

Lab 12.3

2303A51494

B27

Task 1: Sorting Student Records for Placement Drive Scenario

PROMPT:

generate a python program thaat stores student records name,roll no ,gpa to implement quick sort and merge sort algorithms

CODE:

```
❶ 12-3.py > ...
1  #generate a python program thaat stores student records name,roll no ,gpa to implement quick sort and merge sort algorithms
2  class Student:
3      def __init__(self, name, roll_no, gpa):
4          self.name = name
5          self.roll_no = roll_no
6          self.gpa = gpa
7  def quick_sort(students):
8      if len(students) <= 1:
9          return students
10     pivot = students[len(students) // 2].gpa
11     left = [s for s in students if s.gpa < pivot]
12     middle = [s for s in students if s.gpa == pivot]
13     right = [s for s in students if s.gpa > pivot]
14     return quick_sort(left) + middle + quick_sort(right)
15  def merge_sort(students):
16      if len(students) <= 1:
17          return students
18      mid = len(students) // 2
19      left_half = merge_sort(students[:mid])
20      right_half = merge_sort(students[mid:])
21      return merge(left_half, right_half)
22  def merge(left, right):
23      result = []
24      i = j = 0
25      while i < len(left) and j < len(right):
26          if left[i].gpa < right[j].gpa:
27              result.append(left[i])
28              i += 1
29          else:
30              result.append(right[j])
31              j += 1
32      result.extend(left[i:])
33      result.extend(right[j:])
34      return result
35  # Example usage:
36  students = [
```

```

        result.extend(left[i:])
        result.extend(right[j:])
        return result
    # Example usage:
students = [
    Student("Alice", 1, 3.5),
    Student("Bob", 2, 3.8),
    Student("Charlie", 3, 3.2),
    Student("David", 4, 3.9)
]
sorted_students_quick = quick_sort(students)

print("Students sorted by GPA using Quick Sort:")
for student in sorted_students_quick:
    print(f"{student.name} (Roll No: {student.roll_no}, GPA: {student.gpa})")
sorted_students_merge = merge_sort(students)
print("\nStudents sorted by GPA using Merge Sort:")
for student in sorted_students_merge:
    print(f"{student.name} (Roll No: {student.roll_no}, GPA: {student.gpa})")

```

OUTPUT:

[3.py](#)

Students sorted by GPA using Quick Sort:
 Charlie (Roll No: 3, GPA: 3.2)
 Alice (Roll No: 1, GPA: 3.5)
 Bob (Roll No: 2, GPA: 3.8)
 David (Roll No: 4, GPA: 3.9)

Students sorted by GPA using Merge Sort:
 Charlie (Roll No: 3, GPA: 3.2)
 Alice (Roll No: 1, GPA: 3.5)
 Bob (Roll No: 2, GPA: 3.8)
 David (Roll No: 4, GPA: 3.9)

› PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>

EXPLANATION:

- Quick Sort is generally faster than Merge Sort for smaller datasets due to lower constant factors and better cache performance
- merge sort is more efficient for larger datasets and is stable,
- while quick sort is not stable and can degrade to $O(n^2)$ in the worst case if the pivot selection is poor.
- Merge Sort has a consistent $O(n \log n)$ time complexity regardless of input,
- while Quick Sort's average time complexity is $O(n \log n)$ but can be $O(n^2)$ in the worst case.

Task 2: Implementing Bubble Sort with AI Comments

PROMPT:

generate a python programme to implement bubble sort and explain with comments

CODE:

```
#generate a python programme to implement bubble sort and explain with comments
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all elements in the array
    for i in range(n):
        # Last i elements are already in place, no need to check them
        for j in range(0, n-i-1):
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return arr
# Example usage:
arr = [64, 34, 25, 12, 22, 11, 90]
sorted_arr = bubble_sort(arr)
print("Sorted array is:", sorted_arr)
# Explanation:
```

OUTPUT:

```
3.py
● Sorted array is: [11, 12, 22, 25, 34, 64, 90]
```

EXPLANATION:

- The bubble_sort function takes an array as input and determines its length.
- It uses two nested loops: the outer loop iterates through each element, while the inner loop compares adjacent elements and swaps them if they are in the wrong order.
- After each pass through the array, the largest unsorted element "bubbles up" to its correct position at the end of the array.
- The process is repeated until the entire array is sorted, with the inner loop reducing its range after each pass since the last elements are already sorted.

Task 3: Quick Sort and Merge Sort Comparison

PROMPT: generate a program to implement Quick Sort and Merge Sort using recursion

CODE:

```

#generate a program to implement Quick Sort and Merge Sort using recursion
def quick_sort(arr):
    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(left) + middle + quick_sort(right)
def merge_sort(arr):
    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):
    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
# Example usage:
arr = [38, 27, 43, 3, 9, 82, 10]
sorted_arr_quick = quick_sort(arr)
print("Sorted array using Quick Sort:", sorted_arr_quick)
sorted_arr_merge = merge_sort(arr)
print("Sorted array using Merge Sort:", sorted arr merge)

```

OUTPUT:

`3.py"`
 Sorted array using Quick Sort: [3, 9, 10, 27, 38, 43, 82]
 Sorted array using Merge Sort: [3, 9, 10, 27, 38, 43, 82]

EXPLANATION:

- The `quick_sort` function implements the Quick Sort algorithm using recursion.
- It selects a pivot element and partitions the array into three sub-arrays: elements less than the pivot, elements equal to the pivot, and elements greater than the pivot.
- It then recursively sorts the left and right sub-arrays and combines them with the middle array to produce the sorted result.
- The `merge_sort` function implements the Merge Sort algorithm using recursion.

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

PROMPT:

suggest the most efficient search and sort algorithms for 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:

CODE:

```
#suggest the most efficient search and sort algorithms for a
#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.
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 binary_search(products, target_id):
    left, right = 0, len(products) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if products[mid].product_id == target_id:
            return products[mid]
        elif products[mid].product_id < target_id:
            left = mid + 1
        else:
            right = mid - 1
    return None
def quick_sort(products, key):
    if len(products) <= 1:
        return products
    pivot = products[len(products) // 2]
    left = [p for p in products if getattr(p, key) < getattr(pivot, key)]
    middle = [p for p in products if getattr(p, key) == getattr(pivot, key)]
    right = [p for p in products if getattr(p, key) > getattr(pivot, key)]
    return quick_sort(left, key) + middle + quick_sort(right, key)
# Example usage:
products = [
    Product(101, "Laptop", 999.99, 10),
    Product(102, "Smartphone", 499.99, 50),
    Product(103, "Headphones", 199.99, 100),
    Product(104, "Monitor", 299.99, 20)
]
# Sort products by price
sorted_by_price = quick_sort(products, key='price')
print("Products sorted by price:")
for product in sorted_by_price:
    print(f"{product.name} (ID: {product.product_id}, Price: {product.price})")
# Sort products by stock quantity
sorted_by_stock = quick_sort(products, key='stock_quantity')
print("\nProducts sorted by stock quantity:")
for product in sorted_by_stock:
    print(f"{product.name} (ID: {product.product_id}, Stock Quantity: {product.stock_quantity})")
# Search for a product by ID
target_product = binary_search(sorted_by_price, 102)
if target_product:
    print(f"\nProduct found: {target_product.name} (ID: {target_product.product_id}, Price: {target_product.price})")
else:
    print("\nProduct not found.")
```

OUTPUT:

`3.py"`

Products sorted by price:
Headphones (ID: 103, Price: 199.99)
Monitor (ID: 104, Price: 299.99)
Smartphone (ID: 102, Price: 499.99)
Laptop (ID: 101, Price: 999.99)

Products sorted by stock quantity:
Laptop (ID: 101, Stock Quantity: 10)
Monitor (ID: 104, Stock Quantity: 20)
Smartphone (ID: 102, Stock Quantity: 50)
Headphones (ID: 103, Stock Quantity: 100)

Product not found.

EXPLANATION:

- Binary Search is efficient ($O(\log n)$) but works only when products are sorted by product ID, so data must be pre-sorted before searching.
- Quick Sort is fast on average ($O(n \log n)$) and suitable for sorting products by price or stock quantity.
- For even faster searching, a Hash Map (dictionary) can be used to get products by ID or name in $O(1)$ time.
- Overall, combining Quick Sort for sorting and Hash Map for searching gives the best performance for large inventory systems.

TASK 5: Real Time Stock Data Sorting and Searching

PROMPT:

generate a python programme to fetch or simulate stock price data like stock symbol,opening price and closing price for a tool for analyzing stock price movements the requirement is to quickly sort stocks by daily gain/loss and search for a specific stock symbol efficiently 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.

CODE:

```
#generate a python programme to fetch or simulate stock price data like stock symbol,opening price and closing price
#for a tool for analyzing stock price movements the requirement is to quickly sort stocks by daily gain/loss
#and search for a specific stock symbol efficiently
#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.
import random

# 1. Simulate Stock Data

def generate_stock_data(n):
    stocks = []
    symbols = ["AAPL", "GOOG", "MSFT", "AMZN", "TSLA", "NFLX", "META", "NVDA", "BABA", "ORCL"]

    for i in range(n):
        symbol = random.choice(symbols) + str(i) # unique symbol
        open_price = round(random.uniform(100, 1000), 2)
        close_price = round(open_price * random.uniform(0.9, 1.1), 2)

        change_percent = round(((close_price - open_price) / open_price) * 100, 2)

        stocks.append({
            "symbol": symbol,
            "open": open_price,
            "close": close_price,
            "change": change_percent
        })

    return stocks

# 2. Heap Sort Implementation

def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[left]["change"] > arr[largest]["change"]:
        largest = left

    if right < n and arr[right]["change"] > arr[largest]["change"]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)

def heap_sort(stocks):
    n = len(stocks)

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(stocks, n, i)

    # Extract elements
    for i in range(n - 1, 0, -1):
        stocks[i], stocks[0] = stocks[0], stocks[i]
        heapify(stocks, i, 0)
```

```

1     # Reverse for descending order (highest gain first)
2     stocks.reverse()
3     return stocks
4
5 def create_stock_map(stocks):
6     stock_map = {}
7     for stock in stocks:
8         stock_map[stock["symbol"]] = stock
9     return stock_map
10
11
12 def search_stock(stock_map, symbol):
13     return stock_map.get(symbol, "Stock not found")
14 if __name__ == "__main__":
15     # Generate data
16     stocks = generate_stock_data(10)
17
18     print("\n Original Stock Data:")
19     for s in stocks:
20         print(s)
21
22     # Sort stocks
23     sorted_stocks = heap_sort(stocks)
24
25     print("\n Stocks Sorted by % Change (Highest to Lowest):")
26     for s in sorted_stocks:
27         print(s)
28
29     |   Python\w/
30
31
32     # Create hash map
33     stock_map = create_stock_map(sorted_stocks)
34
35     # Search example
36     symbol = input("\n Enter stock symbol to search: ")
37     result = search_stock(stock_map, symbol)
38
39     print("\nSearch Result:")
40     print(result)

```

OUTPUT:

3.py"

Products sorted by price:

Headphones (ID: 103, Price: 199.99)
Monitor (ID: 104, Price: 299.99)
Smartphone (ID: 102, Price: 499.99)
Laptop (ID: 101, Price: 999.99)

Products sorted by stock quantity:

Laptop (ID: 101, Stock Quantity: 10)
Monitor (ID: 104, Stock Quantity: 20)
Smartphone (ID: 102, Stock Quantity: 50)
Headphones (ID: 103, Stock Quantity: 100)

Product not found.

Original Stock Data:

```
{"symbol": "META0", "open": 314.91, "close": 325.61, "change": 3.4}  
{"symbol": "GOOG1", "open": 698.67, "close": 641.11, "change": -8.24}  
{"symbol": "MSFT2", "open": 816.16, "close": 742.11, "change": -9.07}  
{"symbol": "TSLA3", "open": 399.35, "close": 381.11, "change": -4.57}  
{"symbol": "BABA4", "open": 389.59, "close": 414.06, "change": 6.28}  
{"symbol": "ORCL5", "open": 961.23, "close": 1020.12, "change": 6.13}  
{"symbol": "NFLX6", "open": 954.7, "close": 938.85, "change": -1.66}  
{"symbol": "BABA7", "open": 423.26, "close": 427.27, "change": 0.95}  
{"symbol": "MSFT8", "open": 673.24, "close": 722.55, "change": 7.32}  
{"symbol": "MSFT9", "open": 928.49, "close": 890.05, "change": -4.14}
```

Original Stock Data:

```
{"symbol": "META0", "open": 314.91, "close": 325.61, "change": 3.4}  
{"symbol": "GOOG1", "open": 698.67, "close": 641.11, "change": -8.24}  
{"symbol": "MSFT2", "open": 816.16, "close": 742.11, "change": -9.07}  
 {"symbol": "TSLA3", "open": 399.35, "close": 381.11, "change": -4.57}  
 {"symbol": "BABA4", "open": 389.59, "close": 414.06, "change": 6.28}  
 {"symbol": "ORCL5", "open": 961.23, "close": 1020.12, "change": 6.13}  
 {"symbol": "NFLX6", "open": 954.7, "close": 938.85, "change": -1.66}  
 {"symbol": "BABA7", "open": 423.26, "close": 427.27, "change": 0.95}  
 {"symbol": "MSFT8", "open": 673.24, "close": 722.55, "change": 7.32}  
 {"symbol": "MSFT9", "open": 928.49, "close": 890.05, "change": -4.14}
```

Stocks Sorted by % Change (Highest to Lowest):

```
{"symbol": "MSFT8", "open": 673.24, "close": 722.55, "change": 7.32}  
 {"symbol": "BABA4", "open": 389.59, "close": 414.06, "change": 6.28}  
 {"symbol": "ORCL5", "open": 961.23, "close": 1020.12, "change": 6.13}  
 {"symbol": "META0", "open": 314.91, "close": 325.61, "change": 3.4}  
 {"symbol": "BABA7", "open": 423.26, "close": 427.27, "change": 0.95}  
 {"symbol": "NFLX6", "open": 954.7, "close": 938.85, "change": -1.66}  
 {"symbol": "MSFT9", "open": 928.49, "close": 890.05, "change": -4.14}  
 {"symbol": "TSLA3", "open": 399.35, "close": 381.11, "change": -4.57}  
 {"symbol": "GOOG1", "open": 698.67, "close": 641.11, "change": -8.24}  
 {"symbol": "MSFT2", "open": 816.16, "close": 742.11, "change": -9.07}
```

Enter stock symbol to search: MSFT8

Search Result:

```
{"symbol": "MSFT8", "open": 673.24, "close": 722.55, "change": 7.32}
```

EXPLANATION:

- Heap Sort has a time complexity of $O(n \log n)$, ensuring efficient and consistent sorting for large datasets.
- Hash Map search works in $O(1)$ average time, making stock lookup extremely fast.
- Compared to linear search ($O(n)$), Hash Maps drastically reduce search time.
- Combining Heap Sort and Hash Maps gives both efficient ranking and instant data retrieval.