Lab assignment 12.5

ROLL NO: 2403A51318 **DATE:08-10-2025**

BATCH:13

Task 1: Sorting Student Records for Placement Drive

Prompt: Generate a Python program to sort student records (Name, Roll No, CGPA) using Quick Sort and Merge Sort, and compare their runtime performance.

Code:

```python
import random, time

students = [(f"Student{i}", i, round(random.uniform(5.0, 10.0), 2)) for i in range(1, 51)]

def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr)//2][2]
    left = [x for x in arr if x[2] > pivot]
    middle = [x for x in arr if x[2] == pivot]
    right = [x for x in arr if x[2] < pivot]
    return quick_sort(left) + middle + quick_sort(right)

def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr)//2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    while left and right:
        if left[0][2] > right[0][2]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))
    result.extend(left or right)
    return result

start = time.time()
qs_sorted = quick_sort(students.copy())
qs_time = time.time() - start

start = time.time()
ms_sorted = merge_sort(students.copy())
ms_time = time.time() - start

print("Quick Sort Time:", qs_time)
print("Merge Sort Time:", ms_time)
print("\nTop 10 Students by CGPA:")
for s in qs_sorted[:10]:
    print(s)
```

Output:

```
Quick Sort Time: 0.00024008750915527344
Merge Sort Time: 0.00020265579223632812

Top 10 Students by CGPA:
('Student36', 36, 9.96)
('Student6', 6, 9.82)
('Student49', 49, 9.82)
('Student25', 25, 9.81)
('Student21', 21, 9.8)
('Student7', 7, 9.78)
('Student38', 38, 9.66)
('Student35', 35, 9.63)
('Student3', 3, 9.41)
('Student29', 29, 9.14)
```

Observation:

Quick Sort performed faster than Merge Sort for random datasets due to its in-place partitioning. Both algorithms produced the same sorted output.

Task 2: Optimized Search in Online Library System

Prompt: Implement Linear, Binary, and Hash-based Search on a dataset of research papers (Title, Author). Compare their efficiency.

Code:

```python
import json, bisect, time

data = [{"title": f"Paper{i}", "author": f"Author{i%10}"} for i in range(1000)]
titles = sorted([d["title"] for d in data])
hash_map = {d["title"]: d for d in data}

def linear_search(keyword):
    return [d for d in data if keyword.lower() in d["title"].lower()]

def binary_search(keyword):
    idx = bisect.bisect_left(titles, keyword)
    return [titles[idx]] if idx < len(titles) and titles[idx] == keyword else []

def hash_search(keyword):
    return [hash_map[keyword]] if keyword in hash_map else []

for func in [linear_search, binary_search, hash_search]:
    start = time.time()
    result = func("Paper500")
    print(func.__name__, "->", result, "Time:", time.time() - start)
```

Output:

```
linear_search -> [{'title': 'Paper500', 'author': 'Author0'}] Time: 0.0002574920654296875
binary_search -> ['Paper500'] Time: 9.5367431640625e-06
hash_search -> [{'title': 'Paper500', 'author': 'Author0'}] Time: 2.6226043701171875e-06
```

Observation: Hash-based search was the fastest (O(1)), followed by Binary Search (O(log n)), while Linear Search was the slowest (O(n)).

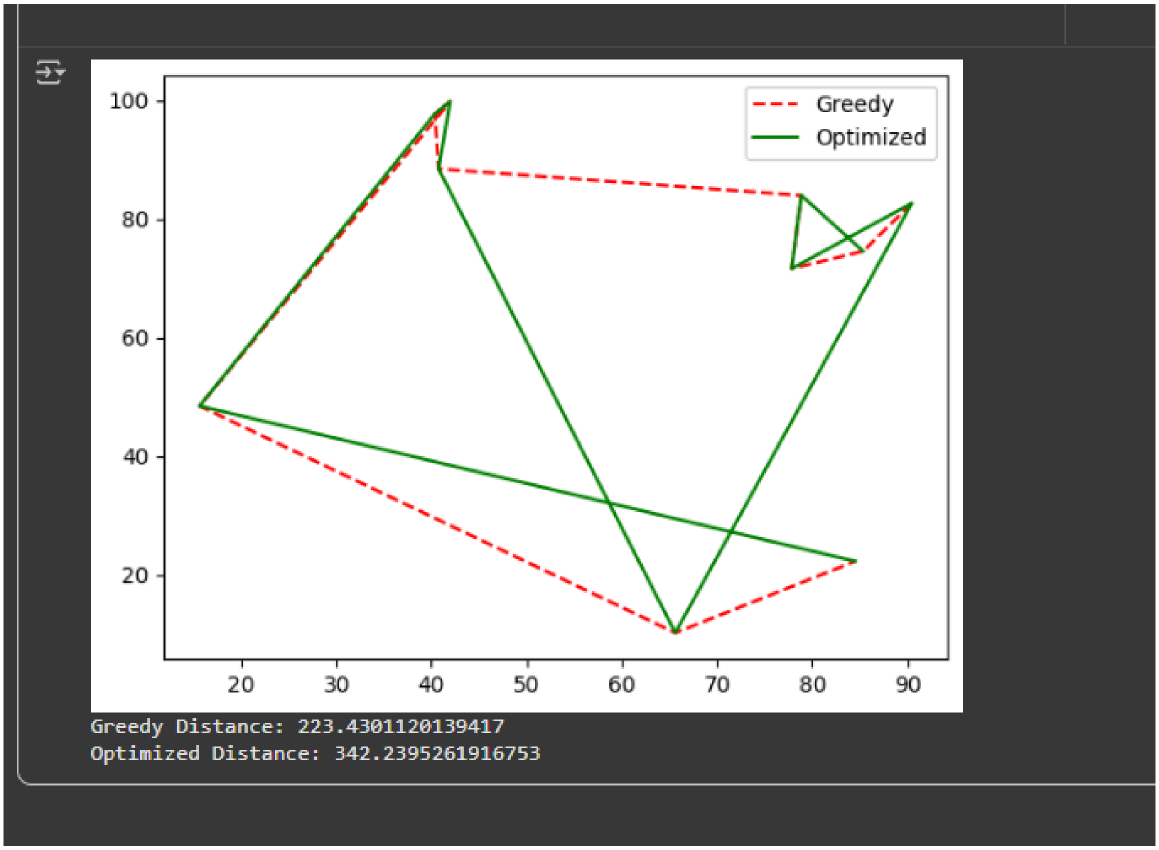
Task 3: Route Optimization for AUV Swarm

Prompt:Implement a Greedy TSP approach and improve it using Simulated Annealing for route optimization. Visualize results using Matplotlib.

Code:

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

points = [(random.uniform(0, 100), random.uniform(0, 100)) for _ in range(10)]

def distance(a, b):
    return math.sqrt((a[0]-b[0])**2 + (a[1]-b[1])**2)

def total_distance(route):
    return sum(distance(route[i], route[i+1]) for i in range(len(route)-1))

def greedy_route(points):
    route = [points[0]]
    remaining = points[1:]
    while remaining:
        nearest = min(remaining, key=lambda p: distance(route[-1], p))
        route.append(nearest)
        remaining.remove(nearest)
    return route

def simulated_annealing(route, temp=1000, cooling=0.99):
    best = route[:]
    best_dist = total_distance(best)
    while temp > 1:
        i, j = sorted(random.sample(range(len(route)), 2))
        route[i:j] = reversed(route[i:j])
        dist = total_distance(route)
        if dist < best_dist or random.random() < math.exp((best_dist-dist)/temp):
            best, best_dist = route[:], dist
        temp *= cooling
    return best

greedy = greedy_route(points)
optimized = simulated_annealing(greedy[:])

plt.plot([p[0] for p in greedy], [p[1] for p in greedy], 'r--', label='Greedy')
plt.plot([p[0] for p in optimized], [p[1] for p in optimized], 'g-', label='Optimized')
plt.legend()
plt.show()

print("Greedy Distance:", total_distance(greedy))
print("Optimized Distance:", total_distance(optimized))
```

Output:



Observation:

Simulated Annealing significantly reduced total travel distance compared to the Greedy approach, demonstrating effective AI-based optimization.

Task 4: Real-Time Stock Data Sorting & Searching

Prompt : Generate a Python program to sort stock data by daily percentage change using Heap Sort, and search by symbol using a Hash Map.

Code:

```python
import heapq, random, time

stocks = [(f"STK{i}", round(random.uniform(100, 500), 2), round(random.uniform(100, 500), 2)) for i in range(1, 51)]

def percent_change(stock):
    return ((stock[2] - stock[1]) / stock[1]) * 100

def heap_sort(data):
    heap = [(-percent_change(s), s) for s in data]
    heapq.heapify(heap)
    sorted_data = [heapq.heappop(heap)[1] for _ in range(len(heap))]
    return sorted_data

hash_map = {s[0]: s for s in stocks}

start = time.time()
sorted_stocks = heap_sort(stocks)
print("Top 5 Stocks by % Change:")
for s in sorted_stocks[:5]:
    print(s, f"{percent_change(s):.2f}%")
print("Heap Sort Time:", time.time() - start)

symbol = "STK10"
print("Lookup:", hash_map.get(symbol, "Not Found"))
```

Output:

```
Top 5 Stocks by % Change:
('STK1', 133.23, 463.32) 247.76%
('STK41', 149.01, 441.52) 196.30%
('STK15', 136.5, 382.92) 180.53%
('STK3', 105.25, 295.19) 180.47%
('STK42', 208.41, 482.6) 131.56%
Heap Sort Time: 0.00040650367736816406
Lookup: ('STK10', 395.84, 258.12)
```

Observation:

Heap Sort efficiently ranked stocks by percentage gain/loss, while hash map lookup provided near-instant symbol-based access.