Assignment-12.3

G.Sai teja

2303A52135

Batch 41

Task-1: Sorting Student Records for Placement Drive

PROMPT:

Write a Python script placements_sort.py that defines a Student dataclass with fields name, roll, and cgpa, generates datasets using familiar names like Raju, Rahul, Rani, Priya, etc., implements manual Quick Sort and Merge Sort, sorts students by CGPA descending with name and roll as tie-breakers, measures average runtimes for configurable dataset sizes and runs, validates against Python's sorted(), and prints clear sections showing correctly sorted student records, performance comparison with speedup, and the top 10 students, all via an argparse CLI with docstrings and type hints.

CODE:

```
#Write a Python script placements_sort.py that defines a Student dataclass with fields name, roll, a
from __future__ import annotations
import argparse
import random
import time
from dataclasses import dataclass
from typing import List, Callable
@dataclass
class Student:
    """Dataclass representing a student."""
    name: str
    roll: int
    cgpa: float
def generate_students(size: int) -> List[Student]:
    """Generate a random dataset of students with familiar names."""
    sample_names = [
        "Raju", "Rahul", "Rani", "Priya", "Sita", "Gita", "Mohan",
        "Arjun", "Anjali", "Sunita", "Vijay", "Kiran", "Asha",
        "Deepak", "Neha", "Pooja", "Rajesh", "Meena", "Suresh", "Lakshmi"
    ]
    students = []
    for i in range(size):
        name = random.choice(sample_names)
        roll = i + 1
        cgpa = round(random.uniform(5.0, 10.0), 2)
        students.append(Student(name, roll, cgpa))
    return students
def compare_students(a: Student, b: Student) -> int:
    """Comparison function for students: CGPA desc, then name asc, then roll asc."""
    if a.cgpa != b.cgpa:
        return -1 if a.cgpa > b.cgpa else 1
    if a.name != b.name:
        return -1 if a.name < b.name else 1
```

Welcome   Lab-14.py

Lab-14.py > main

```python
    return -1 if a.roll < b.roll else 1 if a.roll > b.roll else 0
def quick_sort(arr: List[Student]) -> List[Student]:
    """Quick Sort implementation."""
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if compare_students(x, pivot) < 0]
    middle = [x for x in arr if compare_students(x, pivot) == 0]
    right = [x for x in arr if compare_students(x, pivot) > 0]
    return quick_sort(left) + middle + quick_sort(right)
def merge_sort(arr: List[Student]) -> List[Student]:
    """Merge Sort implementation."""
    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: List[Student], right: List[Student]) -> List[Student]:
    """Helper function to merge two sorted lists."""
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if compare_students(left[i], right[j]) <= 0:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
def measure_runtime(sort_fn: Callable[[List[Student]], List[Student]], dataset: List[Student], runs:
```

Welcome   Lab-14.py

Lab-14.py > main

```python
def measure_runtime(sort_fn: Callable[[List[Student]], List[Student]], dataset: List[Student], runs:
    """Measure average runtime of a sorting function."""
    total_time = 0.0
    for _ in range(runs):
        data_copy = dataset[:]
        start = time.perf_counter()
        sorted_data = sort_fn(data_copy)
        end = time.perf_counter()
        total_time += (end - start)
        # Validate correctness
        assert sorted_data == sorted(dataset, key=lambda s: (-s.cgpa, s.name, s.roll))
    return total_time / runs
def main() -> None:
    parser = argparse.ArgumentParser(description="Compare Quick Sort and Merge Sort for student place
    parser.add_argument("--size", type=int, default=1000, help="Number of students in dataset")
    parser.add_argument("--runs", type=int, default=5, help="Number of runs for averaging runtime")
    args = parser.parse_args()
    dataset = generate_students(args.size)
    quick_time = measure_runtime(quick_sort, dataset, args.runs)
    merge_time = measure_runtime(merge_sort, dataset, args.runs)
    speedup = merge_time / quick_time if quick_time > 0 else float("inf")
    print("\n=============================")
    print(" Correctly Sorted Student Records ")
    print("=============================")
    sorted_students = quick_sort(dataset)
    for student in sorted_students[:20]:  # show first 20 for verification
        print(f"{student.name:<8} Roll:{student.roll:<4} CGPA:{student.cgpa:.2f}")
    print("\n=============================")
    print(" Performance Comparison ")
    print("=============================")
    print(f"Dataset size: {args.size}, Runs: {args.runs}")
    print(f"Quick Sort: {quick_time:.6f} seconds (avg)")
    print(f"Merge Sort: {merge_time:.6f} seconds (avg)")
```

Welcome   Lab-14.py

Lab-14.py > main

```python
def main() -> None:
    print(f"Merge Sort: {merge_time:.6f} seconds (avg)")
    print(f"Speedup (Merge/Quick): {speedup:.2f}x")
    print("\n=============================")
    print(" Top 10 Students ")
    print("=============================")
    for i, student in enumerate(sorted_students[:10], start=1):
        print(f"{i:2d}. Name: {student.name}, Roll: {student.roll}, CGPA: {student.cgpa:.2f}")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
 Correctly Sorted Student Records
==============================
Meena     Roll:223  CGPA:10.00
Rani      Roll:770  CGPA:9.99
Sunita    Roll:358  CGPA:9.98
Suresh    Roll:590  CGPA:9.98
Arjun     Roll:378  CGPA:9.97
Asha      Roll:210  CGPA:9.97
Kiran     Roll:610  CGPA:9.97
Pooja     Roll:894  CGPA:9.97
Rajesh    Roll:462  CGPA:9.97
Sunita    Roll:508  CGPA:9.97
Neha      Roll:519  CGPA:9.96
Anjali    Roll:583  CGPA:9.95
Asha      Roll:172  CGPA:9.95
Sita      Roll:888  CGPA:9.95
Arjun     Roll:665  CGPA:9.94
Meena     Roll:429  CGPA:9.93
Neha      Roll:644  CGPA:9.93
Deepak    Roll:293  CGPA:9.91
Priya     Roll:367  CGPA:9.91
Priya     Roll:706  CGPA:9.91
```

```
==============================
Dataset size: 1000, Runs: 5
Quick Sort: 0.006461 seconds (avg)
Merge Sort: 0.002273 seconds (avg)
Speedup (Merge/Quick): 0.35x

==============================
 Top 10 Students
==============================
 1. Name: Meena, Roll: 223, CGPA: 10.00
 2. Name: Rani, Roll: 770, CGPA: 9.99
 3. Name: Sunita, Roll: 358, CGPA: 9.98
 4. Name: Suresh, Roll: 590, CGPA: 9.98
 5. Name: Arjun, Roll: 378, CGPA: 9.97
 6. Name: Asha, Roll: 210, CGPA: 9.97
 7. Name: Kiran, Roll: 610, CGPA: 9.97
 8. Name: Pooja, Roll: 894, CGPA: 9.97
 9. Name: Rajesh, Roll: 462, CGPA: 9.97
10. Name: Sunita, Roll: 508, CGPA: 9.97
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

EXPLANATION: The code defines a Student dataclass and generates a dataset with familiar names like Raju, Rahul, Rani, Priya, etc. It implements manual Quick Sort and Merge Sort to order students by CGPA descending, with name and roll as tie-breakers. The program measures average runtimes for both algorithms over configurable dataset sizes and runs, validates correctness against Python's built-in sorted(), and prints three clear sections: a sample of correctly sorted records, a performance comparison with speedup, and the top 10 students. All functionality is accessible via an argparse CLI, with docstrings and type hints for clarity.

Task-2: Implementing Bubble Sort with AI Comments

PROMPT:

Write a Python script that implements Bubble Sort on a list of integers. Students should code the algorithm normally, and then ask AI to generate inline comments explaining the key logic (like swapping, passes, and termination). The AI should also provide a short time complexity analysis. The expected output is a Bubble Sort implementation with clear explanatory comments and a note on its complexity.

CODE:

```
Welcome    Lab-14.py ×
Lab-14.py > ...
#Write a Python script that implements Bubble Sort on a list of integers. Students should code the al
from typing import List
def bubble_sort(arr: List[int]) -> List[int]:
    n = len(arr)
    # Outer loop: controls the number of passes
    for i in range(n):
        # Inner loop: compares adjacent elements
        for j in range(0, n - i - 1):
            # If the current element is greater than the next, swap them
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
        # After each pass, the largest element among the unsorted part
        # "bubbles up" to its correct position at the end
    # When no swaps are needed, the list is sorted and the algorithm terminates
    return arr
if __name__ == "__main__":
    data = [64, 34, 25, 12, 22, 11, 90]
    print("Original array:", data)
    sorted_data = bubble_sort(data)
    print("Sorted array:", sorted_data)
    # Complexity Analysis:
    # Bubble Sort has worst-case and average-case time complexity O(n^2),
    # because it requires nested loops over the list.
    # Best case is O(n) when the list is already sorted (with an optimized version).
    # Space complexity is O(1), since sorting is done in-place.
```

OUTPUT:

```
Original array: [64, 34, 25, 12, 22, 11, 90]
Sorted array: [11, 12, 22, 25, 34, 64, 90]
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

EXPLANATION:

Bubble Sort works by repeatedly comparing adjacent elements in a list and swapping them if they are out of order, so that with each pass the largest unsorted element moves to its correct position at the end. The outer loop controls the number of passes, while the inner loop performs the comparisons and swaps. The process continues until no swaps are needed, meaning the list is sorted. Its worst-case and average-case time complexity is O(n^2) due to the nested loops, while the best case can be O(n) if optimized to stop early when the list is already sorted, and it uses only O(1) extra space since sorting is done in place.

Task-3: Quick Sort and Merge Sort Comparison PROMPT:

Write a Python script that implements Quick Sort and Merge Sort using recursion. Provide partially completed functions and ask AI to fill in the missing recursive logic and add docstrings. Then compare both algorithms on random, sorted, and reverse-sorted lists. The expected output should include working implementations of Quick Sort and Merge Sort, along with AI-generated explanations of their average, best, and worst-case time complexities.

CODE:

```python
#Write a Python script that implements Quick Sort and Merge Sort using recursion. Provide partially c
import random
import time
from typing import List
def quick_sort(arr: List[int]) -> List[int]:
    """
    Recursive Quick Sort implementation.
    Splits the list around a pivot and recursively sorts sublists.
    """
    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: List[int]) -> List[int]:
    """
    Recursive Merge Sort implementation.
    Divides the list into halves, recursively sorts them,
    and merges the sorted halves.
    """
    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: List[int], right: List[int]) -> List[int]:
    """Helper function to merge two sorted lists."""
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
```

```python
def merge(left: List[int], right: List[int]) -> List[int]:
        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 measure_runtime(sort_fn, arr: List[int], runs: int = 3) -> float:
    """Measure average runtime of a sorting function."""
    total = 0.0
    for _ in range(runs):
        data_copy = arr[:]
        start = time.perf_counter()
        sort_fn(data_copy)
        end = time.perf_counter()
        total += (end - start)
    return total / runs
def main():
    sizes = [1000]
    test_cases = {
        "Random": lambda n: [random.randint(0, 10000) for _ in range(n)],
        "Sorted": lambda n: list(range(n)),
        "Reverse-Sorted": lambda n: list(range(n, 0, -1)),
    }
    for size in sizes:
        print(f"\n=== Dataset Size: {size} ===")
        for case_name, generator in test_cases.items():
            dataset = generator(size)
            quick_time = measure_runtime(quick_sort, dataset)
            merge_time = measure_runtime(merge_sort, dataset)
```

```python
def main():
            print(f"\nCase: {case_name}")
            print(f"Quick Sort: {quick_time:.6f} seconds (avg)")
            print(f"Merge Sort: {merge_time:.6f} seconds (avg)")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
=== Dataset Size: 1000 ===

Case: Random
Quick Sort: 0.001073 seconds (avg)
Merge Sort: 0.001181 seconds (avg)

Case: Sorted
Quick Sort: 0.000676 seconds (avg)
Merge Sort: 0.000832 seconds (avg)

Case: Reverse-Sorted
Quick Sort: 0.000929 seconds (avg)
Merge Sort: 0.000956 seconds (avg)
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

EXPLANATION:

This script provides a clear comparison between Quick Sort and Merge Sort across different types of input data. It uses recursive implementations for both sorting algorithms, which are easier to understand and maintain. The performance measurement is done using the time module, and the results are averaged over multiple runs to ensure reliability. The use of lambda functions for generating test cases keeps the code concise and flexible. Overall, this script effectively demonstrates the differences in performance between the two sorting algorithms under various conditions.

Task-4: Real-Time Application – Inventory Management System

PROMPT:

Design a Python program for a retail store's inventory management system. The inventory contains thousands of products, each with attributes like product ID, name, price, and stock quantity. Staff need to quickly search for products by ID or name, and sort products by price or quantity for stock analysis. Use AI to suggest the most efficient search and sort algorithms for this use case, implement them in Python, and justify the choices based on dataset size, update frequency, and performance requirements. The expected output should include a table mapping operation → recommended algorithm → justification, plus working Python functions for searching and sorting the inventory.

CODE:

Welcome   Lab-14.py ×

Lab-14.py > main

```python
#Design a Python program for a retail store’s inventory management system. The inventory contains thc
from dataclasses import dataclass
from typing import List, Optional
@dataclass
class Product:
    product_id: int
    name: str
    price: float
    quantity: int
# --- Search Algorithms ---
def binary_search(products: List[Product], target_id: int) -> Optional[Product]:
    """Binary search for product by ID (requires sorted by ID)."""
    low, high = 0, len(products) - 1
    while low <= high:
        mid = (low + high) // 2
        if products[mid].product_id == target_id:
            return products[mid]
        elif products[mid].product_id < target_id:
            low = mid + 1
        else:
            high = mid - 1
    return None
def linear_search(products: List[Product], target_name: str) -> Optional[Product]:
    """Linear search for product by name (works on unsorted data)."""
    for product in products:
        if product.name.lower() == target_name.lower():
            return product
    return None
# --- Sort Algorithms ---
def merge_sort(products: List[Product], key: str) -> List[Product]:
    """Merge Sort implementation to sort products by a given key."""
    if len(products) <= 1:
        return products
```

```python
def merge_sort(products: List[Product], key: str) -> List[Product]:
    mid = len(products) // 2
    left = merge_sort(products[:mid], key)
    right = merge_sort(products[mid:], key)
    return merge(left, right, key)
def merge(left: List[Product], right: List[Product], key: str) -> List[Product]:
    """Helper function to merge two sorted lists."""
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if getattr(left[i], key) <= getattr(right[j], key):
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
# --- Demo and Justification Output ---
def print_algorithm_table():
    print("\n=== Recommended Algorithms ===")
    print(f"{'Operation':<25}{'Algorithm':<20}{'Justification'}")
    print("-" * 70)
    print(f"{'Search by ID':<25}{'Binary Search':<20}"
          f"{'Efficient O(log n) lookup on sorted IDs'}")
    print(f"{'Search by Name':<25}{'Linear Search':<20}"
          f"{'Names unsorted; O(n) acceptable for occasional lookups'}")
    print(f"{'Sort by Price/Quantity':<25}{'Merge Sort':<20}"
          f"{'Stable O(n log n) sort, good for large datasets'}")
def main():
    # Sample inventory
    inventory = [
```

```python
def main():
        Product(101, "Apple", 50.0, 120),
        Product(205, "Banana", 20.0, 200),
        Product(150, "Orange", 30.0, 80),
        Product(300, "Mango", 100.0, 50),
    ]
    # Show recommended algorithms
    print_algorithm_table()
    # Search examples
    print("\n=== Search Examples ===")
    sorted_by_id = merge_sort(inventory, "product_id")
    found = binary_search(sorted_by_id, 150)
    print("Search by ID 150:", found)
    found_name = linear_search(inventory, "Mango")
    print("Search by Name 'Mango':", found_name)
    # Sort examples
    print("\n=== Sort Examples ===")
    sorted_by_price = merge_sort(inventory, "price")
    print("Sorted by Price:", sorted_by_price)
    sorted_by_quantity = merge_sort(inventory, "quantity")
    print("Sorted by Quantity:", sorted_by_quantity)
if __name__ == "__main__":
    main()
```

OUTPUT:

```
Operation                Algorithm           Justification
----------------------------------------------------------------------
Search by ID             Binary Search       Efficient O(log n) l
ookup on sorted IDs
Search by Name           Linear Search       Names unsorted; O(n)
 acceptable for occasional lookups
Sort by Price/Quantity   Merge Sort          Stable O(n log n) so
rt, good for large datasets

=== Search Examples ===
Search by ID 150: Product(product_id=150, name='Orange', price=30
.0, quantity=80)
Search by Name 'Mango': Product(product_id=300, name='Mango', pri
ce=100.0, quantity=50)

=== Sort Examples ===
Sorted by Price: [Product(product_id=205, name='Banana', price=20
.0, quantity=200), Product(product_id=150, name='Orange', price=3
0.0, quantity=80), Product(product_id=101, name='Apple', price=50
.0, quantity=120), Product(product_id=300, name='Mango', price=10
0.0, quantity=50)]
Sorted by Quantity: [Product(product_id=300, name='Mango', price=
100.0, quantity=50), Product(product_id=150, name='Orange', price
=30.0, quantity=80), Product(product_id=101, name='Apple', price=
50.0, quantity=120), Product(product_id=205, name='Banana', price
=20.0, quantity=200)]
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

EXPLANATION:

The inventory management system code efficiently handles searching and sorting for thousands of products. For product ID lookups, Binary Search is used because IDs are numeric and can be kept sorted, giving fast O(\log n) performance. For product name lookups, Linear Search is chosen since names are not guaranteed to be sorted, and occasional O(n) scans are acceptable. For sorting by price or quantity, Merge Sort is implemented because it is stable, consistently runs in O(n\log n), and works well for large datasets. The program outputs a clear table mapping each operation to its recommended algorithm with justification, and provides working Python functions (binary_search, linear_search, merge_sort) that demonstrate these operations on sample inventory data. This design balances speed, scalability, and practicality for real-time inventory management.

Task-5: Real-Time Stock Data Sorting G Searching

PROMPT:

This program simulates stock price data and provides a simple menu-driven analysis tool. It generates random stocks with symbols, opening and closing prices, and calculates daily percentage change. Heap Sort is used to rank stocks efficiently by gain or loss, while a Hash Map (dictionary) enables instant O(1) lookups by symbol. The menu allows users to view the top five stocks, search for a specific symbol, compare Heap Sort performance against Python's built-in sorted() function, or exit. This design combines algorithmic efficiency with practical usability, making it suitable for real-time stock analysis.

CODE:

```
Welcome    Lab-14.py ×
Lab-14.py > ...
#This program simulates stock price data and provides a simple menu-driven analysis tool. It generate
import random
import time
def generate_stocks(n):
    symbols=["TCS","INFY","HDFC","RELIANCE","WIPRO","SBIN","ITC","ADANI","AXIS","ICICI"]
    stocks=[]
    for i in range(n):
        symbol=random.choice(symbols)+str(i)
        open_price=random.randint(100,5000)
        close_price=open_price+random.randint(-200,200)
        percent_change=((close_price-open_price)/open_price)*100
        stocks.append({"symbol":symbol,"open":open_price,"close":close_price,"change":round(percent_c
    return stocks
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(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[::-1]
def build_hash_map(stocks):
```

Welcome Lab-14.py

Lab-14.py > ...

```python
    return {stock["symbol"]:stock for stock in stocks}
def search_stock(stock_map,symbol):
    return stock_map.get(symbol,"Stock not found")
# Generate data
n=1000
stocks=generate_stocks(n)
stock_map=build_hash_map(stocks)
sorted_heap=heap_sort(stocks.copy())
while True:
    print("\n--- Stock Analysis Menu ---")
    print("1. Show Top 5 Stocks")
    print("2. Search Stock by Symbol")
    print("3. Performance Comparison")
    print("4. Exit")
    choice=input("Enter your choice: ")
    if choice=="1":
        print("\nTop 5 Stocks by % Gain/Loss:\n")
        for i in range(5):
            s=sorted_heap[i]
            print(f"{i+1}. {s['symbol']} | Open:{s['open']} | Close:{s['close']} | Change:{s['change'
    elif choice=="2":
        symbol=input("Enter Stock Symbol: ")
        result=search_stock(stock_map,symbol)
        print("Search Result:",result)
    elif choice=="3":
        start=time.perf_counter()
        heap_sort(stocks.copy())
        heap_time=time.perf_counter()-start
        start=time.perf_counter()
        sorted(stocks,key=lambda x:x["change"],reverse=True)
        builtin_time=time.perf_counter()-start
        print("\nPerformance Comparison:")
        print("Heap Sort Time:",round(heap_time,6),"seconds")
```

Welcome Lab-14.py

Lab-14.py > ...

```python
        print("Heap Sort Time:",round(heap_time,6),"seconds")
        print("Built-in sorted() Time:",round(builtin_time,6),"seconds")
    elif choice=="4":
        print("Exiting program...")
        break
    else:
        print("Invalid choice! Try again.")
```

OUTPUT:

```
--- Stock Analysis Menu ---
1. Show Top 5 Stocks
2. Search Stock by Symbol
3. Performance Comparison
4. Exit
Enter your choice: 1

Top 5 Stocks by % Gain/Loss:

1. RELIANCE979 | Open:131 | Close:305 | Change:132.82%
2. ITC707 | Open:129 | Close:280 | Change:117.05%
3. INFY816 | Open:135 | Close:263 | Change:94.81%
4. TCS371 | Open:148 | Close:279 | Change:88.51%
5. ITC677 | Open:194 | Close:363 | Change:87.11%

--- Stock Analysis Menu ---
1. Show Top 5 Stocks
2. Search Stock by Symbol
3. Performance Comparison
4. Exit
Enter your choice: 4
Exiting program...
PS C:\Users\saite\Downloads\AI ASSISTENT CODING>
```

EXPLANATION:

This code simulates stock data and provides a menu-driven analysis tool. It generates random stocks with opening and closing prices, calculates percentage change, and ranks them using Heap Sort for efficient O(n\log n) sorting. Searching is optimized with a Hash Map (dictionary), allowing instant O(1) lookups by symbol. The menu lets users view the top five stocks, search by symbol, compare Heap Sort against Python's built-in sorted() function, or exit. The performance comparison shows that while built-ins are faster due to internal optimizations, Heap Sort demonstrates algorithmic efficiency and control, making the program practical for real-time stock analysis.