch:	47A
Course:	AI Assistant Coding
Assignment Number:	12
Date:	February 2026

TABLE OF CONTENTS

• Task 1: Student Records Sorting for Placement Drive
• Task 2: Bubble Sort with AI Comments and Complexity Analysis
• Task 3: Quick Sort and Merge Sort Comparison
• Task 4: Inventory Management System
• Task 5: Real-Time Stock Data Sorting & Searching
• Complete Python Code Listing
• Conclusion and Summary

TASK 1: STUDENT RECORDS SORTING FOR PLACEMENT DRIVE

Scenario

SR University's Training and Placement Cell needs to shortlist candidates efficiently during campus placements. Student records must be sorted by CGPA in descending order.

Objective

• Use GitHub Copilot to generate a program that stores student records (Name, Roll Number, CGPA)
• Implement Quick Sort and Merge Sort algorithms
• Measure and compare runtime performance for large datasets
• Display top 10 students based on CGPA

Code Implementation

```
# Student Class
class Student:
    def __init__(self, name: str, roll_number: str, cgpa: float):
        self.name = name
        self.roll_number = roll_number
        self.cgpa = cgpa
    def __repr__(self):
        return f"Student(name='{self.name}', roll='{self.roll_number}', cgpa={self.cgpa})"
# Quick Sort Implementation
def quick_sort_students(students: List[Student], low: int, high: int) -> None:
    if low < high:
        pivot_index = partition(students, low, high)
        quick_sort_students(students, low, pivot_index - 1)
        quick_sort_students(students, pivot_index + 1, high)
def partition(students: List[Student], low: int, high: int) -> int:
    pivot = students[high].cgpa
    i = low - 1
    for j in range(low, high):
        if students[j].cgpa >= pivot:  # Descending order
            i += 1
            students[i], students[j] = students[j], students[i]
    students[i + 1], students[high] = students[high], students[i + 1]
    return i + 1
# Merge Sort Implementation
def merge_sort_students(students: List[Student]) -> List[Student]:
    if len(students) <= 1:
        return students
    mid = len(students) // 2
    left = merge_sort_students(students[:mid])
    right = merge_sort_students(students[mid:])
    return merge(left, right)
def merge(left: List[Student], right: List[Student]) -> List[Student]:
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i].cgpa >= right[j].cgpa:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# Display Top Students
def display_top_students(students: List[Student], n: int = 10) -> None:
    print(f"TOP {n} STUDENTS FOR PLACEMENT DRIVE")
    for rank, student in enumerate(students[:n], 1):
        print(f"{rank}. {student.name} - {student.cgpa}")
```

Performance Comparison Results

Algorithm	Time (100 students)	Space Complexity	Stable	Best For
Quick Sort	0.000051 seconds	O(log n)	No	In-place sorting
Merge Sort	0.000104 seconds	O(n)	Yes	Guaranteed performance

Sample Output - Top 10 Students

Rank	Name	Roll Number	CGPA
1	Arjun Mehta	2303A54087	9.81
2	Aditya Rao	2303A54077	9.79
3	Ishita Patel	2303A54088	9.79
4	Arjun Nair	2303A54046	9.65
5	Amit Nair	2303A54011	9.58

TASK 2: BUBBLE SORT WITH AI COMMENTS

Objective

Write a Python implementation of Bubble Sort with AI-generated inline comments explaining key logic (swapping, passes, termination) and provide time complexity analysis.

Code Implementation with AI Comments

```
def bubble_sort(arr: List[int]) -> List[int]:
    """
    Bubble Sort Algorithm with AI-Generated Comments
    Time Complexity:
        Best Case: O(n) - when array is already sorted
        Average Case: O(n²) - random arrangement
        Worst Case: O(n²) - reverse sorted array
    Space Complexity: O(1) - in-place sorting
    """
    n = len(arr)
    # Outer loop: controls the number of passes through the array
    # We need at most (n-1) passes to sort n elements
    for i in range(n):
        swapped = False  # Optimization flag
        # Inner loop: performs comparisons and swaps in current pass
        # With each pass, the last i elements are already sorted
        for j in range(0, n - i - 1):
            # Compare adjacent elements
            if arr[j] > arr[j + 1]:
                # Swap elements using Python's tuple unpacking
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # Termination: if no swaps occurred, array is sorted
        if not swapped:
            break
    return arr
```

Time Complexity Analysis Results

Array Size	Best Case (sorted)	Average Case (random)	Worst Case (reverse)
100	0.000006 s - O(n)	0.000235 s - O(n²)	0.000324 s - O(n²)
500	0.000021 s - O(n)	0.006592 s - O(n²)	0.008303 s - O(n²)
1000	0.000051 s - O(n)	0.030636 s - O(n²)	0.037523 s - O(n²)
2000	0.000120 s - O(n)	0.136317 s - O(n²)	0.162284 s - O(n²)

TASK 3: QUICK SORT AND MERGE SORT COMPARISON

Objective

Implement Quick Sort and Merge Sort using recursion, complete missing logic with AI assistance, add docstrings, and compare both algorithms on random, sorted, and reverse-sorted lists.

Code Implementation

```
def quick_sort_recursive(arr: List[int]) -> List[int]:
    """
    Recursive Quick Sort Implementation
    Complexity:
        Best/Average: O(n log n) - balanced partitioning
        Worst: O(n²) - unbalanced partitioning (already sorted)
    Space: O(log n) - recursion stack
    """
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort_recursive(left) + middle + quick_sort_recursive(right)
def merge_sort_recursive(arr: List[int]) -> List[int]:
    """
    Recursive Merge Sort Implementation
    Complexity:
        All Cases: O(n log n) - guaranteed performance
    Space: O(n) - requires additional space for merging
    """
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort_recursive(arr[:mid])
    right = merge_sort_recursive(arr[mid:])
    return merge_arrays(left, right)
def merge_arrays(left: List[int], right: List[int]) -> List[int]:
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

Performance Comparison on Different Array Types

Array Type	Array Size	Quick Sort (sec)	Merge Sort (sec)	Winner
Random	1000	0.000987	0.001191	Quick Sort
Sorted	1000	0.000553	0.000800	Quick Sort
Reverse	1000	0.000557	0.000809	Quick Sort
Random	10000	0.013069	0.015935	Quick Sort
Sorted	10000	0.007503	0.009660	Quick Sort
Reverse	10000	0.007267	0.011713	Quick Sort

AI-Generated Complexity Explanation

Quick Sort: Best/Average O(n log n) with good pivot selection. Worst O(n^2) when pivot is always smallest/largest (e.g., sorted array). In-place algorithm with O(log n) space.
Merge Sort: Guaranteed O(n log n) in all cases. Stable sorting algorithm. Requires O(n) extra space for merging. Preferred when stability and predictable performance are required.

TASK 4: INVENTORY MANAGEMENT SYSTEM

Scenario

A retail store's inventory system contains thousands of products with attributes: product ID, name, price, and stock quantity. Store staff need to quickly search for products and sort them for analysis.

AI-Recommended Algorithms

Operation	Recommended Algorithm	Time Complexity	Justification
Search by Product ID	Binary Search	O(log n)	IDs are unique and can be pre-sorted for fast lookup
Search by Name	Hash Map (Dictionary)	O(1) average	Instant lookup for frequent name-based searches
Sort by Price	Quick Sort	O(n log n) avg	In-place sorting, efficient for numerical data
Sort by Stock Quantity	Merge Sort	O(n log n) all	Stable sort, predictable performance for reports

Code Implementation

```
class Product:
    def __init__(self, product_id: str, name: str, price: float, stock_quantity: int):
        self.product_id = product_id
        self.name = name
        self.price = price
        self.stock_quantity = stock_quantity
# Binary Search for Product ID (O(log n))
def binary_search_by_id(products: List[Product], target_id: str) -> int:
    left, right = 0, len(products) - 1
    while left <= right:
        mid = (left + right) // 2
        if products[mid].product_id == target_id:
            return mid
        elif products[mid].product_id < target_id:
            left = mid + 1
        else:
            right = mid - 1
    return -1
# Hash Map for Product Name (O(1))
def hash_search_by_name(products: List[Product]) -> Dict[str, Product]:
    return {product.name.lower(): product for product in products}
# Quick Sort by Price
def quick_sort_by_price(products: List[Product], low: int, high: int, ascending: bool = True):
    if low < high:
        pivot_index = partition_by_price(products, low, high, ascending)
        quick_sort_by_price(products, low, pivot_index - 1, ascending)
        quick_sort_by_price(products, pivot_index + 1, high, ascending)
# Merge Sort by Stock Quantity
def merge_sort_by_quantity(products: List[Product], ascending: bool = True):
    if len(products) <= 1:
        return products
    mid = len(products) // 2
    left = merge_sort_by_quantity(products[:mid], ascending)
    right = merge_sort_by_quantity(products[mid:], ascending)
    return merge_by_quantity(left, right, ascending)
```

Performance Results (1000 products)

- **Binary Search:** 0.00000596 seconds per search
- **Hash Map Search:** 0.00000215 seconds per search (2.8x faster)
- **Quick Sort by Price:** ~0.001 seconds
- **Merge Sort by Quantity:** ~0.001 seconds

TASK 5: REAL-TIME STOCK DATA SORTING & SEARCHING

Scenario

An AI-powered FinTech Lab at SR University is building a tool for analyzing stock price movements. Requirements: quickly sort stocks by daily gain/loss and search for specific stock symbols efficiently.

Code Implementation

```
class Stock:
    def __init__(self, symbol: str, opening_price: float, closing_price: float):
        self.symbol = symbol
        self.opening_price = opening_price
        self.closing_price = closing_price
        # Calculate percentage change
        self.percentage_change = ((closing_price - opening_price) / opening_price) * 100
# Heap Sort for Stock Ranking (O(n log n))
def heap_sort_stocks(stocks: List[Stock], ascending: bool = False):
    n = len(stocks)
    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(stocks, n, i, ascending)
    # Extract elements from heap
    for i in range(n - 1, 0, -1):
        stocks[0], stocks[i] = stocks[i], stocks[0]
        heapify(stocks, i, 0, ascending)
    return stocks
def heapify(stocks: List[Stock], n: int, i: int, ascending: bool):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n:
        condition = stocks[left].percentage_change > stocks[largest].percentage_change if not ascending else stocks[left].pe
        if condition:
            largest = left
    if right < n:
        condition = stocks[right].percentage_change > stocks[largest].percentage_change if not ascending else stocks[right]
        if condition:
            largest = right
    if largest != i:
        stocks[i], stocks[largest] = stocks[largest], stocks[i]
        heapify(stocks, n, largest, ascending)
# Hash Map for Stock Symbol Lookup (O(1))
def create_stock_hashmap(stocks: List[Stock]):
    return {stock.symbol: stock for stock in stocks}
```

Performance Comparison: Custom vs Built-in Functions

Operation	Custom Implementation	Python Built-in	Result
Sorting (500 stocks)	Heap Sort: 0.000715s	sorted(): 0.000084s	Built-in 8.5x faster
Searching (10,000 lookups)	Hash Map: 0.000302s	Linear: 0.051909s	Hash Map 172x faster

Trade-offs Analysis

Algorithm	Advantages	Disadvantages
Heap Sort	✓ Consistent O(n log n) ✓ In-place (low memory) ✓ No worst case O(n^2)	✗ Not stable ✗ Slower than TimSort ✗ Complex implementation
Python sorted()	✓ Highly optimized TimSort ✓ Stable sorting ✓ Easy to use	✗ Requires O(n) space ✗ Black box algorithm

Hash Map	✓ O(1) average lookup ✓ Efficient for searches ✓ Flexible keys	✗ O(n) space required ✗ No ordering ✗ Hash collisions possible

COMPLETE PYTHON CODE LISTING

All Tasks - Complete Implementation

```
"""
SORTING AND SEARCHING ALGORITHMS ASSIGNMENT
Student: Mahanth Arya
Hall Ticket: 2303A54028
Batch: 47A
"""
import random
import time
from typing import List, Dict
# ============================================================================
# TASK 1: STUDENT RECORDS SORTING FOR PLACEMENT DRIVE
# ============================================================================
class Student:
    def __init__(self, name: str, roll_number: str, cgpa: float):
        self.name = name
        self.roll_number = roll_number
        self.cgpa = cgpa
    def __repr__(self):
        return f"Student(name='{self.name}', roll='{self.roll_number}', cgpa={self.cgpa})"
    def __str__(self):
        return f"{self.name:20s} | {self.roll_number:15s} | CGPA: {self.cgpa:.2f}"
def quick_sort_students(students: List[Student], low: int, high: int) -> None:
    if low < high:
        pivot_index = partition(students, low, high)
        quick_sort_students(students, low, pivot_index - 1)
        quick_sort_students(students, pivot_index + 1, high)
def partition(students: List[Student], low: int, high: int) -> int:
    pivot = students[high].cgpa
    i = low - 1
    for j in range(low, high):
        if students[j].cgpa >= pivot:
            i += 1
            students[i], students[j] = students[j], students[i]
    students[i + 1], students[high] = students[high], students[i + 1]
    return i + 1
def merge_sort_students(students: List[Student]) -> List[Student]:
    if len(students) <= 1:
        return students
    mid = len(students) // 2
    left_half = merge_sort_students(students[:mid])
    right_half = merge_sort_students(students[mid:])
    return merge(left_half, right_half)
def merge(left: List[Student], right: List[Student]) -> List[Student]:
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i].cgpa >= right[j].cgpa:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
def display_top_students(students: List[Student], n: int = 10) -> None:
    print(f"\n{'='*60}")
    print(f"TOP {n} STUDENTS FOR PLACEMENT DRIVE")
    print(f"{'='*60}")
    print(f"{'Rank':<6}{'Name':<20}{'Roll Number':<15}{'CGPA':<10}")
    print(f"{'-'*60}")
    for rank, student in enumerate(students[:n], 1):
        print(f"{rank:<6}{student.name:<20}{student.roll_number:<15}{student.cgpa:<10.2f}")
    print(f"{'='*60}\n")
def compare_sorting_performance(students: List[Student]) -> Dict[str, float]:
    results = {}
    students_copy1 = students.copy()
    start_time = time.time()
    quick_sort_students(students_copy1, 0, len(students_copy1) - 1)
    results['quick_sort'] = time.time() - start_time
    students_copy2 = students.copy()
```

```
    start_time = time.time()
    sorted_students = merge_sort_students(students_copy2)
    results['merge_sort'] = time.time() - start_time
    return results, students_copy1
# ================================================================================
# TASK 2: BUBBLE SORT WITH AI COMMENTS
# ================================================================================
def bubble_sort(arr: List[int]) -> List[int]:
    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
def analyze_bubble_sort_complexity():
    print("\n" + "="*70)
    print("BUBBLE SORT TIME COMPLEXITY ANALYSIS")
    print("="*70)
    sizes = [100, 500, 1000, 2000]
    for size in sizes:
        best_case = list(range(size))
        start = time.time()
        bubble_sort(best_case.copy())
        best_time = time.time() - start
        worst_case = list(range(size, 0, -1))
        start = time.time()
        bubble_sort(worst_case.copy())
        worst_time = time.time() - start
        avg_case = random.sample(range(size * 2), size)
        start = time.time()
        bubble_sort(avg_case.copy())
        avg_time = time.time() - start
        print(f"\nArray Size: {size}")
        print(f"  Best Case (Sorted):   {best_time:.6f} seconds - O(n)")
        print(f"  Average Case (Random): {avg_time:.6f} seconds - O(n²)")
        print(f"  Worst Case (Reverse):  {worst_time:.6f} seconds - O(n²)")
# ================================================================================
# TASK 3: QUICK SORT AND MERGE SORT COMPARISON
# ================================================================================
def quick_sort_recursive(arr: List[int]) -> List[int]:
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort_recursive(left) + middle + quick_sort_recursive(right)
def merge_sort_recursive(arr: List[int]) -> List[int]:
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort_recursive(arr[:mid])
    right = merge_sort_recursive(arr[mid:])
    return merge_arrays(left, right)
def merge_arrays(left: List[int], right: List[int]) -> List[int]:
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
def compare_sorting_algorithms():
    print("\n" + "="*70)
    print("QUICK SORT vs MERGE SORT COMPARISON")
    print("="*70)
    sizes = [1000, 5000, 10000]
    for size in sizes:
        print(f"\n{'Array Size: ' + str(size):^70}")
```

```
        print("-" * 70)
        print(f"{'Array Type':<20} {'Quick Sort (sec)':<25} {'Merge Sort (sec)':<25}")
        print("-" * 70)
        random_arr = random.sample(range(size * 2), size)
        start = time.time()
        quick_sort_recursive(random_arr.copy())
        quick_time = time.time() - start
        start = time.time()
        merge_sort_recursive(random_arr.copy())
        merge_time = time.time() - start
        print(f"{'Random':<20} {quick_time:<25.6f} {merge_time:<25.6f}")
        sorted_arr = list(range(size))
        start = time.time()
        quick_sort_recursive(sorted_arr.copy())
        quick_time = time.time() - start
        start = time.time()
        merge_sort_recursive(sorted_arr.copy())
        merge_time = time.time() - start
        print(f"{'Sorted':<20} {quick_time:<25.6f} {merge_time:<25.6f}")
        reverse_arr = list(range(size, 0, -1))
        start = time.time()
        quick_sort_recursive(reverse_arr.copy())
        quick_time = time.time() - start
        start = time.time()
        merge_sort_recursive(reverse_arr.copy())
        merge_time = time.time() - start
        print(f"{'Reverse Sorted':<20} {quick_time:<25.6f} {merge_time:<25.6f}")
# ============================================================================
# TASK 4: INVENTORY MANAGEMENT SYSTEM
# ============================================================================
class Product:
    def __init__(self, product_id: str, name: str, price: float, stock_quantity: int):
        self.product_id = product_id
        self.name = name
        self.price = price
        self.stock_quantity = stock_quantity
    def __repr__(self):
        return f"Product({self.product_id}, {self.name}, ${self.price:.2f}, Stock: {self.stock_quantity})"
def binary_search_by_id(products: List[Product], target_id: str) -> int:
    left, right = 0, len(products) - 1
    while left <= right:
        mid = (left + right) // 2
        if products[mid].product_id == target_id:
            return mid
        elif products[mid].product_id < target_id:
            left = mid + 1
        else:
            right = mid - 1
    return -1
def hash_search_by_name(products: List[Product]) -> Dict[str, Product]:
    return {product.name.lower(): product for product in products}
def quick_sort_by_price(products: List[Product], low: int, high: int, ascending: bool = True) -> None:
    if low < high:
        pivot_index = partition_by_price(products, low, high, ascending)
        quick_sort_by_price(products, low, pivot_index - 1, ascending)
        quick_sort_by_price(products, pivot_index + 1, high, ascending)
def partition_by_price(products: List[Product], low: int, high: int, ascending: bool) -> int:
    pivot = products[high].price
    i = low - 1
    for j in range(low, high):
        condition = products[j].price <= pivot if ascending else products[j].price >= pivot
        if condition:
            i += 1
            products[i], products[j] = products[j], products[i]
    products[i + 1], products[high] = products[high], products[i + 1]
    return i + 1
def merge_sort_by_quantity(products: List[Product], ascending: bool = True) -> List[Product]:
    if len(products) <= 1:
        return products
    mid = len(products) // 2
    left = merge_sort_by_quantity(products[:mid], ascending)
    right = merge_sort_by_quantity(products[mid:], ascending)
    return merge_by_quantity(left, right, ascending)
def merge_by_quantity(left: List[Product], right: List[Product], ascending: bool) -> List[Product]:
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        condition = left[i].stock_quantity <= right[j].stock_quantity if ascending else left[i].stock_quantity >= right[j].s
```

```
        if condition:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# ============================================================================
# TASK 5: STOCK DATA SORTING & SEARCHING
# ============================================================================
class Stock:
    def __init__(self, symbol: str, opening_price: float, closing_price: float):
        self.symbol = symbol
        self.opening_price = opening_price
        self.closing_price = closing_price
        self.percentage_change = ((closing_price - opening_price) / opening_price) * 100
    def __repr__(self):
        return f"Stock({self.symbol}, Change: {self.percentage_change:+.2f}%)"
def heap_sort_stocks(stocks: List[Stock], ascending: bool = False) -> List[Stock]:
    n = len(stocks)
    for i in range(n // 2 - 1, -1, -1):
        heapify(stocks, n, i, ascending)
    for i in range(n - 1, 0, -1):
        stocks[0], stocks[i] = stocks[i], stocks[0]
        heapify(stocks, i, 0, ascending)
    return stocks
def heapify(stocks: List[Stock], n: int, i: int, ascending: bool) -> None:
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n:
        condition = stocks[left].percentage_change > stocks[largest].percentage_change if not ascending else stocks[left].pe
        if condition:
            largest = left
    if right < n:
        condition = stocks[right].percentage_change > stocks[largest].percentage_change if not ascending else stocks[right]
        if condition:
            largest = right
    if largest != i:
        stocks[i], stocks[largest] = stocks[largest], stocks[i]
        heapify(stocks, n, largest, ascending)
def create_stock_hashmap(stocks: List[Stock]) -> Dict[str, Stock]:
    return {stock.symbol: stock for stock in stocks}
def compare_with_builtin(stocks: List[Stock]) -> None:
    print("\n" + "="*70)
    print("PERFORMANCE COMPARISON: CUSTOM vs BUILT-IN")
    print("="*70)
    stocks_copy1 = stocks.copy()
    start = time.time()
    heap_sort_stocks(stocks_copy1)
    heap_time = time.time() - start
    stocks_copy2 = stocks.copy()
    start = time.time()
    sorted(stocks_copy2, key=lambda x: x.percentage_change, reverse=True)
    builtin_time = time.time() - start
    print(f"\nSorting Performance ({len(stocks)} stocks):")
    print(f"  Heap Sort:          {heap_time:.6f} seconds")
    print(f"  Python sorted():    {builtin_time:.6f} seconds")
    print(f"  Difference:         {abs(heap_time - builtin_time):.6f} seconds")
    stock_map = create_stock_hashmap(stocks)
    search_symbol = stocks[len(stocks) // 2].symbol
    start = time.time()
    for _ in range(10000):
        result = stock_map.get(search_symbol)
    hash_time = time.time() - start
    start = time.time()
    for _ in range(10000):
        result = next((s for s in stocks if s.symbol == search_symbol), None)
    linear_time = time.time() - start
    print(f"\nSearching Performance (10,000 lookups):")
    print(f"  Hash Map (O(1)):    {hash_time:.6f} seconds")
    print(f"  Linear Search (O(n)): {linear_time:.6f} seconds")
    print(f"  Speedup:            {linear_time/hash_time:.2f}x faster")
# ============================================================================
# HELPER FUNCTIONS
# ============================================================================
```

```
def generate_sample_students(n: int = 100) -> List[Student]:
    first_names = ['Rahul', 'Priya', 'Amit', 'Sneha', 'Karan', 'Ananya', 'Vikram',
                   'Divya', 'Rohan', 'Kavya', 'Arjun', 'Riya', 'Aditya', 'Ishita']
    last_names = ['Sharma', 'Patel', 'Kumar', 'Singh', 'Reddy', 'Mehta', 'Gupta',
                  'Rao', 'Nair', 'Iyer']
    students = []
    for i in range(n):
        name = f"{random.choice(first_names)} {random.choice(last_names)}"
        roll = f"2303A540{i:02d}"
        cgpa = round(random.uniform(6.0, 10.0), 2)
        students.append(Student(name, roll, cgpa))
    return students
def generate_sample_products(n: int = 1000) -> List[Product]:
    categories = ['Electronics', 'Clothing', 'Books', 'Food', 'Toys']
    items = ['Laptop', 'Phone', 'Shirt', 'Novel', 'Chips', 'Game', 'Watch', 'Shoes']
    products = []
    for i in range(n):
        prod_id = f"PROD{i:04d}"
        name = f"{random.choice(categories)} - {random.choice(items)} {i}"
        price = round(random.uniform(10.0, 1000.0), 2)
        stock = random.randint(0, 500)
        products.append(Product(prod_id, name, price, stock))
    return products
def generate_sample_stocks(n: int = 500) -> List[Stock]:
    symbols = ['AAPL', 'GOOGL', 'MSFT', 'AMZN', 'TSLA', 'META', 'NVDA', 'NFLX',
               'AMD', 'INTC', 'ORCL', 'IBM', 'CSCO', 'ADBE', 'CRM']
    stocks = []
    for i in range(n):
        symbol = random.choice(symbols) + str(i)
        opening = round(random.uniform(50.0, 500.0), 2)
        closing = round(opening * random.uniform(0.95, 1.05), 2)
        stocks.append(Stock(symbol, opening, closing))
    return stocks
# ============================================================================
# MAIN EXECUTION
# ============================================================================
if __name__ == "__main__":
    print("="*70)
    print(" SORTING AND SEARCHING ALGORITHMS ASSIGNMENT")
    print(" Student: Mahanth Arya | Hall Ticket: 2303A54028 | Batch: 47A")
    print("="*70)
    # TASK 1
    print("\n\nTASK 1: STUDENT RECORDS SORTING")
    students = generate_sample_students(100)
    perf_results, sorted_students = compare_sorting_performance(students)
    print(f"\nPerformance Comparison:")
    print(f"  Quick Sort: {perf_results['quick_sort']:.6f} seconds")
    print(f"  Merge Sort: {perf_results['merge_sort']:.6f} seconds")
    display_top_students(sorted_students, 10)
    # TASK 2
    print("\n\nTASK 2: BUBBLE SORT")
    sample_array = [64, 34, 25, 12, 22, 11, 90]
    print(f"Original array: {sample_array}")
    sorted_array = bubble_sort(sample_array.copy())
    print(f"Sorted array:   {sorted_array}")
    analyze_bubble_sort_complexity()
    # TASK 3
    print("\n\nTASK 3: ALGORITHM COMPARISON")
    compare_sorting_algorithms()
    # TASK 4
    print("\n\nTASK 4: INVENTORY MANAGEMENT")
    products = generate_sample_products(1000)
    products_by_price = products.copy()
    quick_sort_by_price(products_by_price, 0, len(products_by_price) - 1, ascending=True)
    print(f"\nTop 5 cheapest products:")
    for p in products_by_price[:5]:
        print(f"  {p.name}: ${p.price:.2f}")
    # TASK 5
    print("\n\nTASK 5: STOCK DATA ANALYSIS")
    stocks = generate_sample_stocks(500)
    sorted_stocks = heap_sort_stocks(stocks.copy(), ascending=False)
    print("\nTop 10 Gainers:")
    print(f"{'Symbol':<15} {'Opening':<12} {'Closing':<12} {'Change':<12}")
    print("-" * 51)
    for stock in sorted_stocks[:10]:
        print(f"{stock.symbol:<15} ${stock.opening_price:<11.2f} ${stock.closing_price:<11.2f} {stock.percentage_change:>+1
    compare_with_builtin(stocks)
    print("\n" + "="*70)
```

```
print("ASSIGNMENT COMPLETED")
print("="*70)
```

CONCLUSION

Summary of Findings

This assignment explored various sorting and searching algorithms through practical applications using AI assistance (GitHub Copilot). Each task demonstrated different algorithm characteristics and their suitability for specific use cases.

Key Learnings

1. Algorithm Selection: Choice depends on data characteristics, memory constraints, and stability requirements.

2. Trade-offs: Faster algorithms may use more memory (Merge Sort O(n)), while in-place algorithms may have worse worst-case (Quick Sort $O(n^2)$).

3. Data Structures: Hash maps provide O(1) lookup but require O(n) space. Binary search offers O(log n) but needs sorted data.

4. AI Assistance: GitHub Copilot accelerated development and provided well-commented, production-ready code.

5. Built-in Optimizations: Python's sorted() uses TimSort, highly optimized for real-world patterns.

Performance Summary Table

Application	Best Algorithm	Reason
Placement Rankings	Quick Sort	Fast, in-place, efficient for CGPA sorting
Small Datasets	Bubble Sort	Simple implementation, good for n<100
General Purpose	Merge Sort	Guaranteed O(n log n), stable
Inventory Search	Hash Map + Binary	O(1) name lookup, O(log n) ID search
Stock Analysis	Heap Sort + Hash	Efficient ranking and instant symbol lookup

Acknowledgments

This assignment (Assignment 12) for AI Assistant Coding was completed with GitHub Copilot and AI-powered code generation tools, demonstrating how AI enhances learning and development productivity while maintaining deep understanding of algorithmic principles.