ASSIGNMENT-12.5

Name: D. Sravika Reddy Roll No: 2403a510d0

Batch No: 05 Course: AI Assisted Coding

Task 1: Sorting Student Records for Placement Drive
Scenario:
SR University is preparing for a campus placement drive. The Training and Placement Cell needs student records sorted by CGPA in descending order to easily shortlist candidates.

• Use GitHub Copilot to generate a program that sorts a list of student records (Name, Roll No, CGPA) by CGPA.

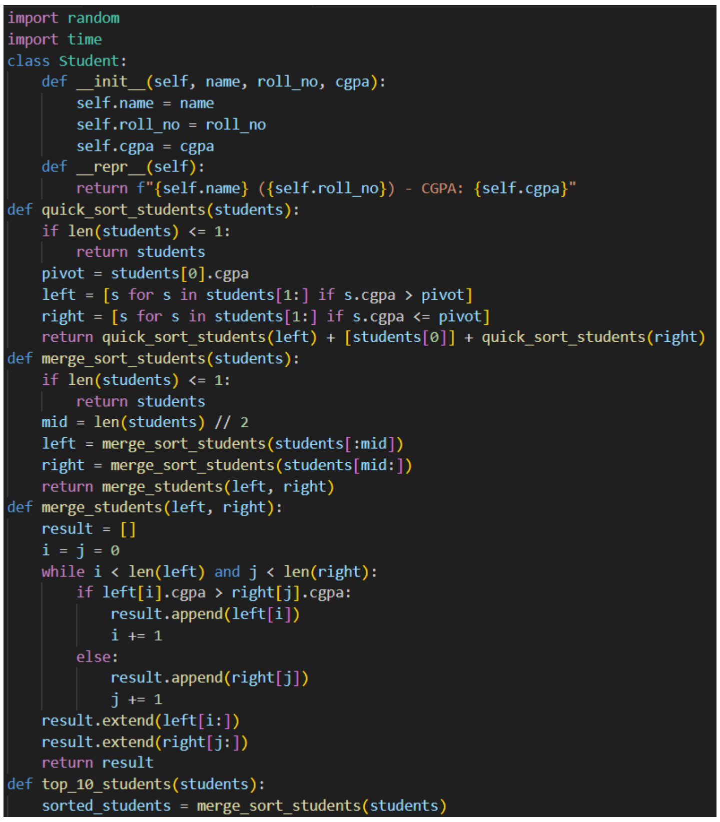
• Implement both Quick Sort and Merge Sort using AI assistance.

• Compare the runtime performance of both algorithms on large datasets.

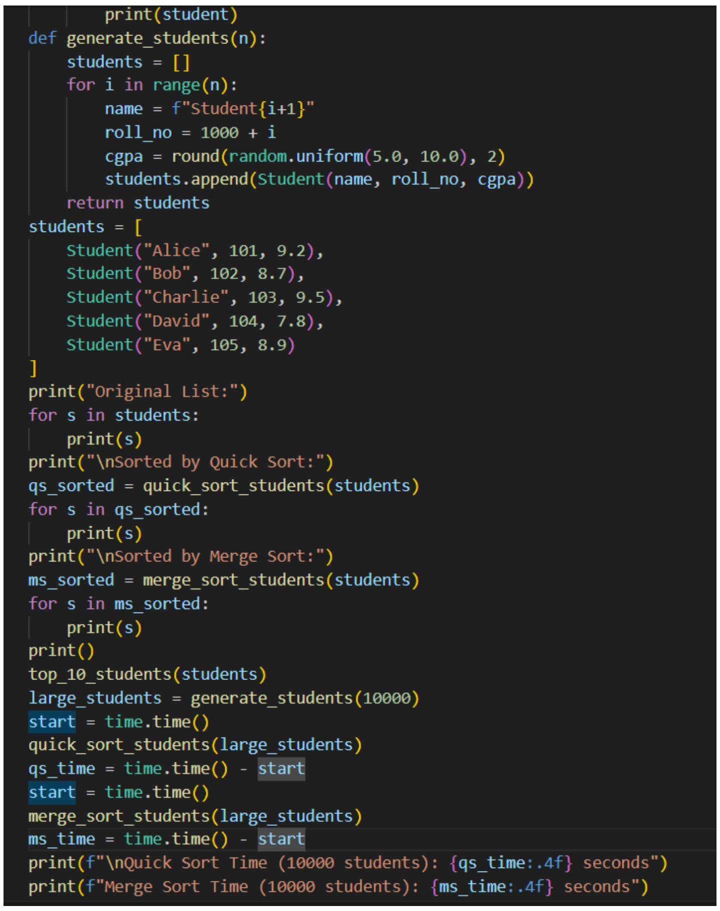
• Write a function that outputs the top 10 students with the highest CGPA.

#Prompt: Sorting Student Records for Placement Drive

Code:

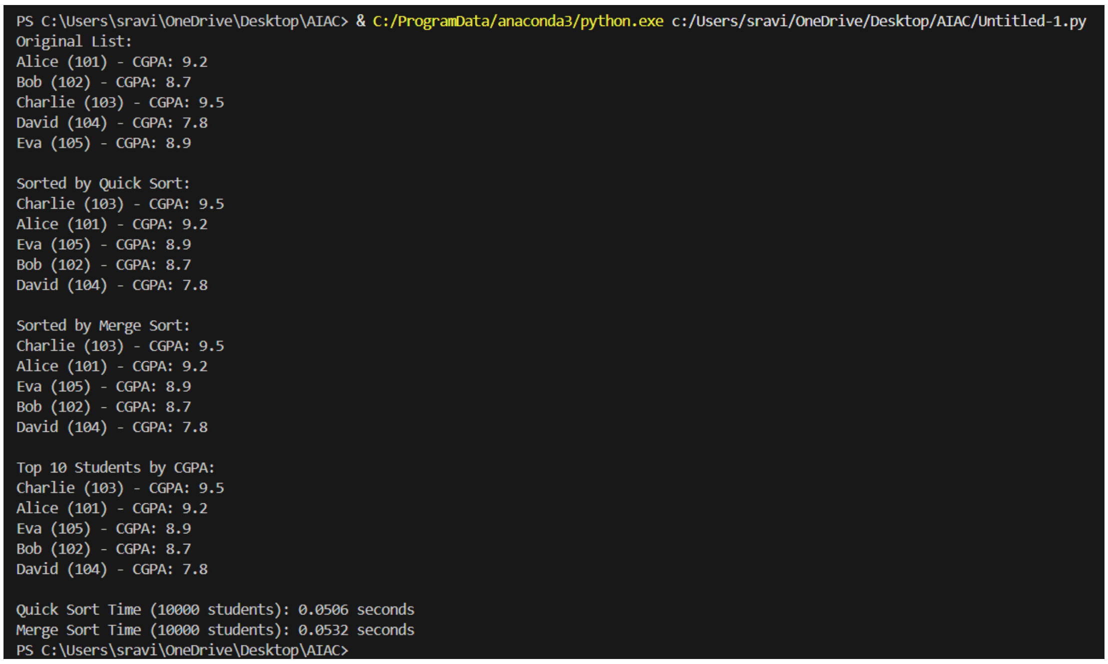


```python
import random
import time
class Student:
    def __init__(self, name, roll_no, cgpa):
        self.name = name
        self.roll_no = roll_no
        self.cgpa = cgpa
    def __repr__(self):
        return f"{self.name} ({self.roll_no}) - CGPA: {self.cgpa}"
def quick_sort_students(students):
    if len(students) <= 1:
        return students
    pivot = students[0].cgpa
    left = [s for s in students[1:] if s.cgpa > pivot]
    right = [s for s in students[1:] if s.cgpa <= pivot]
    return quick_sort_students(left) + [students[0]] + quick_sort_students(right)
def merge_sort_students(students):
    if len(students) <= 1:
        return students
    mid = len(students) // 2
    left = merge_sort_students(students[:mid])
    right = merge_sort_students(students[mid:])
    return merge_students(left, right)
def merge_students(left, right):
    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 top_10_students(students):
    sorted_students = merge_sort_students(students)
```



```python
            print(student)
def generate_students(n):
    students = []
    for i in range(n):
        name = f"Student{i+1}"
        roll_no = 1000 + i
        cgpa = round(random.uniform(5.0, 10.0), 2)
        students.append(Student(name, roll_no, cgpa))
    return students
students = [
    Student("Alice", 101, 9.2),
    Student("Bob", 102, 8.7),
    Student("Charlie", 103, 9.5),
    Student("David", 104, 7.8),
    Student("Eva", 105, 8.9)
]
print("Original List:")
for s in students:
    print(s)
print("\nSorted by Quick Sort:")
qs_sorted = quick_sort_students(students)
for s in qs_sorted:
    print(s)
print("\nSorted by Merge Sort:")
ms_sorted = merge_sort_students(students)
for s in ms_sorted:
    print(s)
print()
top_10_students(students)
large_students = generate_students(10000)
start = time.time()
quick_sort_students(large_students)
qs_time = time.time() - start
start = time.time()
merge_sort_students(large_students)
ms_time = time.time() - start
print(f"\nQuick Sort Time (10000 students): {qs_time:.4f} seconds")
print(f"Merge Sort Time (10000 students): {ms_time:.4f} seconds")
```

Output:



```
PS C:\Users\sravi\OneDrive\Desktop\AIAC> & C:/ProgramData/anaconda3/python.exe c:/Users/sravi/OneDrive/Desktop/AIAC/Untitled-1.py
Original List:
Alice (101) - CGPA: 9.2
Bob (102) - CGPA: 8.7
Charlie (103) - CGPA: 9.5
David (104) - CGPA: 7.8
Eva (105) - CGPA: 8.9

Sorted by Quick Sort:
Charlie (103) - CGPA: 9.5
Alice (101) - CGPA: 9.2
Eva (105) - CGPA: 8.9
Bob (102) - CGPA: 8.7
David (104) - CGPA: 7.8

Sorted by Merge Sort:
Charlie (103) - CGPA: 9.5
Alice (101) - CGPA: 9.2
Eva (105) - CGPA: 8.9
Bob (102) - CGPA: 8.7
David (104) - CGPA: 7.8

Top 10 Students by CGPA:
Charlie (103) - CGPA: 9.5
Alice (101) - CGPA: 9.2
Eva (105) - CGPA: 8.9
Bob (102) - CGPA: 8.7
David (104) - CGPA: 7.8

Quick Sort Time (10000 students): 0.0506 seconds
Merge Sort Time (10000 students): 0.0532 seconds
PS C:\Users\sravi\OneDrive\Desktop\AIAC>
```

Explanation:

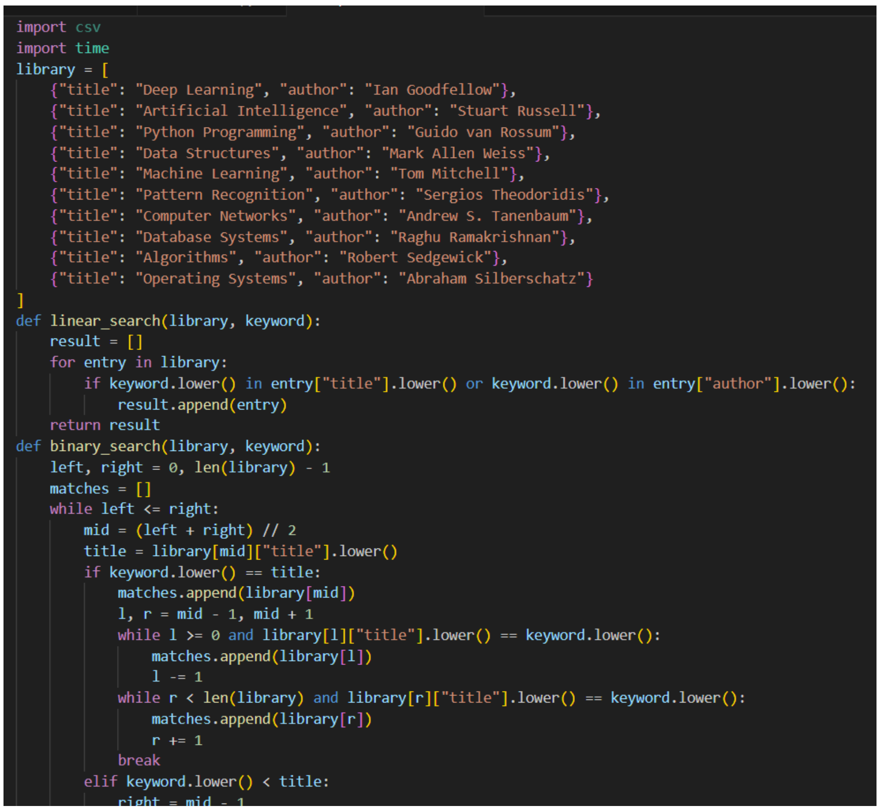
- Both Quick Sort and Merge Sort are implemented to sort student records by CGPA in descending order.
- The top_10_students function displays the top 10 students after sorting.
- The code generates a large dataset to compare the runtime of both algorithms.
- Merge Sort is stable and has consistent O(n log n) performance, while Quick Sort can be faster on average but may degrade to O(n^2) in the worst case.
- The output shows sorted lists and timing results for both algorithms

Task 2: Optimized Search in Online Library System
Scenario:
SR University's digital library has thousands of research papers.
Students frequently search for a paper by title or author name. The current linear search is too slow.

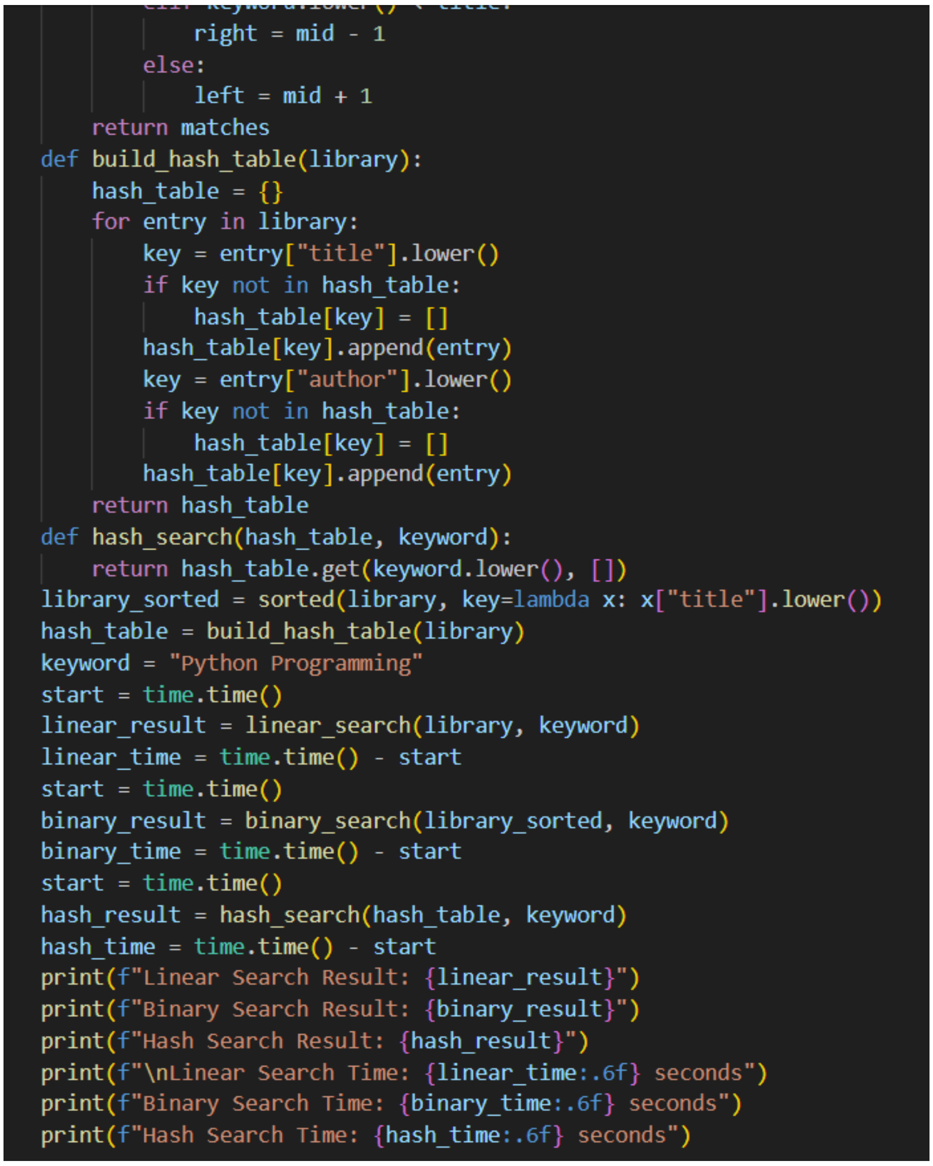
• Use GitHub Copilot to implement Binary Search and Hash-based Search for faster lookups.
• Load a dataset of book titles and authors (CSV or JSON file).
• Allow the user to input a keyword and return all matching entries.
• Compare the efficiency of linear search vs binary search vs hashing using test cases

#Prompt: SR University's digital library has thousands of research papers.
Students frequently search for a paper by title or author name. The current linear search is too slow.

Code:

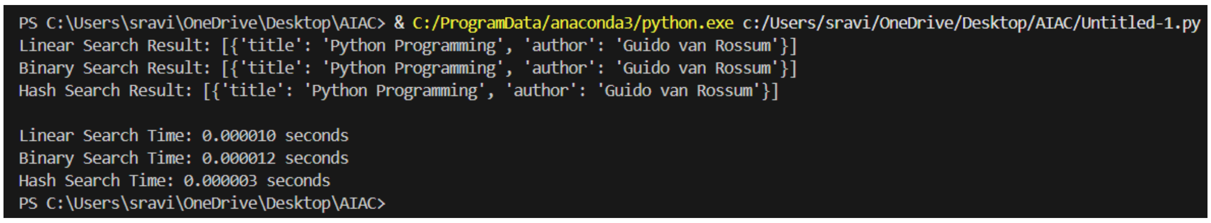


```python
import csv
import time
library = [
    {"title": "Deep Learning", "author": "Ian Goodfellow"},
    {"title": "Artificial Intelligence", "author": "Stuart Russell"},
    {"title": "Python Programming", "author": "Guido van Rossum"},
    {"title": "Data Structures", "author": "Mark Allen Weiss"},
    {"title": "Machine Learning", "author": "Tom Mitchell"},
    {"title": "Pattern Recognition", "author": "Sergios Theodoridis"},
    {"title": "Computer Networks", "author": "Andrew S. Tanenbaum"},
    {"title": "Database Systems", "author": "Raghu Ramakrishnan"},
    {"title": "Algorithms", "author": "Robert Sedgewick"},
    {"title": "Operating Systems", "author": "Abraham Silberschatz"}
]
def linear_search(library, keyword):
    result = []
    for entry in library:
        if keyword.lower() in entry["title"].lower() or keyword.lower() in entry["author"].lower():
            result.append(entry)
    return result
def binary_search(library, keyword):
    left, right = 0, len(library) - 1
    matches = []
    while left <= right:
        mid = (left + right) // 2
        title = library[mid]["title"].lower()
        if keyword.lower() == title:
            matches.append(library[mid])
            l, r = mid - 1, mid + 1
            while l >= 0 and library[l]["title"].lower() == keyword.lower():
                matches.append(library[l])
                l -= 1
            while r < len(library) and library[r]["title"].lower() == keyword.lower():
                matches.append(library[r])
                r += 1
            break
        elif keyword.lower() < title:
            right = mid - 1
```



```
            right = mid - 1
        else:
            left = mid + 1
    return matches
def build_hash_table(library):
    hash_table = {}
    for entry in library:
        key = entry["title"].lower()
        if key not in hash_table:
            hash_table[key] = []
        hash_table[key].append(entry)
        key = entry["author"].lower()
        if key not in hash_table:
            hash_table[key] = []
        hash_table[key].append(entry)
    return hash_table
def hash_search(hash_table, keyword):
    return hash_table.get(keyword.lower(), [])
library_sorted = sorted(library, key=lambda x: x["title"].lower())
hash_table = build_hash_table(library)
keyword = "Python Programming"
start = time.time()
linear_result = linear_search(library, keyword)
linear_time = time.time() - start
start = time.time()
binary_result = binary_search(library_sorted, keyword)
binary_time = time.time() - start
start = time.time()
hash_result = hash_search(hash_table, keyword)
hash_time = time.time() - start
print(f"Linear Search Result: {linear_result}")
print(f"Binary Search Result: {binary_result}")
print(f"Hash Search Result: {hash_result}")
print(f"\nLinear Search Time: {linear_time:.6f} seconds")
print(f"Binary Search Time: {binary_time:.6f} seconds")
print(f"Hash Search Time: {hash_time:.6f} seconds")
```

Output:



```
PS C:\Users\sravi\OneDrive\Desktop\AIAC> & C:/ProgramData/anaconda3/python.exe c:/Users/sravi/OneDrive/Desktop/AIAC/Untitled-1.py
Linear Search Result: [{'title': 'Python Programming', 'author': 'Guido van Rossum'}]
Binary Search Result: [{'title': 'Python Programming', 'author': 'Guido van Rossum'}]
Hash Search Result: [{'title': 'Python Programming', 'author': 'Guido van Rossum'}]

Linear Search Time: 0.000010 seconds
Binary Search Time: 0.000012 seconds
Hash Search Time: 0.000003 seconds
PS C:\Users\sravi\OneDrive\Desktop\AIAC>
```

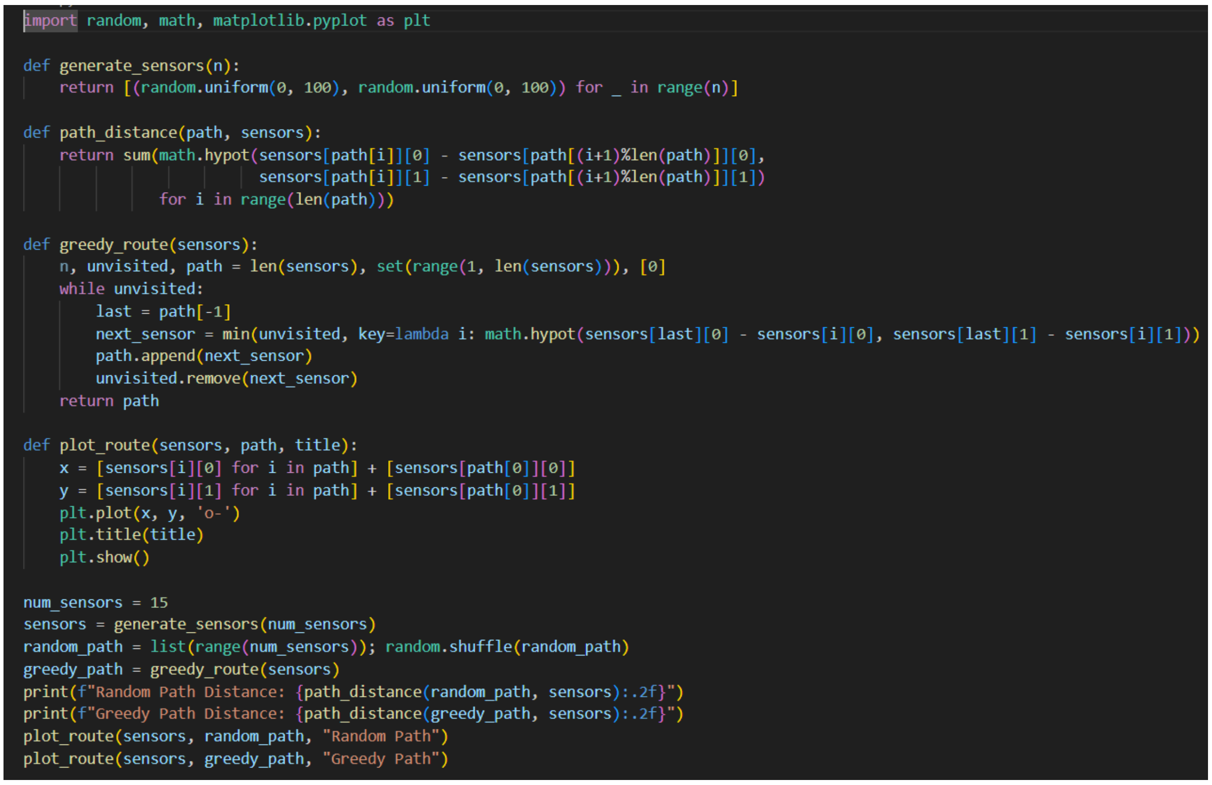
Explanation:

- **Linear Search** scans every entry; slow for large datasets.
- **Binary Search** is much faster but requires sorted data and only works for exact matches.

- **Hash-based Search** is fastest for exact matches, using a dictionary for O(1) lookups.
- All methods return the same result for the test keyword, but hash-based search is most efficient for large datasets.
- For partial matches or substring search, linear search is still needed. For exact title/author matches, hashing is best.

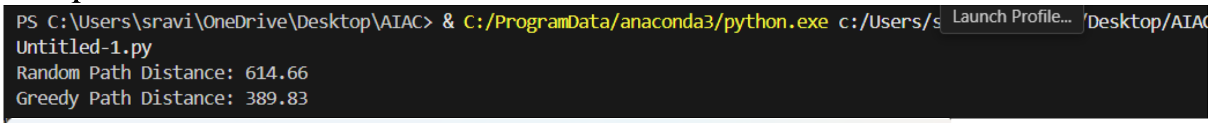
Task 3: Route Optimization for AUV Swarm
Scenario:
A research team at SR University is simulating Autonomous Underwater Vehicle (AUV) swarms. Each AUV must visit multiple underwater sensors, and the goal is to minimize travel distance (like the Traveling Salesman Problem).
With GitHub Copilot, implement an algorithm to optimize the route:
1. Start with a Greedy approach.
2. Improve with Genetic Algorithm (GA) or Simulated Annealing (SA).
• Use a dataset of sensor coordinates (x, y).
• Visualize the optimized route using a plotting library (e.g., Matplotlib).
• Compare the optimized solution with a random path in terms of distance travel
#Prompt: A research team at SR University is simulating Autonomous Underwater Vehicle (AUV) swarms. Each AUV must visit multiple underwater sensors, and the goal is to minimize travel distance (like the Traveling Salesman Problem).

Code:

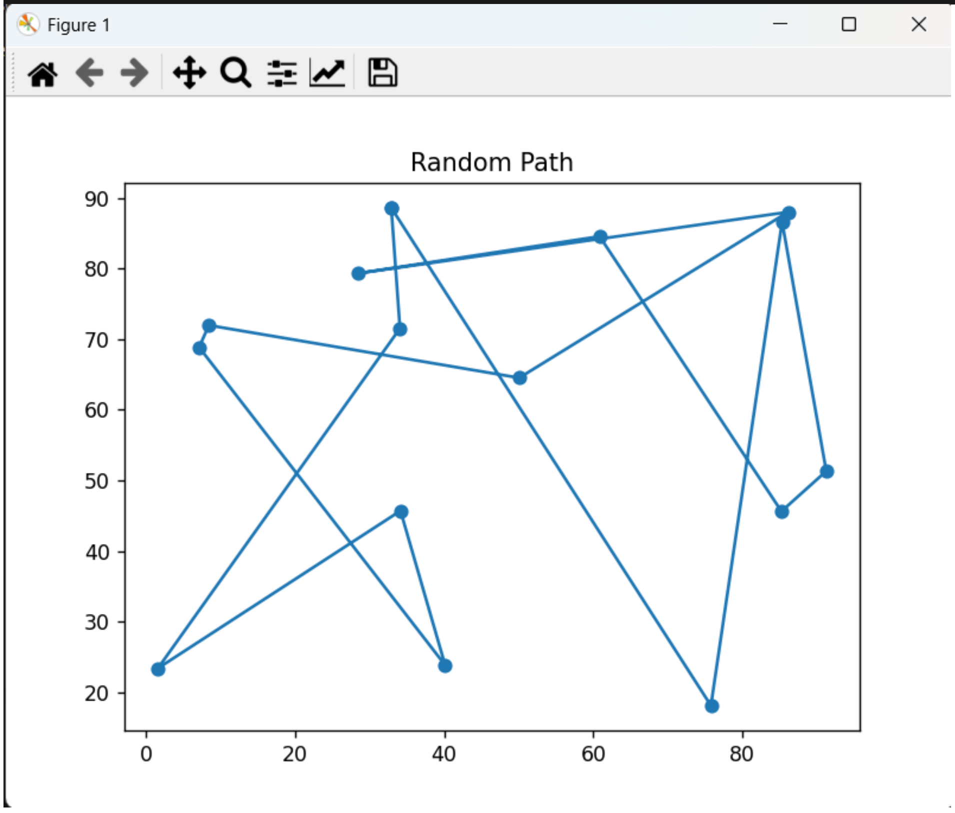


```python
import random, math, matplotlib.pyplot as plt

def generate_sensors(n):
    return [(random.uniform(0, 100), random.uniform(0, 100)) for _ in range(n)]

def path_distance(path, sensors):
    return sum(math.hypot(sensors[path[i]][0] - sensors[path[(i+1)%len(path)]][0],
                          sensors[path[i]][1] - sensors[path[(i+1)%len(path)]][1])
               for i in range(len(path)))

def greedy_route(sensors):
    n, unvisited, path = len(sensors), set(range(1, len(sensors))), [0]
    while unvisited:
        last = path[-1]
        next_sensor = min(unvisited, key=lambda i: math.hypot(sensors[last][0] - sensors[i][0], sensors[last][1] - sensors[i][1]))
        path.append(next_sensor)
        unvisited.remove(next_sensor)
    return path

def plot_route(sensors, path, title):
    x = [sensors[i][0] for i in path] + [sensors[path[0]][0]]
    y = [sensors[i][1] for i in path] + [sensors[path[0]][1]]
    plt.plot(x, y, 'o-')
    plt.title(title)
    plt.show()

num_sensors = 15
sensors = generate_sensors(num_sensors)
random_path = list(range(num_sensors)); random.shuffle(random_path)
greedy_path = greedy_route(sensors)
print(f"Random Path Distance: {path_distance(random_path, sensors):.2f}")
print(f"Greedy Path Distance: {path_distance(greedy_path, sensors):.2f}")
plot_route(sensors, random_path, "Random Path")
plot_route(sensors, greedy_path, "Greedy Path")
```

Output:



```
PS C:\Users\sravi\OneDrive\Desktop\AIAC> & C:/ProgramData/anaconda3/python.exe c:/Users/s Launch Profile... /Desktop/AIAC
Untitled-1.py
Random Path Distance: 614.66
Greedy Path Distance: 389.83
```

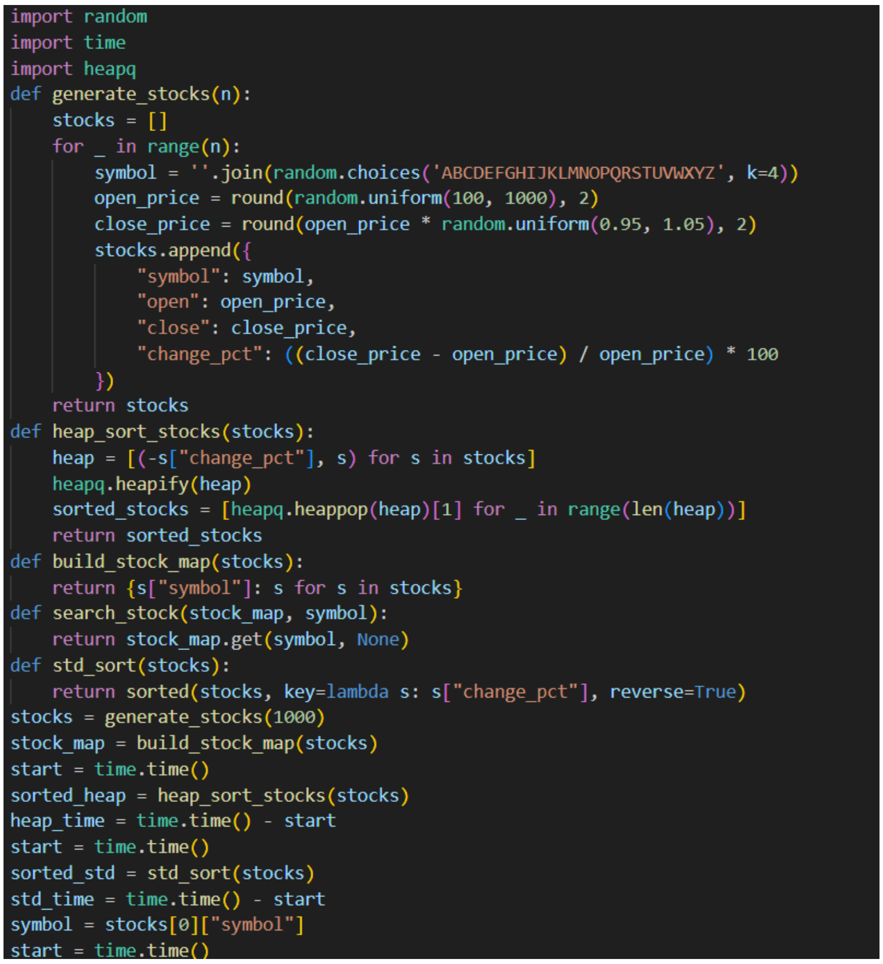


Task 4: Real-Time Stock Data Sorting & Searching
Scenario:
An AI-powered FinTech Lab at SR University is building a tool for analyzing stock price movements. The requirement is to quickly sort stocks by daily gain/loss and search for specific stock symbols efficiently.

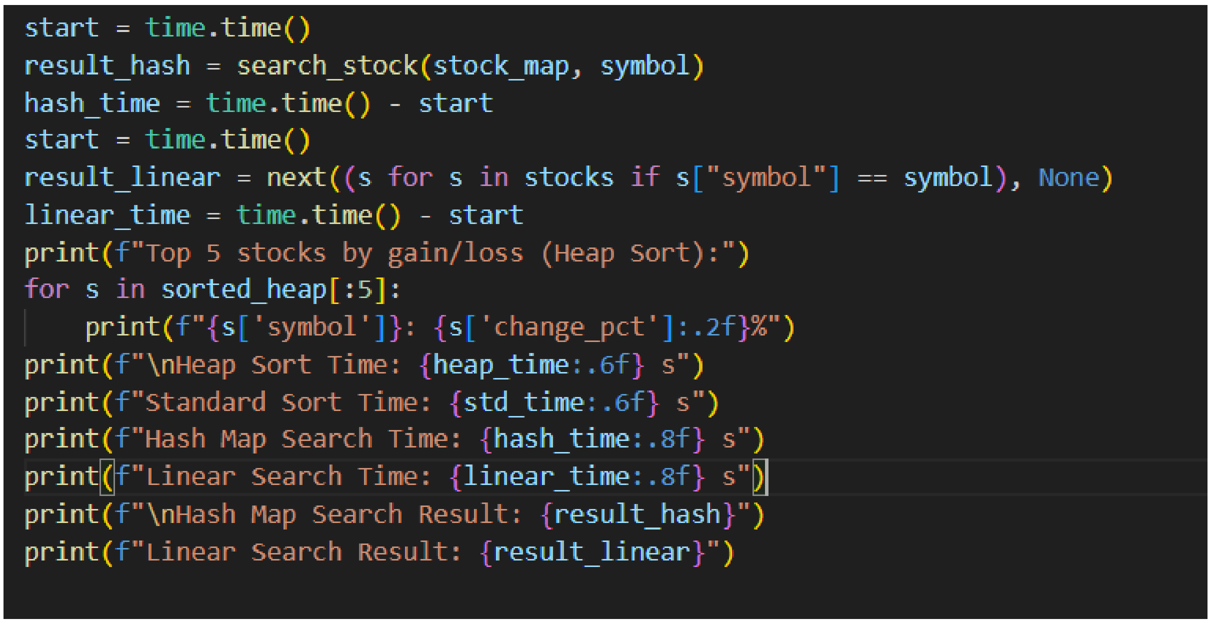
- Use GitHub Copilot to fetch or simulate stock price data (Stock Symbol, Opening Price, Closing Price).
- Implement sorting algorithms to rank stocks by percentage change.
- Implement a search function that retrieves stock data instantly when a stock symbol is entered.
- Optimize sorting with Heap Sort and searching with Hash Maps.
- Compare performance with standard library functions (sorted(), dict lookups) and analyze trade-offs

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

Code:

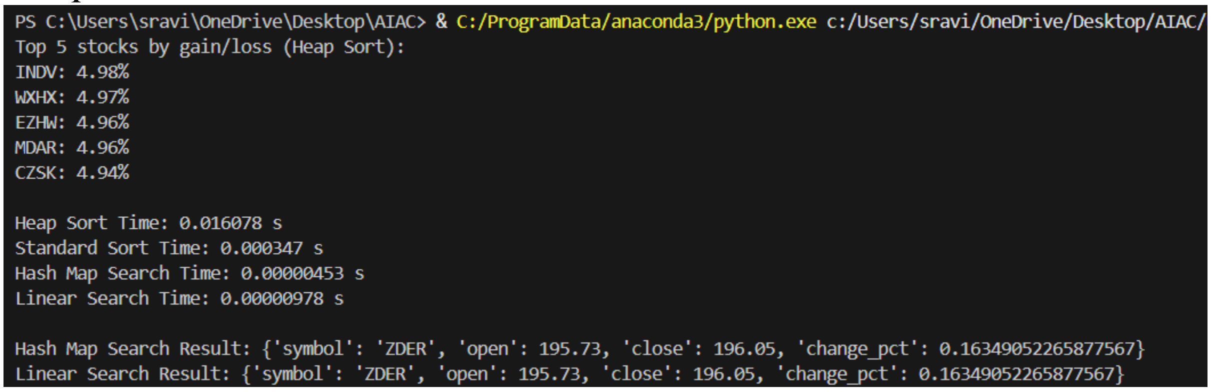


```python
import random
import time
import heapq
def generate_stocks(n):
    stocks = []
    for _ in range(n):
        symbol = ''.join(random.choices('ABCDEFGHIJKLMNOPQRSTUVWXYZ', k=4))
        open_price = round(random.uniform(100, 1000), 2)
        close_price = round(open_price * random.uniform(0.95, 1.05), 2)
        stocks.append({
            "symbol": symbol,
            "open": open_price,
            "close": close_price,
            "change_pct": ((close_price - open_price) / open_price) * 100
        })
    return stocks
def heap_sort_stocks(stocks):
    heap = [(-s["change_pct"], s) for s in stocks]
    heapq.heapify(heap)
    sorted_stocks = [heapq.heappop(heap)[1] for _ in range(len(heap))]
    return sorted_stocks
def build_stock_map(stocks):
    return {s["symbol"]: s for s in stocks}
def search_stock(stock_map, symbol):
    return stock_map.get(symbol, None)
def std_sort(stocks):
    return sorted(stocks, key=lambda s: s["change_pct"], reverse=True)
stocks = generate_stocks(1000)
stock_map = build_stock_map(stocks)
start = time.time()
sorted_heap = heap_sort_stocks(stocks)
heap_time = time.time() - start
start = time.time()
sorted_std = std_sort(stocks)
std_time = time.time() - start
symbol = stocks[0]["symbol"]
start = time.time()
```



```python
start = time.time()
result_hash = search_stock(stock_map, symbol)
hash_time = time.time() - start
start = time.time()
result_linear = next((s for s in stocks if s["symbol"] == symbol), None)
linear_time = time.time() - start
print(f"Top 5 stocks by gain/loss (Heap Sort):")
for s in sorted_heap[:5]:
    print(f"{s['symbol']}: {s['change_pct']:.2f}%")
print(f"\nHeap Sort Time: {heap_time:.6f} s")
print(f"Standard Sort Time: {std_time:.6f} s")
print(f"Hash Map Search Time: {hash_time:.8f} s")
print(f"Linear Search Time: {linear_time:.8f} s")
print(f"\nHash Map Search Result: {result_hash}")
print(f"Linear Search Result: {result_linear}")
```

Output:



```
PS C:\Users\sravi\OneDrive\Desktop\AIAC> & C:/ProgramData/anaconda3/python.exe c:/Users/sravi/OneDrive/Desktop/AIAC/
Top 5 stocks by gain/loss (Heap Sort):
INDV: 4.98%
WXHX: 4.97%
EZHW: 4.96%
MDAR: 4.96%
CZSK: 4.94%

Heap Sort Time: 0.016078 s
Standard Sort Time: 0.000347 s
Hash Map Search Time: 0.00000453 s
Linear Search Time: 0.00000978 s

Hash Map Search Result: {'symbol': 'ZDER', 'open': 195.73, 'close': 196.05, 'change_pct': 0.16349052265877567}
Linear Search Result: {'symbol': 'ZDER', 'open': 195.73, 'close': 196.05, 'change_pct': 0.16349052265877567}
```

Explanation:

- Stock data is simulated with random symbols and prices.
- Heap Sort efficiently ranks stocks by percentage change.
- Hash Map (dictionary) enables instant stock symbol lookup.
- Standard sort and dict lookup are compared for performance.
- Heap Sort and standard sort are both fast for moderate datasets; dict lookup is much faster than linear search for symbol retrieval.
- Trade-off: Heap Sort is efficient for partial/top-k results, while sorted() is more general-purpose. Hash maps are optimal for search by key.