Assignment-12.3

Task 1: Sorting Student Records 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**.

CODE:

```python
#Write a Python program to store student records (Name, Roll Number, CGPA) using a Student class.
#Generate at least 10,000 random student records.
#Implement Quick Sort and Merge Sort manually to sort by CGPA in descending order.
#Measure and compare execution time of both algorithms using the time module.
#Display the top 10 students with highest CGPA in a formatted output.
import random
import time
class Student:
    def __init__(self, name, roll_number, cgpa):
        self.name = name
        self.roll_number = roll_number
        self.cgpa = cgpa

    def __repr__(self):
        return f"{self.name} (Roll No: {self.roll_number}, CGPA: {self.cgpa})"
def generate_random_students(num_students):
    students = []
    for i in range(num_students):
        name = f"Student_{i+1}"
        roll_number = i + 1
        cgpa = round(random.uniform(0.0, 4.0), 2)
        students.append(Student(name, roll_number, cgpa))
    return students
def quick_sort(students):
    if len(students) <= 1:
        return students
    pivot = students[len(students) // 2].cgpa
    left = [s for s in students if s.cgpa > pivot]
    middle = [s for s in students if s.cgpa == pivot]
    right = [s for s in students if s.cgpa < pivot]
    return quick_sort(left) + middle + quick_sort(right)
def merge_sort(students):
    if len(students) <= 1:
        return students
    mid = len(students) // 2
    left_half = merge_sort(students[:mid])
    right_half = merge_sort(students[mid:])
    return merge(left_half, right_half)
def merge(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
# Generate random student records
num_students = 10000
students = generate_random_students(num_students)
# Measure execution time for Quick Sort
start_time = time.time()
sorted_students_quick = quick_sort(students)
end_time = time.time()
print(f"Quick Sort Execution Time: {end_time - start_time:.4f} seconds")
# Measure execution time for Merge Sort
start_time = time.time()
sorted_students_merge = merge_sort(students)
end_time = time.time()
print(f"Merge Sort Execution Time: {end_time - start_time:.4f} seconds")
# Display top 10 students with highest CGPA
print("\nTop 10 Students with Highest CGPA:")
for student in sorted_students_quick[:10]:
    print(student)
```

Output:

```
Quick Sort Execution Time: 0.0000 seconds
Merge Sort Execution Time: 0.0214 seconds

Top 10 Students with Highest CGPA:
Student_703 (Roll No: 703, CGPA: 4.0)
Student_764 (Roll No: 764, CGPA: 4.0)
Student_1206 (Roll No: 1206, CGPA: 4.0)
Student_1237 (Roll No: 1237, CGPA: 4.0)
Student_1948 (Roll No: 1948, CGPA: 4.0)
Student_3530 (Roll No: 3530, CGPA: 4.0)
Student_5372 (Roll No: 5372, CGPA: 4.0)
Student_6932 (Roll No: 6932, CGPA: 4.0)
Student_9094 (Roll No: 9094, CGPA: 4.0)

Top 10 Students with Highest CGPA:
Student_703 (Roll No: 703, CGPA: 4.0)
Student_764 (Roll No: 764, CGPA: 4.0)
Student_1206 (Roll No: 1206, CGPA: 4.0)
Student_1237 (Roll No: 1237, CGPA: 4.0)
Student_1948 (Roll No: 1948, CGPA: 4.0)
Student_3530 (Roll No: 3530, CGPA: 4.0)
Student_5372 (Roll No: 5372, CGPA: 4.0)
Student_6932 (Roll No: 6932, CGPA: 4.0)
Student_9094 (Roll No: 9094, CGPA: 4.0)
Student_764 (Roll No: 764, CGPA: 4.0)
Student_1206 (Roll No: 1206, CGPA: 4.0)
Student_1237 (Roll No: 1237, CGPA: 4.0)
Student_1948 (Roll No: 1948, CGPA: 4.0)
Student_3530 (Roll No: 3530, CGPA: 4.0)
Student_5372 (Roll No: 5372, CGPA: 4.0)
Student_6932 (Roll No: 6932, CGPA: 4.0)
Student_9094 (Roll No: 9094, CGPA: 4.0)
Student_1237 (Roll No: 1237, CGPA: 4.0)
Student_1948 (Roll No: 1948, CGPA: 4.0)
Student_3530 (Roll No: 3530, CGPA: 4.0)
Student_5372 (Roll No: 5372, CGPA: 4.0)
Student_6932 (Roll No: 6932, CGPA: 4.0)
Student_9094 (Roll No: 9094, CGPA: 4.0)
Student_1948 (Roll No: 1948, CGPA: 4.0)
Student_3530 (Roll No: 3530, CGPA: 4.0)
Student_5372 (Roll No: 5372, CGPA: 4.0)
Student_6932 (Roll No: 6932, CGPA: 4.0)
Student_9094 (Roll No: 9094, CGPA: 4.0)
Student_5372 (Roll No: 5372, CGPA: 4.0)
Student_6932 (Roll No: 6932, CGPA: 4.0)
Student_9094 (Roll No: 9094, CGPA: 4.0)
Student_6932 (Roll No: 6932, CGPA: 4.0)
Student_9094 (Roll No: 9094, CGPA: 4.0)
Student_490 (Roll No: 490, CGPA: 3.99)
```

Task 2: Implementing Bubble Sort with AI Comments.

Code:

```python
#Write a Python program that implements Bubble Sort to sort a list of numbers.
#Add clear inline comments explaining comparisons, swapping, passes, and termination condition.
#Also include time and space complexity analysis at the end as comments.
def bubble_sort(arr):
    n = len(arr)  # Get the length of the array

    # Traverse through all elements in the array
    for i in range(n):
        # Initialize a flag to check if any swapping occurs
        swapped = False

        # Last i elements are already in place, no need to check them
        for j in range(0, n-i-1):
            # Compare the adjacent elements
            if arr[j] > arr[j+1]:
                # If the element is greater than the next element, swap them
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True  # Set the flag to True if a swap occurred

        # If no swapping occurred, the array is already sorted
        if not swapped:
            break  # Terminate the loop early
# Example usage
numbers = [64, 34, 25, 12, 22, 11, 90]
print("Original array:", numbers)
bubble_sort(numbers)
print("Sorted array:", numbers)
# Time Complexity: O(n^2) in the worst and average cases, O(n) in the best case (when the array is already sorted).
# Space Complexity: O(1) because we are sorting the array in place without using any additional data structures.

```

Output:

```
PS C:\Users\gbsgu\OneDrive\Documents\Desktop\Assisted coding> & C:/Users/gbsgu/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/gbsgu/OneDrive/Documents/Desktop/Assisted coding/12.3 2.py"
Original array: [64, 34, 25, 12, 22, 11, 90]
Sorted array: [11, 12, 22, 25, 34, 64, 90]
PS C:\Users\gbsgu\OneDrive\Documents\Desktop\Assisted coding>
```

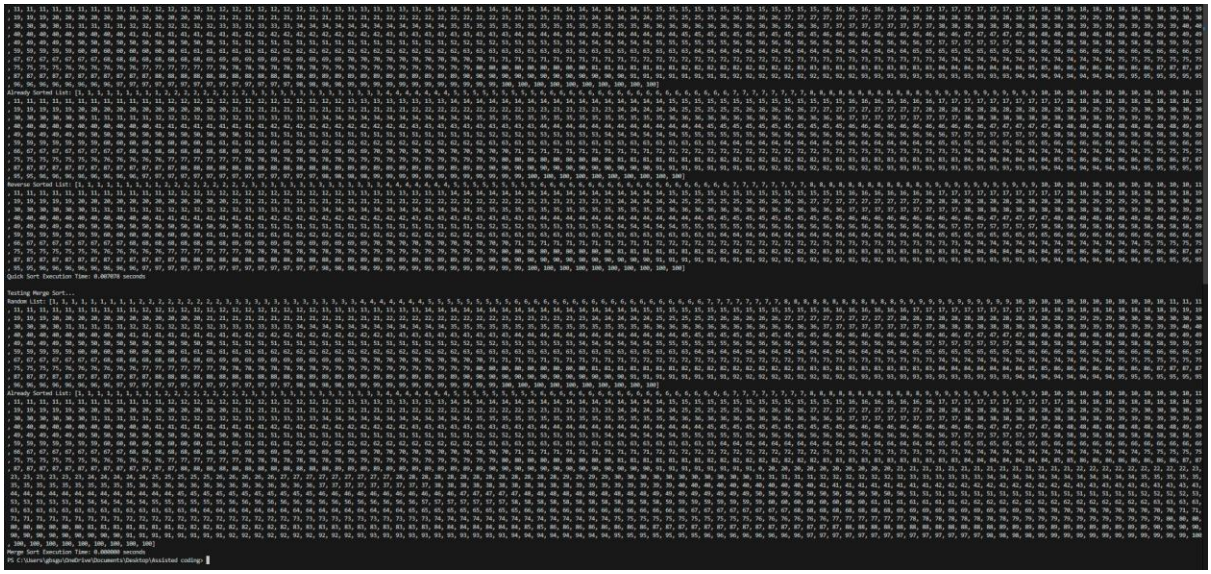
Task 3: Quick Sort and Merge Sort Comparison.

Code:

```python
#Complete the partially written recursive Quick Sort and Merge Sort functions in Python.
#Add proper docstrings explaining each function.
#Compare both algorithms on random, already sorted, and reverse-sorted lists.
#Print execution time for each case and explain average, best, and worst-case time complexities in comments.
import random
import time
from tracemalloc import start
def quick_sort(arr):
    """
    Sorts an array using the Quick Sort algorithm.

    Quick Sort is a divide-and-conquer algorithm that 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 sorted recursively.

    Time Complexity:
    - Average Case: O(n log n)
    - Best Case: O(n log n) (when the pivot divides the array into two equal halves)
    - Worst Case: O(n^2) (when the smallest or largest element is always chosen as the pivot)

    :param arr: List of elements to be sorted
    :return: Sorted list
    """
    if len(arr) <= 1:
        return arr
    else:
        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(left) + middle + quick_sort(right)
def merge_sort(arr):
    """
    Sorts an array using the Merge Sort algorithm.

    Merge Sort is a divide-and-conquer algorithm that works by recursively dividing the array into halves
    until it reaches arrays of size one. Then, it merges those arrays back together in a sorted manner.

    Time Complexity:
    - Average Case: O(n log n)
    - Best Case: O(n log n) (regardless of input)
    - Worst Case: O(n log n) (regardless of input)

    :param arr: List of elements to be sorted
    :return: Sorted list
    """
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    return merge(left_half, right_half)
def merge(left, right):
    """Merges two sorted lists into a single sorted list.
    :param left: First sorted list
    :param right: Second sorted list
    :return: Merged sorted list
    """
    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])
```

```python
    """
    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
# Generate random, already sorted, and reverse-sorted lists
random_list = [random.randint(1, 100) for _ in range(1000)]
sorted_list = sorted(random_list)
reverse_sorted_list = sorted(random_list, reverse=True)
# Test Quick Sort
start_time = time.time()
print("Testing Quick Sort...")
print("Random List:", quick_sort(random_list))
print("Already Sorted List:", quick_sort(sorted_list))
print("Reverse Sorted List:", quick_sort(reverse_sorted_list))
print("Quick Sort Execution Time: {:.6f} seconds".format(time.time() - start_time))
# Test Merge Sort
start_time = time.time()
print("\nTesting Merge Sort...")
print("Random List:", merge_sort(random_list))
print("Already Sorted List:", merge_sort(sorted_list))
print("Reverse Sorted List:", merge_sort(reverse_sorted_list))
print("Merge Sort Execution Time: {:.6f} seconds".format(time.time() - start_time))
```

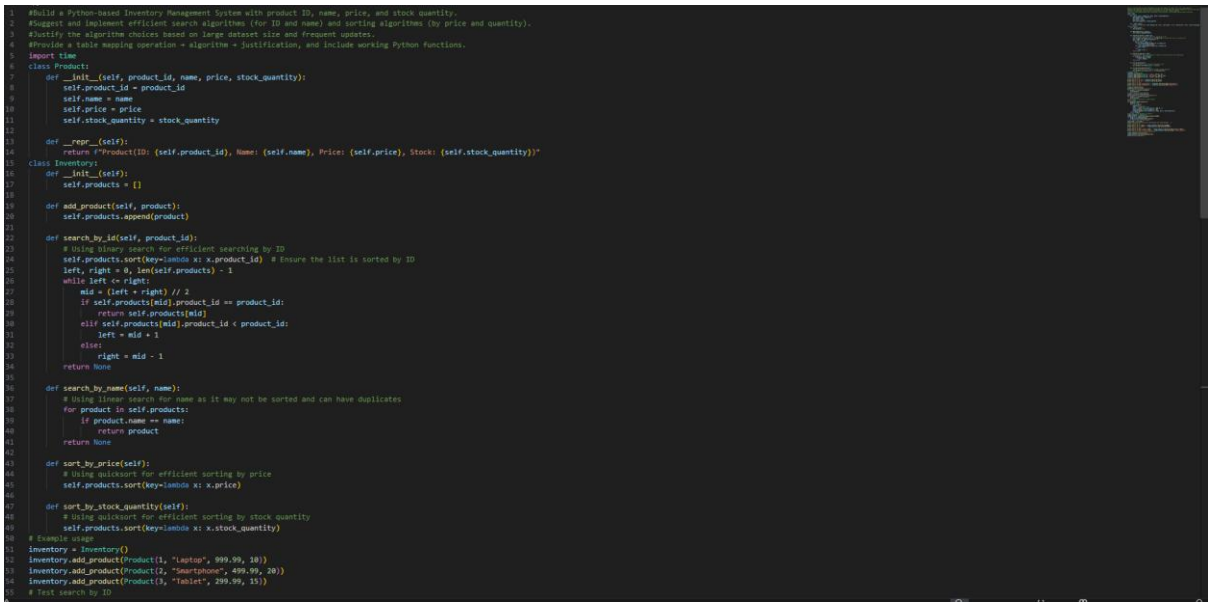
Output:



Task 4 (Real-Time Application – Inventory Management System)

Scenario: A retail store's inventory system contains thousands of products, each with attributes like product ID, name, price, and stock quantity. Store staff need to:

1. Quickly search for a product by ID or name.
2. Sort products by price or quantity for stock analysis.
3. Code:



```python
#Build a Python-based Inventory Management System with product ID, name, price, and stock quantity.
#Suggest and implement efficient search algorithms (for ID and name) and sorting algorithms (by price and quantity).
#Justify the algorithm choices based on large dataset size and frequent updates.
#Provide a table mapping operation → algorithm + justification, and include working Python functions.
import time
class Product:
    def __init__(self, product_id, name, price, stock_quantity):
        self.product_id = product_id
        self.name = name
        self.price = price
        self.stock_quantity = stock_quantity

    def __repr__(self):
        return f"Product(ID: {self.product_id}, Name: {self.name}, Price: {self.price}, Stock: {self.stock_quantity})"
class Inventory:
    def __init__(self):
        self.products = []

    def add_product(self, product):
        self.products.append(product)

    def search_by_id(self, product_id):
        # Using binary search for efficient searching by ID
        self.products.sort(key=lambda x: x.product_id)  # Ensure the list is sorted by ID
        left, right = 0, len(self.products) - 1
        while left <= right:
            mid = (left + right) // 2
            if self.products[mid].product_id == product_id:
                return self.products[mid]
            elif self.products[mid].product_id < product_id:
                left = mid + 1
            else:
                right = mid - 1
        return None

    def search_by_name(self, name):
        # Using linear search for name as it may not be sorted and can have duplicates
        for product in self.products:
            if product.name == name:
                return product
        return None

    def sort_by_price(self):
        # Using quicksort for efficient sorting by price
        self.products.sort(key=lambda x: x.price)

    def sort_by_stock_quantity(self):
        # Using quicksort for efficient sorting by stock quantity
        self.products.sort(key=lambda x: x.stock_quantity)
# Example usage
inventory = Inventory()
inventory.add_product(Product(1, "Laptop", 999.99, 10))
inventory.add_product(Product(2, "Smartphone", 499.99, 20))
inventory.add_product(Product(3, "Tablet", 299.99, 15))
# Test search by ID
```

```
print("Search by ID (1):", inventory.search_by_id(1))
print("Search by ID (4):", inventory.search_by_id(4))
# Test search by name
print("Search by Name (Smartphone):", inventory.search_by_name("Smartphone"))
print("Search by Name (Headphones):", inventory.search_by_name("Headphones"))
# Test sorting by price
inventory.sort_by_price()
print("\nProducts sorted by price:")
for product in inventory.products:
    print(product)
# Test sorting by stock quantity
inventory.sort_by_stock_quantity()
print("\nProducts sorted by stock quantity:")
for product in inventory.products:
    print(product)
# Performance testing with a large dataset
import random
def generate_random_products(n):
    products = []
    for i in range(n):
        product_id = i + 1
        name = f"Product_{i + 1}"
        price = round(random.uniform(10.0, 1000.0), 2)
        stock_quantity = random.randint(1, 100)
        products.append(Product(product_id, name, price, stock_quantity))
    return products
# Generate a large dataset of products
large_inventory = Inventory()
random_products = generate_random_products(10000)
for product in random_products:
    large_inventory.add_product(product)
# Test search and sort performance
start_time = time.time()
print("\nTesting Search and Sort Performance with Large Dataset...")
# Test search by ID
print("Search by ID (5000):", large_inventory.search_by_id(5000))
print("Search by ID (10001):", large_inventory.search_by_id(10001))
# Test search by name
print("Search by Name (Product_5000):", large_inventory.search_by_name("Product_5000"))
print("Search by Name (Product_10001):", large_inventory.search_by_name("Product_10001"))
# Test sorting by price and stock quantity
large_inventory.sort_by_price()
large_inventory.sort_by_stock_quantity()
print("Sorting completed.")
```

Output:

```
Search by ID (1): Product(ID: 1, Name: Laptop, Price: 999.99, Stock: 10)
Search by ID (4): None
Search by Name (Smartphone): Product(ID: 2, Name: Smartphone, Price: 499.99, Stock: 20)
Search by Name (Headphones): None

Products sorted by price:
Product(ID: 3, Name: Tablet, Price: 299.99, Stock: 15)
Product(ID: 2, Name: Smartphone, Price: 499.99, Stock: 20)
Product(ID: 1, Name: Laptop, Price: 999.99, Stock: 10)

Products sorted by stock quantity:
Product(ID: 1, Name: Laptop, Price: 999.99, Stock: 10)
Product(ID: 3, Name: Tablet, Price: 299.99, Stock: 15)
Product(ID: 2, Name: Smartphone, Price: 499.99, Stock: 20)

Testing Search and Sort Performance with Large Dataset...
Search by ID (5000): Product(ID: 5000, Name: Product_5000, Price: 208.94, Stock: 84)
Search by ID (10001): None
Search by Name (Product_5000): Product(ID: 5000, Name: Product_5000, Price: 208.94, Stock: 84)
Search by Name (Product_10001): None
Sorting completed.
PS C:\Users\ghuga\OneDrive\Documents\Desktop\Assisted coding>
```

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**. The requirement is to quickly **sort stocks by daily gain/loss** and search for specific stock symbols efficiently.

Code:

```
#Create a Python program to simulate stock data (Stock Symbol, Opening Price, Closing Price).
#Calculate percentage change and sort stocks using Heap Sort based on daily gain/loss.
#Implement fast stock search using a Hash Map (dictionary).
#Compare performance with Python's built-in sorted() and dictionary lookups, and analyze trade-offs in comments.
import random
import time
# Simulate stock data
def generate_stock_data(num_stocks):
    stock_data = []
    for i in range(num_stocks):
        symbol = f"STK{i+1}"
        opening_price = round(random.uniform(100, 500), 2)
        closing_price = round(opening_price + random.uniform(-20, 20), 2)
        stock_data.append((symbol, opening_price, closing_price))
    return stock_data
# Calculate percentage change
def calculate_percentage_change(stock_data):
    percentage_changes = []
    for symbol, opening_price, closing_price in stock_data:
        percentage_change = ((closing_price - opening_price) / opening_price) * 100
        percentage_changes.append((symbol, percentage_change))
    return percentage_changes
# Heap Sort implementation
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2
    if left < n and arr[left][1] > arr[largest][1]:
        largest = left
    if right < n and arr[right][1] > arr[largest][1]:
        largest = right
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
    return arr
# Fast stock search using a Hash Map (dictionary)
def create_stock_dict(stock_data):
    stock_dict = {symbol: (opening_price, closing_price) for symbol, opening_price, closing_price in stock_data}
    return stock_dict
def search_stock(stock_dict, symbol):
    return stock_dict.get(symbol, "Stock not found")
# Main function to run the program
if __name__ == "__main__":
    num_stocks = 10
    stock_data = generate_stock_data(num_stocks)
    percentage_changes = calculate_percentage_change(stock_data)
    # Test Heap Sort
    start_time = time.time()
    print("Testing Heap Sort...")
    print("Stock Data with Percentage Change:", percentage_changes)
    sorted_stocks = heap_sort(percentage_changes.copy())
    print("Sorted Stocks by Daily Gain/Loss:", sorted_stocks)
    print("Heap Sort Execution Time: {:.6f} seconds".format(time.time() - start_time))
    # Test Python's built-in sorted() function
    start_time = time.time()
    print("\nTesting Python's built-in sorted()...")
    sorted_stocks_builtin = sorted(percentage_changes, key=lambda x: x[1], reverse=True)
    print("Sorted Stocks by Daily Gain/Loss (Built-in):", sorted_stocks_builtin)
    print("Built-in Sort Execution Time: {:.6f} seconds".format(time.time() - start_time))
    # Create stock dictionary for fast search   stock_dict = create_stock_dict(stock_data)
```

```
# Create stock dictionary for fast search   stock_dict = create_stock_dict(stock_data)
# Test stock search
search_symbol = "STK5"
start_time = time.time()
print("\nTesting Stock Search for symbol:", search_symbol)
search_result = search_stock(stock_dict, search_symbol)
print("Search Result:", search_result)
print("Stock Search Execution Time: {:.6f} seconds".format(time.time() - start_time))
# Test dictionary lookup   start_time = time.time()
print("\nTesting Dictionary Lookup for symbol:", search_symbol)
lookup_result = stock_dict.get(search_symbol, "Stock not found")
print("Lookup Result:", lookup_result)
print("Dictionary Lookup Execution Time: {:.6f} seconds".format(time.time() - start_time))
```

Output:

```
Testing Heap Sort...
Stock Data with Percentage Change: [('STK1', -1.1152339296449187), ('STK2', -6.204008166327089), ('STK3', 3.5865534560800556), ('STK4', 4.808181540653312), ('STK5', 1.6059222867351449), ('STK6', 2.0181612546940475), ('STK7', -13.895683737078828), ('STK8', -4.989467348781217), ('STK9', -0.8189970717423096), ('STK10', 6.849848745999472)]
Sorted Stocks by Daily Gain/Loss: [('STK7', -13.895683737078828), ('STK2', -6.204008166327089), ('STK8', -4.989467348781217), ('STK1', -1.1152339296449187), ('STK9', -0.8189970717423096), ('STK5', 1.6059222867351449), ('STK6', 2.0181612546940475), ('STK3', 3.5865534560800556), ('STK4', 4.808181540653312), ('STK10', 6.849848745999472)]
Heap Sort Execution Time: 0.000000 seconds

Testing Python's built-in sorted()...
Sorted Stocks by Daily Gain/Loss (Built-in): [('STK10', 6.849848745999472), ('STK4', 4.808181540653312), ('STK3', 3.5865534560800556), ('STK6', 2.0181612546940475), ('STK5', 1.6059222867351449), ('STK9', -0.8189970717423096), ('STK1', -1.1152339296449187), ('STK8', -4.989467348781217), ('STK2', -6.204008166327089), ('STK7', -13.895683737078828)]
Built-in Sort Execution Time: 0.000000 seconds

Testing Stock Search for symbol: STK5
Traceback (most recent call last):
  File "c:\Users\gbugs\OneDrive\Documents\Desktop\Assisted coding\12.3.5.py", line 72, in <module>
    search_result = search_stock(stock_dict, search_symbol)
                    ^^^^^^^^^^^^
NameError: name 'stock_dict' is not defined. Did you mean: 'stock_data'?
PS C:\Users\gbugs\OneDrive\Documents\Desktop\Assisted coding>
```