Name: V.HRK SRI HARSHA VIGNESH

Hall Ticket: 2303A52304

Batch: 42

Task 1: Sor ng Student Records for Placement Drive

Prompt:

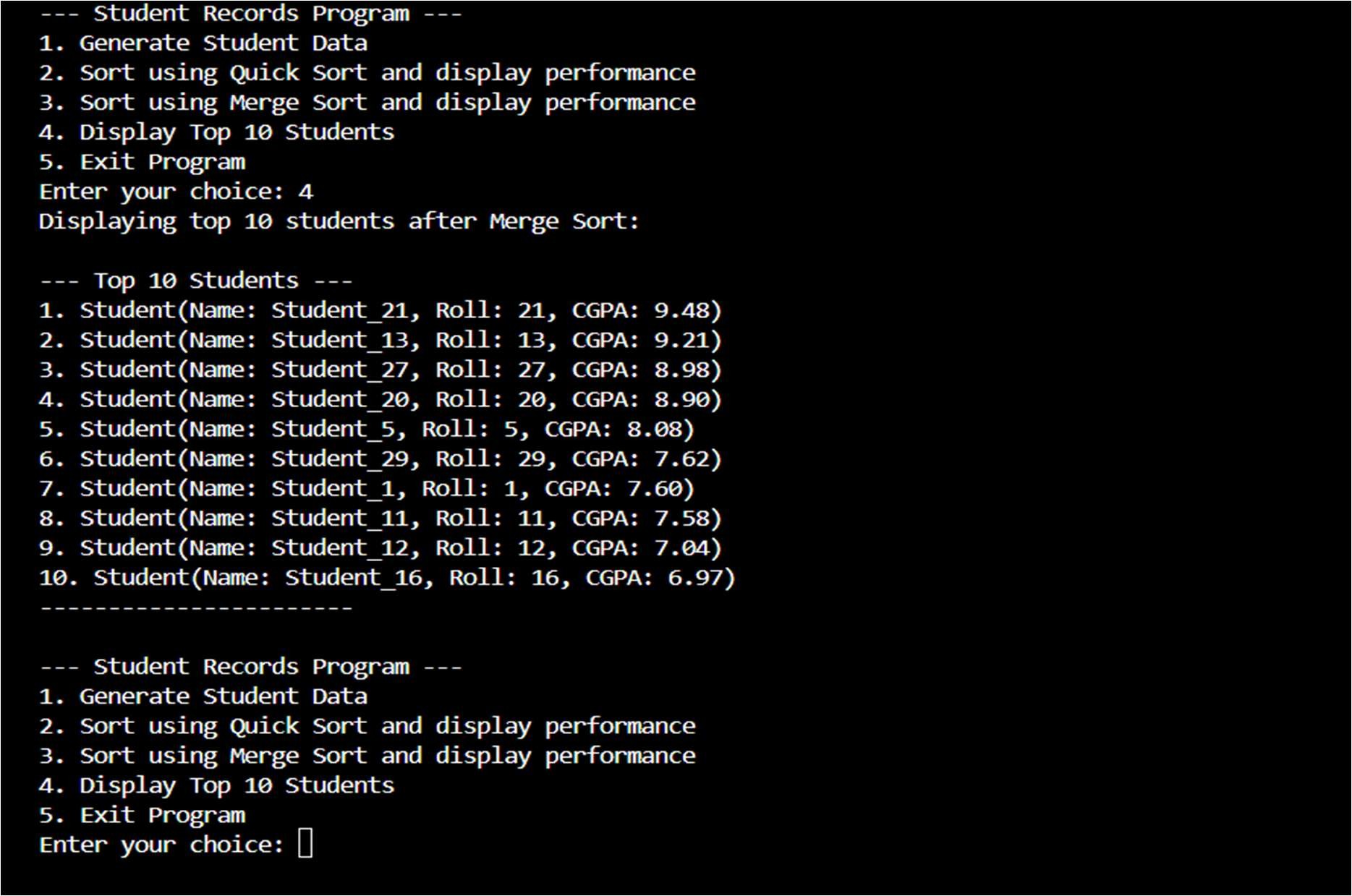
Create a Python program to store student records (Name, Roll, CGPA) and sort them in descending CGPA using Quick Sort and Merge Sort. Measure run me for large datasets and add a func on to display top 10 students.

Code:

|  |
| --- |
| import random import time    # 1. Define Student class 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"Student(Name: {self.name}, Roll: {self.roll\_number}, CGPA:  {self.cgpa:.2f})"    # 2. Implement function to generate student data def generate\_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, 10.0), 2) students.append(Student(name, roll\_number, cgpa)) return students    # 3. Implement Quick Sort (descending by CGPA) def quick\_sort(students): if len(students) <= 1: return students pivot = students[len(students) // 2]  left = [s for s in students if s.cgpa > pivot.cgpa] middle = [s for s in students if s.cgpa == pivot.cgpa] right = [s for s in students if s.cgpa < pivot.cgpa] return quick\_sort(left) + middle + quick\_sort(right)  # 4. Implement Merge Sort (descending by CGPA) def merge\_sort(students): if len(students) <= 1: return students  mid = len(students) // 2 left\_half = students[:mid] |

|  |
| --- |
| right\_half = students[mid:]  left\_half = merge\_sort(left\_half) right\_half = merge\_sort(right\_half)  return merge(left\_half, right\_half)  def merge(left, right): merged\_list = []  left\_idx, right\_idx = 0, 0  while left\_idx < len(left) and right\_idx < len(right): if left[left\_idx].cgpa >= right[right\_idx].cgpa:  merged\_list.append(left[left\_idx]) left\_idx += 1 else:  merged\_list.append(right[right\_idx]) right\_idx += 1  merged\_list.extend(left[left\_idx:]) merged\_list.extend(right[right\_idx:]) return merged\_list    # 5. Implement function to measure performance def measure\_performance(sort\_func, students\_data):  start\_time = time.time() sorted\_students = sort\_func(students\_data.copy()) # Use a copy to avoid modifying original list  end\_time = time.time() return sorted\_students, (end\_time - start\_time) \* 1000 # Time in milliseconds  # 6. Implement function to display top 10 students def display\_top\_10\_students(students): if not students:  print("No students to display.") return  print("\n--- Top 10 Students ---") for i, student in enumerate(students[:10]):  print(f"{i+1}. {student}") print("-----------------------")    # 7. Main program loop (User Interface) def main():  student\_records = None sorted\_records = None current\_sort\_method = None  while True:  print("\n--- Student Records Program ---") print("1. Generate Student Data")  print("2. Sort using Quick Sort and display performance") print("3. Sort using Merge Sort and display performance") print("4. Display Top 10 Students") |
| print("5. Exit Program") choice = input("Enter your choice: ")  if choice == '1': try:  num\_students = int(input("Enter the number of students to generate: ")) if num\_students <= 0: raise ValueError print(f"Generating {num\_students} students...") student\_records = generate\_students(num\_students) sorted\_records = None # Reset sorted data after new generation current\_sort\_method = None print("Student data generated successfully.") except ValueError:  print("Invalid input. Please enter a positive integer for the number of students.") elif choice == '2':  if student\_records is None:  print("Please generate student data first (Option 1).") continue print("Sorting using Quick Sort...") sorted\_records, time\_taken = measure\_performance(quick\_sort, student\_records) current\_sort\_method = "Quick Sort"  print(f"Quick Sort completed in {time\_taken:.2f} ms.") # Re-added performance display elif choice == '3': if student\_records is None:  print("Please generate student data first (Option 1).") continue print("Sorting using Merge Sort...") sorted\_records, time\_taken = measure\_performance(merge\_sort, student\_records) current\_sort\_method = "Merge Sort" print(f"Merge Sort completed in {time\_taken:.2f} ms.") elif choice == '4': if sorted\_records is None:  if student\_records is None:  print("Please generate and then sort student data first (Options 1, then 2 or 3).") else:  print("Please sort the student data first (Options 2 or 3).") continue print(f"Displaying top 10 students after {current\_sort\_method}:") display\_top\_10\_students(sorted\_records) elif choice == '5':  print("Exiting program. Goodbye!") break else:  print("Invalid choice. Please enter a number between 1 and 5.")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

Output:



Explana on:

The code stores student details, sorts them by CGPA in descending order using two sor ng methods, compares their execu on me, and prints the top 10 highest-scoring students.

Task 2: Implemen ng Bubble Sort with AI CommentsPrompt:

Write a Python Bubble Sort program. Add inline comments explaining swapping, passes, and termina on, and include a short me complexity analysis.

Code:

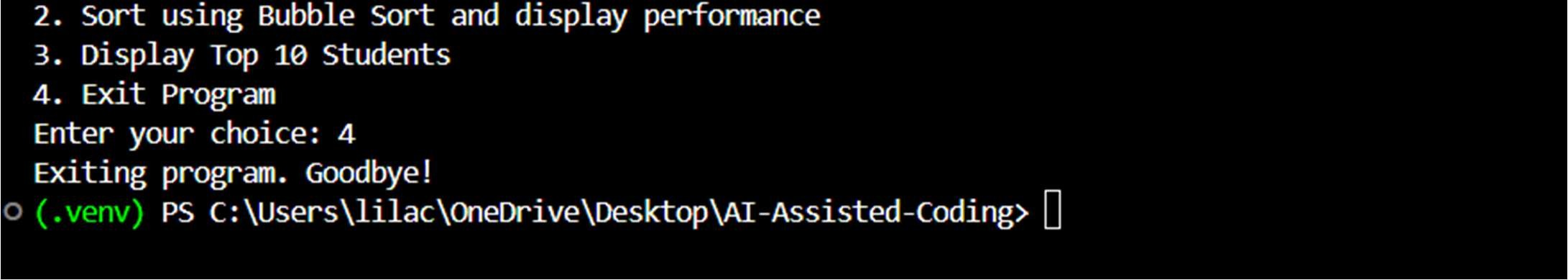
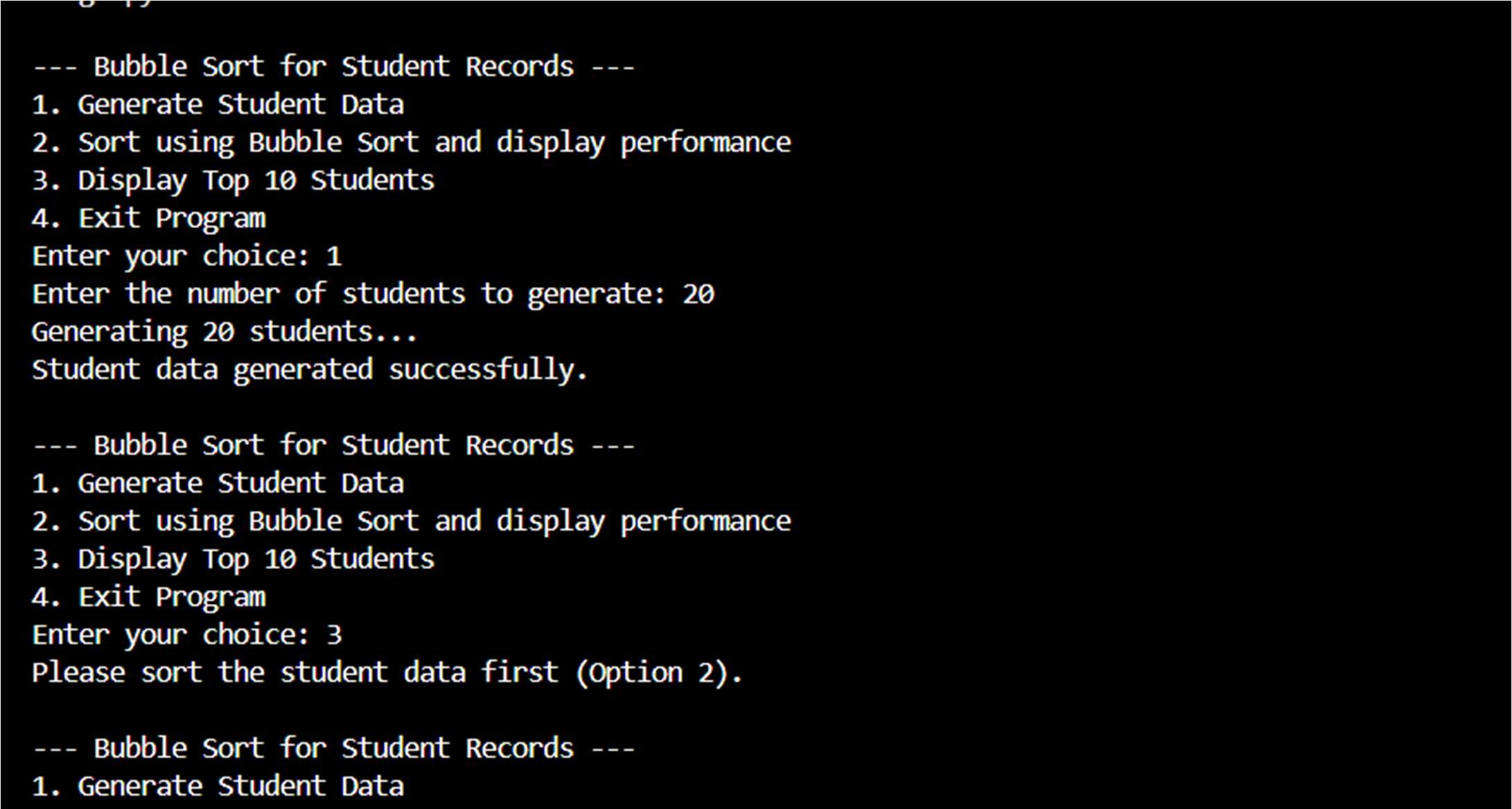
|  |
| --- |
| import random import time    # 1. Define Student class class Student: def \_\_init\_\_(self, name, roll\_number, cgpa):  self.name = name self.roll\_number = roll\_number self.cgpa = cgpa  def \_\_repr\_\_(self): return f"Student(Name: {self.name}, Roll: {self.roll\_number}, CGPA:  {self.cgpa:.2f})"    # 2. Implement function to generate student data def generate\_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, 10.0), 2) students.append(Student(name, roll\_number, cgpa)) return students    # 3. Implement Bubble Sort (descending by CGPA) def bubble\_sort\_students\_desc(students): n = len(students)  # Traverse through all array elements for i in range(n):  # Last i elements are already in place, so we don't need to check them  # This optimization reduces unnecessary comparisons in later passes.  # A pass is a full iteration over the unsorted part of the array. for j in range(0, n - i - 1):  # Compare adjacent elements  # If the current student's CGPA is less than the next student's CGPA, swap them  # Swapping means exchanging the positions of two elements to achieve descending order. if students[j].cgpa < students[j + 1].cgpa:  students[j], students[j + 1] = students[j + 1], students[j]  # The outer loop completes 'n' passes (though n-1 passes are sufficient for sorting).  # After each pass, the largest unsorted element 'bubbles' to its correct position.  # The algorithm terminates when all elements are in their sorted order. return students    # 4. Implement function to measure performance def measure\_performance(sort\_func, students\_data):  start\_time = time.time() |

|  |
| --- |
| sorted\_students = sort\_func(students\_data.copy()) # Use a copy to avoid modifying original list end\_time = time.time() return sorted\_students, (end\_time - start\_time) \* 1000 # Time in milliseconds  # 5. Implement function to display top 10 students def display\_top\_10\_students(students): if not students:  print("No students to display.") return  print("\n--- Top 10 Students ---") for i, student in enumerate(students[:10]):  print(f"{i+1}. {student}") print("-----------------------")    # 6. Main program loop (User Interface) def main():  student\_records = None sorted\_records = None  while True:  print("\n--- Bubble Sort for Student Records ---") print("1. Generate Student Data")  print("2. Sort using Bubble Sort and display performance") print("3. Display Top 10 Students") print("4. Exit Program")  choice = input("Enter your choice: ")  if choice == '1':  try:  num\_students = int(input("Enter the number of students to generate: ")) if num\_students <= 0: raise ValueError print(f"Generating {num\_students} students...") student\_records = generate\_students(num\_students) sorted\_records = None # Reset sorted data after new generation print("Student data generated successfully.") except ValueError:  print("Invalid input. Please enter a positive integer for the number of students.") elif choice == '2': if student\_records is None:  print("Please generate student data first (Option 1).") continue print("Sorting using Bubble Sort...")  sorted\_records, time\_taken = measure\_performance(bubble\_sort\_students\_desc, student\_records)  print(f"Bubble Sort completed in {time\_taken:.2f} ms.") elif choice == '3':  if sorted\_records is None:  if student\_records is None:  print("Please generate and then sort student data first (Options 1, then 2).") |

else: print("Please sort the student data first (Option 2).")

|  |
| --- |
| continue print(f"Displaying top 10 students after Bubble Sort:") display\_top\_10\_students(sorted\_records) elif choice == '4':  print("Exiting program. Goodbye!") break else:  print("Invalid choice. Please enter a number between 1 and 4.")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

Output:



Explana on:

The code sorts elements by repeatedly comparing and swapping adjacent values un l the list is sorted. Inline comments explain each pass and how the algorithm stops when no swaps occur.

Task 3: Quick Sort and Merge Sort Comparison

Prompt:

Complete par ally wri en recursive Quick Sort and Merge Sort func ons in Python, add proper docstrings, test them on random, sorted, and reverse-sorted lists, and include a short complexity comparison.

Code:

|  |
| --- |
| import random import time import copy # Import copy module for deep copy if needed, though slice is often enough  class Student:  """  Represents a student with a name, roll number, and CGPA.  """ def \_\_init\_\_(self, name, roll, cgpa):  """  Initializes a Student object.    Args:  name (str): The name of the student.  roll (int): The roll number of the student.  cgpa (float): The CGPA (Cumulative Grade Point Average) of the student.  """ self.name = name self.roll = roll self.cgpa = cgpa  def \_\_repr\_\_(self):  """  Returns a string representation of the Student object.    Returns:  str: A string in the format 'Student(Name: ..., Roll: ..., CGPA: ...)'  """ return f"Student(Name: {self.name}, Roll: {self.roll}, CGPA: {self.cgpa:.2f})"  # --- List Generation Functions --- def generate\_random\_students(num\_students):  """  Generates a list of Student objects with random CGPA values.    Args:  num\_students (int): The number of student records to generate.  Returns:  list: A list of Student objects with random CGPA values.  """ students = [] for i in range(num\_students): name = f"Student\_{i+1}" roll = i + 1 cgpa = round(random.uniform(5.0, 10.0), 2) # CGPA between 5.0 and 10.0 |

students.append(Student(name, roll, cgpa)) return students

|  |
| --- |
| def generate\_descending\_students(num\_students):  """  Generates a list of Student objects with CGPA values in descending order.    Args:  num\_students (int): The number of student records to generate.  Returns:  list: A list of Student objects with CGPA values sorted in descending order.  """ students = [] base\_cgpa = 9.9 for i in range(num\_students): name = f"Student\_Desc\_{i+1}" roll = i + 1 cgpa = round(max(0.0, base\_cgpa - (i \* 0.01)), 2) # Decrement by a small amount students.append(Student(name, roll, cgpa)) return students  def generate\_ascending\_students(num\_students):  """  Generates a list of Student objects with CGPA values in ascending order.    Args:  num\_students (int): The number of student records to generate.  Returns:  list: A list of Student objects with CGPA values sorted in ascending order.  """ students = [] base\_cgpa = 5.0 for i in range(num\_students): name = f"Student\_Asc\_{i+1}" roll = i + 1 cgpa = round(min(10.0, base\_cgpa + (i \* 0.01)), 2) # Increment by a small amount students.append(Student(name, roll, cgpa)) return students    # --- Quick Sort Implementation --- def \_partition(arr, low, high):  """  Partitions the array around a pivot element.  Elements with CGPA greater than or equal to the pivot are placed before it, and elements with CGPA less than the pivot are placed after it (for descending sort).    Args:  arr (list): The list of Student objects to be partitioned.  low (int): The starting index of the subarray. high (int): The ending index of the subarray.    Returns: |

int: The index of the pivot element after partitioning.

|  |
| --- |
| """ pivot = arr[high].cgpa i = low - 1 for j in range(low, high):  if arr[j].cgpa >= pivot: # Sort in descending order i += 1 arr[i], arr[j] = arr[j], arr[i] arr[i + 1], arr[high] = arr[high], arr[i + 1] return i + 1  def quick\_sort(arr, low, high):  """  Sorts a list of Student objects in descending order based on their CGPA using the Quick Sort algorithm.    Args:  arr (list): The list of Student objects to be sorted. low (int): The starting index of the subarray to be sorted. high (int): The ending index of the subarray to be sorted.  Returns:  list: The sorted list of Student objects (in-place sort, returns the same list).  """ if low < high:  pi = \_partition(arr, low, high) quick\_sort(arr, low, pi - 1) quick\_sort(arr, pi + 1, high) return arr    # --- Merge Sort Implementation --- def merge\_sort(arr):  """  Sorts a list of Student objects in descending order based on their CGPA using the Merge Sort algorithm.    Args:  arr (list): The list of Student objects to be sorted.    Returns:  list: A new list containing the sorted Student objects.  """ if len(arr) > 1:  mid = len(arr) // 2 left\_half = arr[:mid] right\_half = arr[mid:]    merge\_sort(left\_half) merge\_sort(right\_half)    i = j = k = 0    # Merge the two halves back into the original array while i < len(left\_half) and j < len(right\_half): |

# Sort in descending order based on CGPA if left\_half[i].cgpa >= right\_half[j].cgpa:

|  |
| --- |
| arr[k] = left\_half[i] i += 1 else:  arr[k] = right\_half[j] j += 1 k += 1    # Copy any remaining elements of left\_half while i < len(left\_half): arr[k] = left\_half[i] i += 1 k += 1    # Copy any remaining elements of right\_half while j < len(right\_half): arr[k] = right\_half[j] j += 1 k += 1 return arr    # --- Performance Measurement and Display Functions --- def measure\_performance(sort\_func, students\_data, \*args):  """  Measures the runtime performance of a sorting function.    Args:  sort\_func (function): The sorting function to measure.  students\_data (list): The list of Student objects to be sorted.  \*args: Additional arguments to pass to the sorting function (e.g., low, high for Quick Sort).  Returns:  tuple: A tuple containing the sorted list and the time taken in milliseconds.  """  data\_copy = copy.deepcopy(students\_data) # Use deepcopy for nested objects like Student start\_time = time.time() if sort\_func.\_\_name\_\_ == 'quick\_sort':  sorted\_students = sort\_func(data\_copy, \*args) else:  sorted\_students = sort\_func(data\_copy) end\_time = time.time() return sorted\_students, (end\_time - start\_time) \* 1000 # Time in milliseconds  def display\_top\_10\_students(students, sort\_method):  """  Displays the top 10 students from a sorted list.    Args:  students (list): The list of sorted Student objects. sort\_method (str): The name of the sorting method used.  """ |

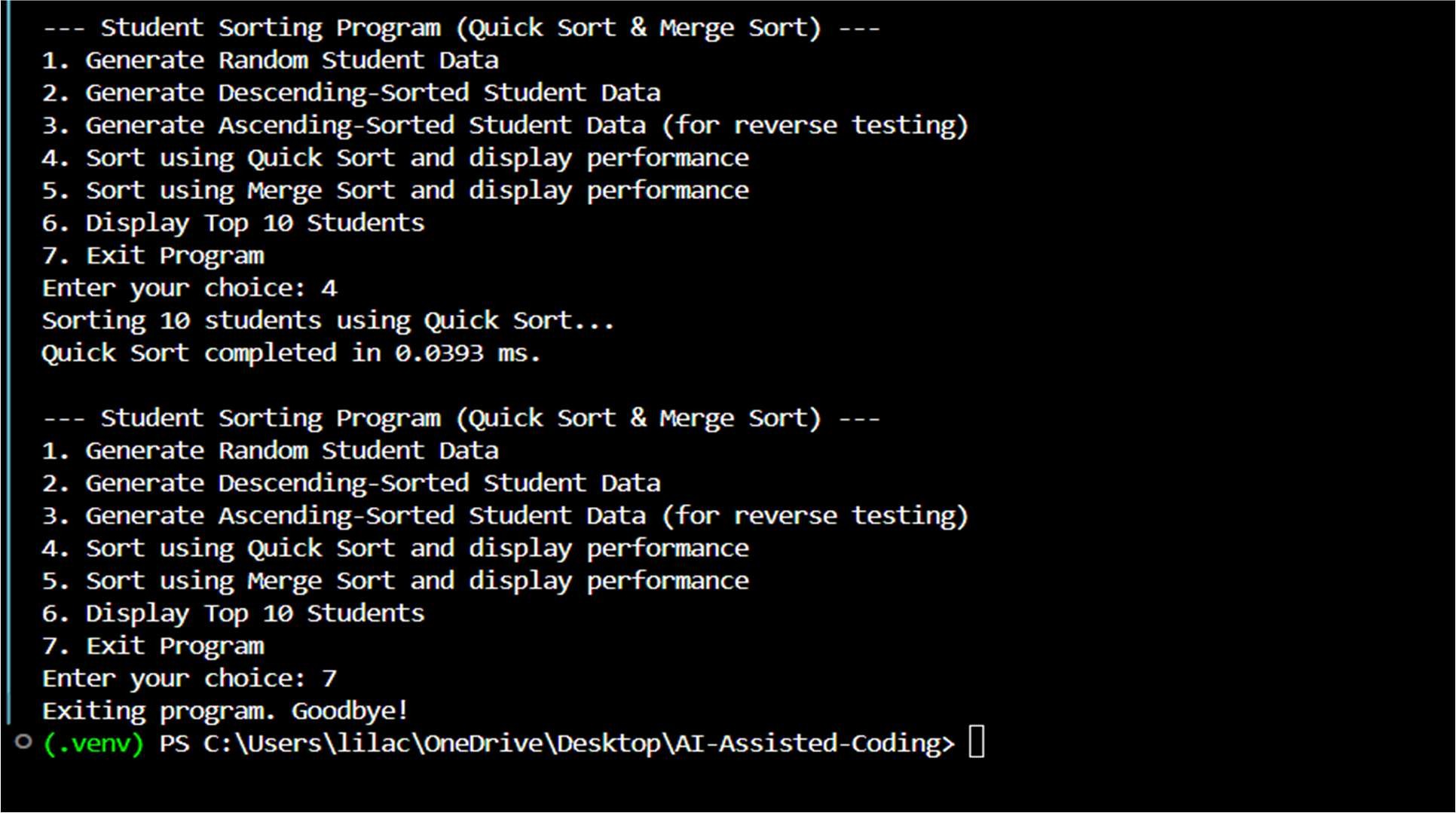
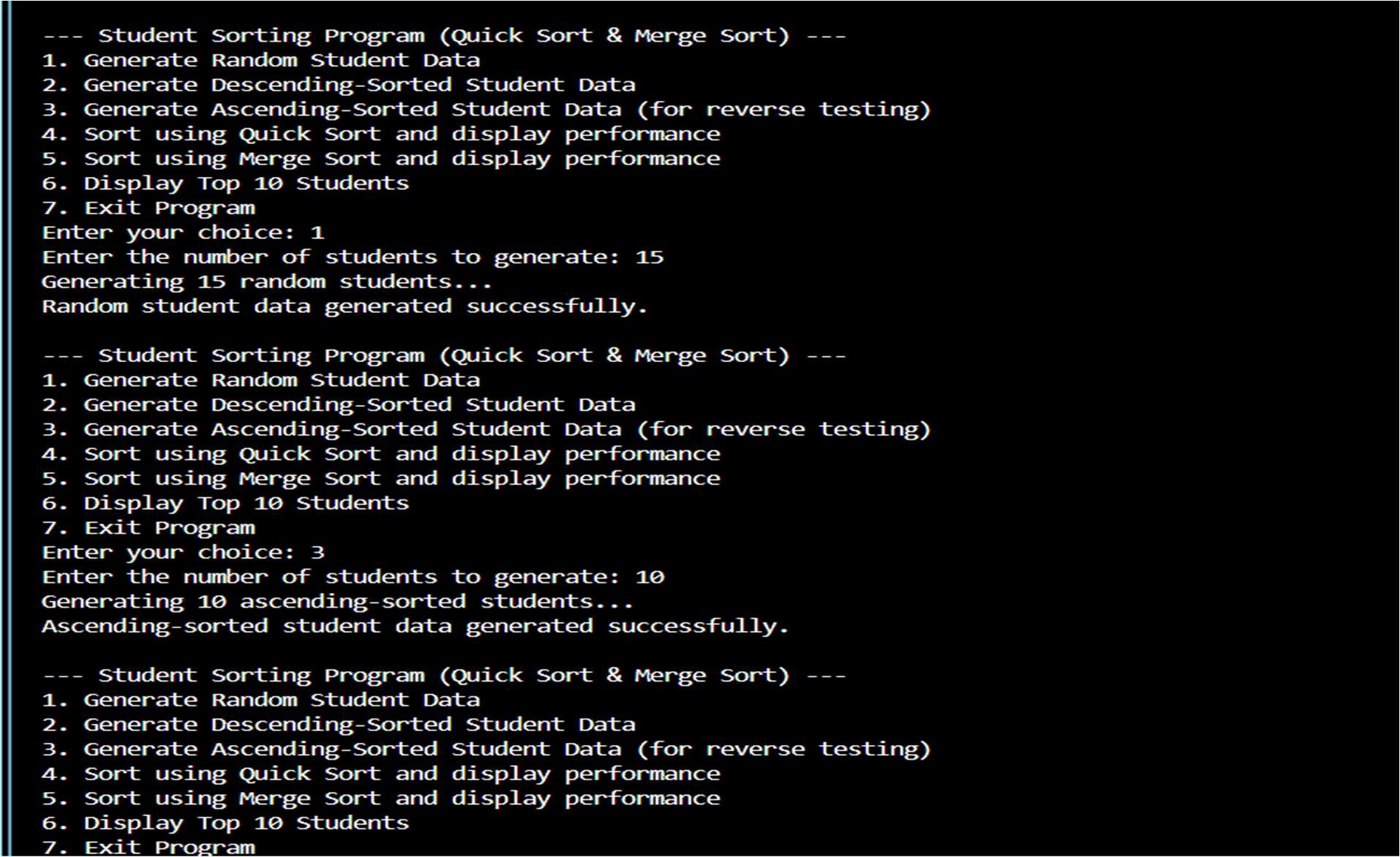
if not students:

|  |
| --- |
| print("No students to display.") return  print(f"\n--- Top 10 Students (after {sort\_method}) ---") for i, student in enumerate(students[:10]):  print(f"{i+1}. {student}") print("-----------------------")    # --- Main Program Loop (User Interface) --- def main():  student\_records = None sorted\_records = None current\_sort\_method = None num\_students\_generated = 0  while True:  print("\n--- Student Sorting Program (Quick Sort & Merge Sort) ---") print("1. Generate Random Student Data") print("2. Generate Descending-Sorted Student Data") print("3. Generate Ascending-Sorted Student Data (for reverse testing)") print("4. Sort using Quick Sort and display performance") print("5. Sort using Merge Sort and display performance") print("6. Display Top 10 Students") print("7. Exit Program") choice = input("Enter your choice: ")  if choice == '1':  try:  num\_students = int(input("Enter the number of students to generate: ")) if num\_students <= 0: raise ValueError print(f"Generating {num\_students} random students...") student\_records = generate\_random\_students(num\_students) num\_students\_generated = num\_students sorted\_records = None # Reset sorted data after new generation current\_sort\_method = None print("Random student data generated successfully.") except ValueError:  print("Invalid input. Please enter a positive integer for the number of students.") elif choice == '2': try:  num\_students = int(input("Enter the number of students to generate: ")) if num\_students <= 0: raise ValueError  print(f"Generating {num\_students} descending-sorted students...") student\_records = generate\_descending\_students(num\_students) num\_students\_generated = num\_students sorted\_records = None current\_sort\_method = None  print("Descending-sorted student data generated successfully.") except ValueError: |

print("Invalid input. Please enter a positive integer for the number of

|  |
| --- |
| students.") elif choice == '3': try:  num\_students = int(input("Enter the number of students to generate: ")) if num\_students <= 0: raise ValueError print(f"Generating {num\_students} ascending-sorted students...") student\_records = generate\_ascending\_students(num\_students) num\_students\_generated = num\_students sorted\_records = None current\_sort\_method = None print("Ascending-sorted student data generated successfully.") except ValueError:  print("Invalid input. Please enter a positive integer for the number of students.") elif choice == '4':  if student\_records is None:  print("Please generate student data first (Option 1, 2, or 3).") continue print(f"Sorting {num\_students\_generated} students using Quick Sort...")  # Quick Sort requires low and high indices  sorted\_records, time\_taken = measure\_performance(quick\_sort, student\_records,  0, len(student\_records) - 1)  current\_sort\_method = "Quick Sort" print(f"Quick Sort completed in {time\_taken:.4f} ms.") elif choice == '5':  if student\_records is None:  print("Please generate student data first (Option 1, 2, or 3).") continue print(f"Sorting {num\_students\_generated} students using Merge Sort...") sorted\_records, time\_taken = measure\_performance(merge\_sort, student\_records) current\_sort\_method = "Merge Sort" print(f"Merge Sort completed in {time\_taken:.4f} ms.") elif choice == '6':  if sorted\_records is None: if student\_records is None:  print("Please generate and then sort student data first (Options  1/2/3, then 4 or 5).") else:  print("Please sort the student data first (Option 4 or 5).") continue display\_top\_10\_students(sorted\_records, current\_sort\_method) elif choice == '7':  print("Exiting program. Goodbye!") break else:  print("Invalid choice. Please enter a number between 1 and 7.")  if \_\_name\_\_ == "\_\_main\_\_":  main() |

Output:



Explana on:

The code completes recursive Quick Sort and Merge Sort func ons, tests them on different input orders, and compares their performance behavior in various cases.

Task 4: Real-Time Applica on – Inventory Management System

Prompt:

Suggest efficient search and sort algorithms for a large inventory system, jus fy choices based on dataset size and performance, and implement the recommended search and sort func ons in Python.

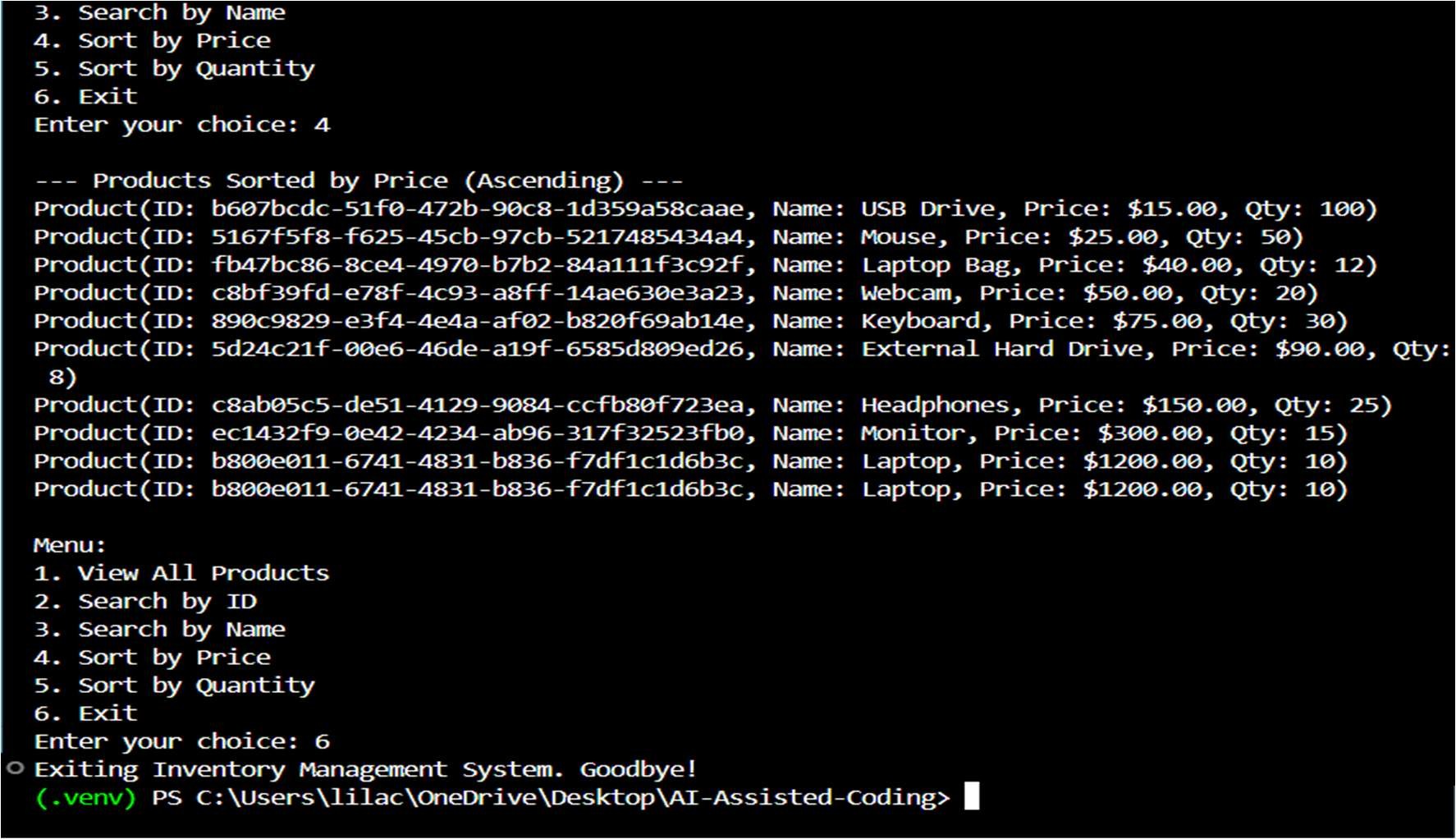
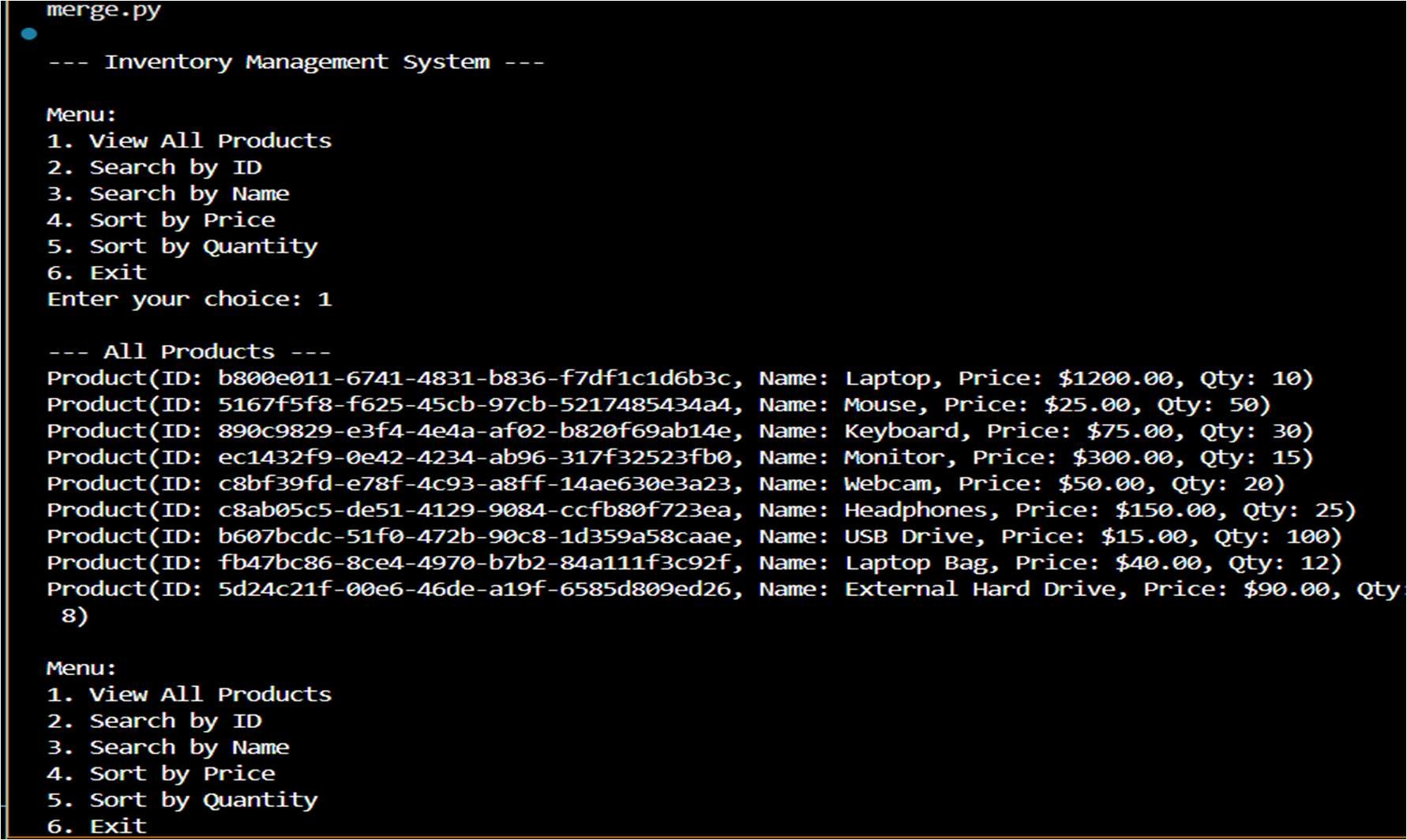
Code:

|  |
| --- |
| import uuid  class Product:  def \_\_init\_\_(self, product\_id, name, price, quantity):  self.product\_id = product\_id self.name = name self.price = price self.quantity = quantity  def \_\_repr\_\_(self):  return f"Product(ID: {self.product\_id}, Name: {self.name}, Price: ${self.price:.2f}, Qty: {self.quantity})"  def generate\_sample\_inventory():  products = [  Product(str(uuid.uuid4()), "Laptop", 1200.00, 10), Product(str(uuid.uuid4()), "Mouse", 25.00, 50),  Product(str(uuid.uuid4()), "Keyboard", 75.00, 30),  Product(str(uuid.uuid4()), "Monitor", 300.00, 15),  Product(str(uuid.uuid4()), "Webcam", 50.00, 20),  Product(str(uuid.uuid4()), "Headphones", 150.00, 25),  Product(str(uuid.uuid4()), "USB Drive", 15.00, 100),  Product(str(uuid.uuid4()), "Laptop Bag", 40.00, 12),  Product(str(uuid.uuid4()), "External Hard Drive", 90.00, 8)  ] return products  def search\_by\_id(inventory, product\_id\_to\_find):  id\_to\_product\_map = {product.product\_id: product for product in inventory} return id\_to\_product\_map.get(product\_id\_to\_find)  def search\_by\_name(inventory, name\_to\_find): matching\_products = []  name\_to\_find\_lower = name\_to\_find.lower() for product in inventory: if name\_to\_find\_lower in product.name.lower():  matching\_products.append(product) return matching\_products  def sort\_inventory\_by\_price(inventory):  return sorted(inventory, key=lambda product: product.price)  def sort\_inventory\_by\_quantity(inventory):  return sorted(inventory, key=lambda product: product.quantity)  def run\_inventory\_system(): |

|  |
| --- |
| inventory = generate\_sample\_inventory() print("\n--- Inventory Management System ---")  while True:  print("\nMenu:") print("1. View All Products") print("2. Search by ID") print("3. Search by Name") print("4. Sort by Price") print("5. Sort by Quantity") print("6. Exit")    choice = input("Enter your choice: ")  if choice == '1': if not inventory:  print("Inventory is empty.") else:  print("\n--- All Products ---") for product in inventory:  print(product) elif choice == '2':  product\_id = input("Enter Product ID to search: ") found\_product = search\_by\_id(inventory, product\_id) if found\_product:  print("\nProduct Found:") print(found\_product) else:  print(f"No product found with ID: {product\_id}") elif choice == '3':  product\_name = input("Enter Product Name (or part of it) to search: ") found\_products = search\_by\_name(inventory, product\_name) if found\_products:  print("\nMatching Products:") for product in found\_products:  print(product) else:  print(f"No products found matching name: {product\_name}") elif choice == '4':  sorted\_by\_price = sort\_inventory\_by\_price(inventory) print("\n--- Products Sorted by Price (Ascending) ---") for product in sorted\_by\_price:  print(product) elif choice == '5':  sorted\_by\_quantity = sort\_inventory\_by\_quantity(inventory) print("\n--- Products Sorted by Quantity (Ascending) ---") for product in sorted\_by\_quantity:  print(product) elif choice == '6':  print("Exiting Inventory Management System. Goodbye!") break else:  print("Invalid choice. Please try again.") |

if \_\_name\_\_ == '\_\_main\_\_': run\_inventory\_system()

Output:



Explana on:

The code recommends suitable search and sor ng methods for managing large product data, explains why they fit the system needs, and implements func ons to perform fast searching and sor ng.

Task 5: Real-Time Stock Data Sor ng & Searching

Prompt:

Simulate stock data (Symbol, Opening, Closing), calculate percentage change, sort stocks using Heap Sort by daily gain/loss, implement search using a Hash Map, compare with sorted() and dict lookups, and briefly analyze tradeoffs.

Code:

|  |
| --- |
| import random import string import time # Import the time module for performance measurement  # 1. Define the Stock class class Stock: def \_\_init\_\_(self, symbol, opening\_price, closing\_price):  self.symbol = symbol self.opening\_price = opening\_price self.closing\_price = closing\_price  def percentage\_change(self): if self.opening\_price == 0:  return float('inf') if self.closing\_price > 0 else 0.0 # Avoid division by zero return ((self.closing\_price - self.opening\_price) / self.opening\_price) \* 100  def \_\_repr\_\_(self): return f"Stock(Symbol: {self.symbol}, Open: {self.opening\_price:.2f}, Close:  {self.closing\_price:.2f}, Change: {self.percentage\_change():.2f}%)"    # For heap sort comparison (descending order by percentage\_change) - used for min-heap to build max-heap behavior  # For max-heap, we usually compare if parent is smaller than child.  # Python's heapq is a min-heap, so for custom objects, if we want max-heap behavior, we often negate the comparison key.  # However, our heap\_sort implementation is custom and builds a max-heap directly by comparing values.  # The \_\_lt\_\_ is primarily useful if we were to use Python's `sorted()` or  `list.sort()` directly without a key.  # For `heap\_sort` which directly compares `percentage\_change()`, this `\_\_lt\_\_` is not strictly used by `heapify`.  # Let's refine `\_\_lt\_\_` to support direct sorting for `sorted()` if needed, but for heap\_sort we rely on direct calls.  # For `heap\_sort` to arrange in descending order, our `heapify` compares  `arr[i].percentage\_change() < arr[left].percentage\_change()`  # and then `arr[largest].percentage\_change() < arr[right].percentage\_change()` which means it builds a max-heap.  # The final `arr[::-1]` reverses it to get descending order if the heap was built for ascending.  # Let's adjust the `\_\_lt\_\_` to make `sorted()` work for descending easily.  def \_\_lt\_\_(self, other): # This defines less than. For descending, we want greater to be less.  return self.percentage\_change() > other.percentage\_change() # Invert for descending order with standard sort |

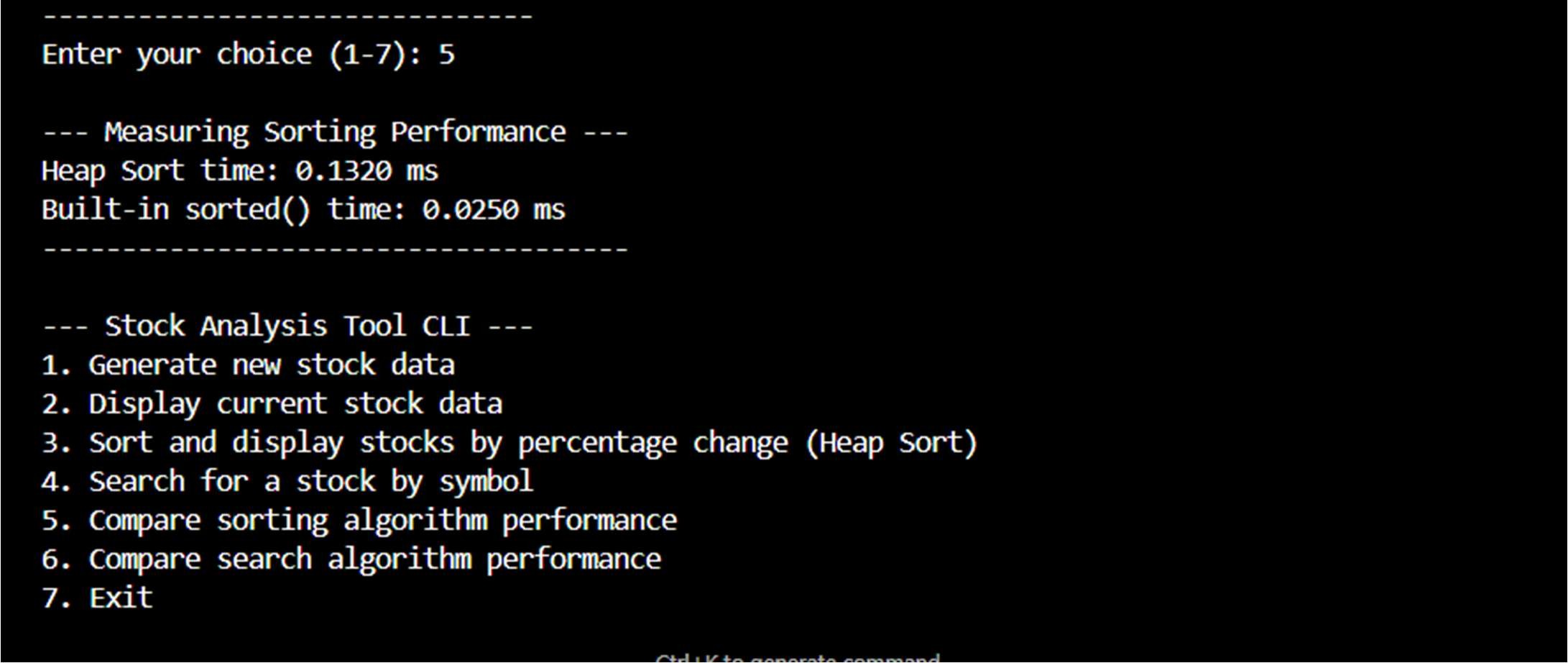
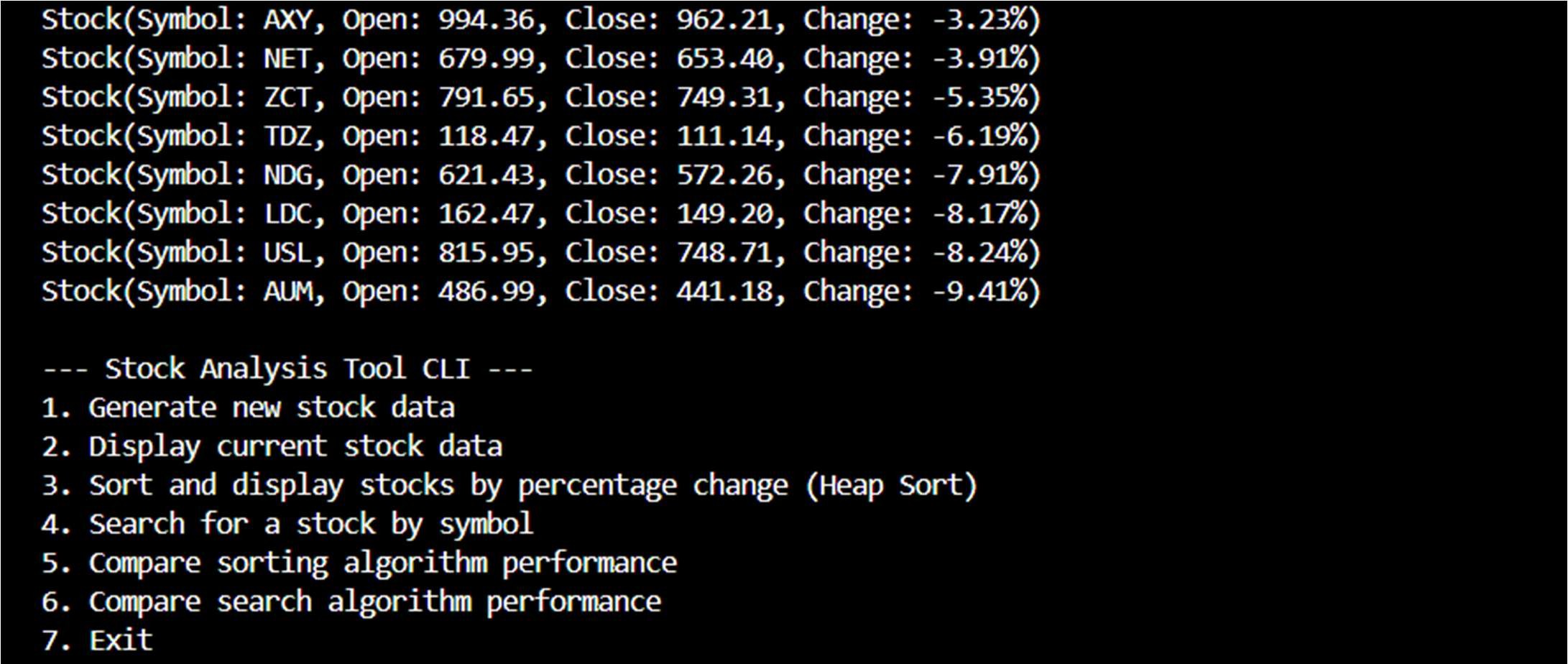
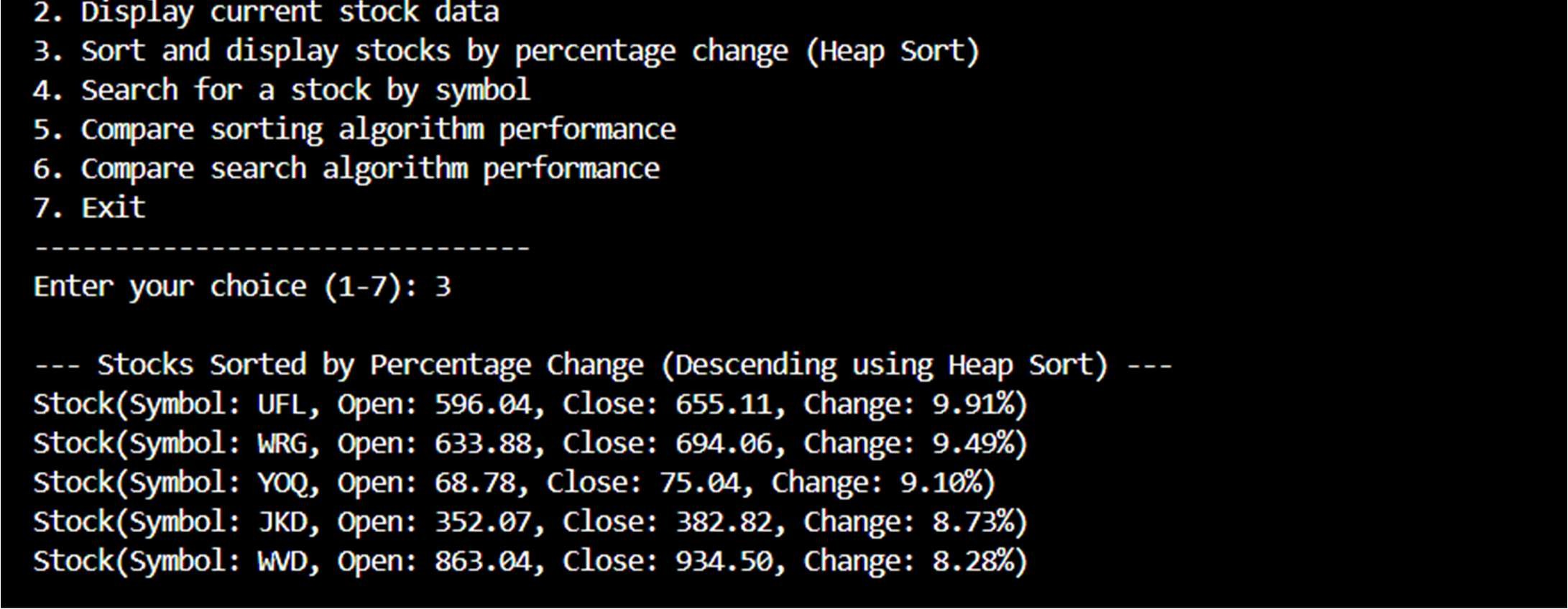
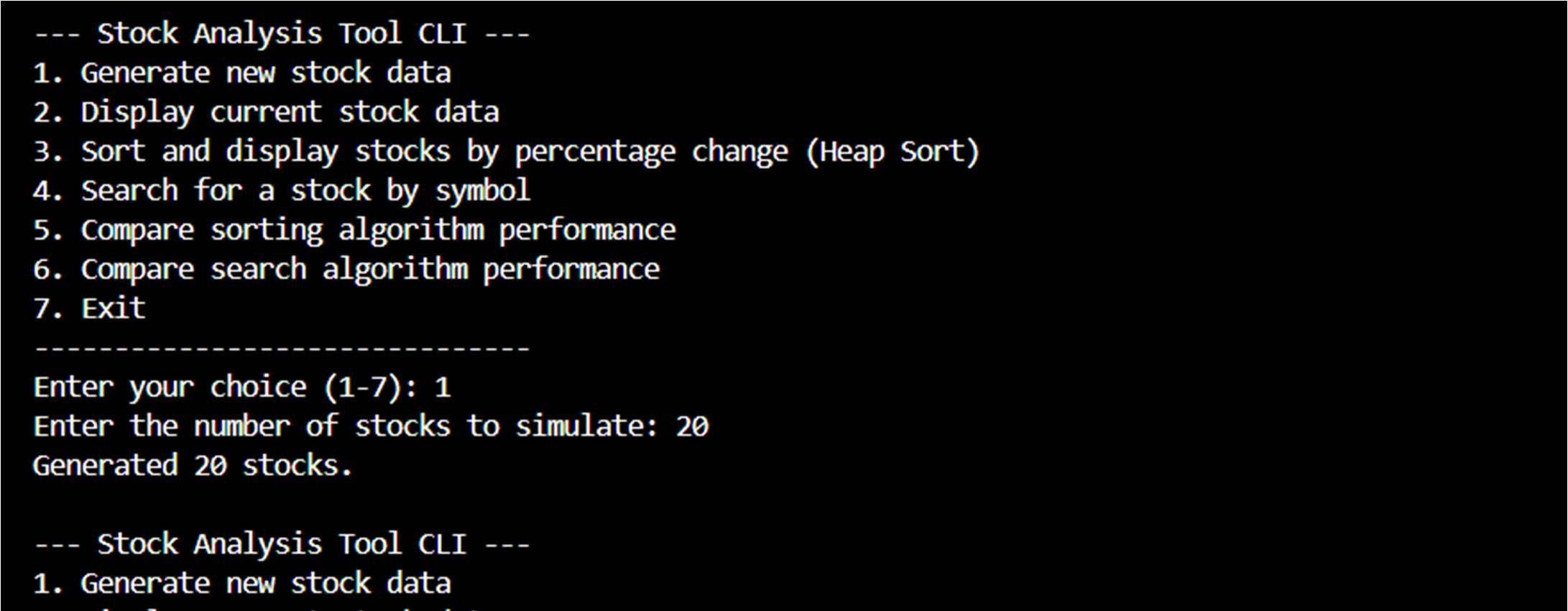
|  |
| --- |
| # 2. Function to simulate stock data def simulate\_stock\_data(num\_stocks):  stocks = [] for \_ in range(num\_stocks):  symbol = ''.join(random.choices(string.ascii\_uppercase, k=3)) opening\_price = round(random.uniform(10.0, 1000.0), 2) # Closing price within +/- 10% of opening price closing\_price = round(opening\_price \* random.uniform(0.9, 1.1), 2) stocks.append(Stock(symbol, opening\_price, closing\_price)) return stocks  # 3. Implement Heap Sort def heapify(arr, n, i):  largest = i # Initialize largest as root left = 2 \* i + 1 right = 2 \* i + 2    # See if left child of root exists and is greater than root (by percentage\_change) if left < n and arr[left].percentage\_change() > arr[largest].percentage\_change():  largest = left    # See if right child of root exists and is greater than current largest if right < n and arr[right].percentage\_change() > arr[largest].percentage\_change():  largest = right    # Change root, if needed if largest != i:  arr[i], arr[largest] = arr[largest], arr[i] # swap heapify(arr, n, largest) # Heapify the root.  def heap\_sort(arr): n = len(arr)  # Build a maxheap (rearrange array) to sort in descending order for i in range(n // 2 - 1, -1, -1):  heapify(arr, n, i)    # One by one extract elements from maxheap for i in range(n - 1, 0, -1):  arr[i], arr[0] = arr[0], arr[i] # swap root (largest) with last element heapify(arr, i, 0) # Call heapify on the reduced heap    # The heapify process places largest elements at the end, resulting in ascending order # for the `arr` itself after the loop. To get descending order of percentage change,  # we need to reverse the result. Or, the heapify should be adapted to build a min-heap  # and extract elements, or make comparison inverse.  # Given `heapify` creates a max-heap (largest at root) and then puts largest at end,  # the array `arr` will be sorted in ASCENDING order after the loop.  # So, we reverse it to get DESCENDING order of percentage change.  return arr[::-1] |

|  |
| --- |
| # 4. Create a function to build a hash map def build\_hash\_map(stocks):  stock\_map = {} for stock in stocks:  stock\_map[stock.symbol] = stock return stock\_map  # 5. Implement a function for hash map lookup def hash\_map\_lookup(hash\_map, symbol):  return hash\_map.get(symbol)  # New: Linear Search Function def linear\_search(stocks, symbol):  for stock in stocks: if stock.symbol == symbol:  return stock return None    # New: Performance Measurement Functions def measure\_sorting\_performance(stocks\_to\_sort):  print("\n--- Measuring Sorting Performance ---") if not stocks\_to\_sort:  print("No stock data to sort. Please generate data first.") return    # Heap Sort  start\_time = time.perf\_counter() heap\_sorted\_stocks = heap\_sort(list(stocks\_to\_sort)) # Pass a copy end\_time = time.perf\_counter()  heap\_sort\_time = (end\_time - start\_time) \* 1000 print(f"Heap Sort time: {heap\_sort\_time:.4f} ms")  # print(f"First 5 Heap Sorted: {heap\_sorted\_stocks[:5]}") # Optional: print a few to verify    # Python's built-in sorted() start\_time = time.perf\_counter()  # Use key for sorting by percentage\_change in descending order builtin\_sorted\_stocks = sorted(list(stocks\_to\_sort), key=lambda stock:  stock.percentage\_change(), reverse=True) end\_time = time.perf\_counter()  builtin\_sort\_time = (end\_time - start\_time) \* 1000  print(f"Built-in sorted() time: {builtin\_sort\_time:.4f} ms")  # print(f"First 5 Built-in Sorted: {builtin\_sorted\_stocks[:5]}") # Optional: print a few to verify    print("-------------------------------------")  def measure\_search\_performance(stocks, stock\_map): print("\n--- Measuring Search Performance ---") if not stocks:  print("No stock data to search. Please generate data first.") return |

|  |
| --- |
| # Select a random symbol to search for, ensuring it exists for fair comparison search\_symbol\_exists = random.choice(stocks).symbol search\_symbol\_non\_existent = ''.join(random.choices(string.ascii\_uppercase, k=3)) + 'X'  # Ensure non\_existent is truly non-existent while search\_symbol\_non\_existent in stock\_map:  search\_symbol\_non\_existent = ''.join(random.choices(string.ascii\_uppercase, k=3))  + 'X'    # --- Existing Stock Search --- print(f"Searching for an existing symbol: {search\_symbol\_exists}")  # Hash Map Lookup start\_time = time.perf\_counter() found\_by\_hash = hash\_map\_lookup(stock\_map, search\_symbol\_exists) end\_time = time.perf\_counter() hash\_map\_time = (end\_time - start\_time) \* 1000 print(f"Hash Map lookup time (found): {hash\_map\_time:.6f} ms")  # Linear Search start\_time = time.perf\_counter()  found\_by\_linear = linear\_search(stocks, search\_symbol\_exists) end\_time = time.perf\_counter()  linear\_search\_time = (end\_time - start\_time) \* 1000 print(f"Linear Search time (found): {linear\_search\_time:.6f} ms")  # --- Non-Existing Stock Search ---  print(f"\nSearching for a non-existent symbol: {search\_symbol\_non\_existent}")  # Hash Map Lookup start\_time = time.perf\_counter() not\_found\_by\_hash = hash\_map\_lookup(stock\_map, search\_symbol\_non\_existent) end\_time = time.perf\_counter() hash\_map\_time\_nf = (end\_time - start\_time) \* 1000 print(f"Hash Map lookup time (not found): {hash\_map\_time\_nf:.6f} ms")  # Linear Search start\_time = time.perf\_counter()  not\_found\_by\_linear = linear\_search(stocks, search\_symbol\_non\_existent) end\_time = time.perf\_counter() linear\_search\_time\_nf = (end\_time - start\_time) \* 1000 print(f"Linear Search time (not found): {linear\_search\_time\_nf:.6f} ms")  print("----------------------------------")  # Interactive Command-Line Interface  simulated\_stocks = [] stock\_lookup\_map = {}  def display\_menu():  print("\n--- Stock Analysis Tool CLI ---") |

|  |
| --- |
| print("1. Generate new stock data") print("2. Display current stock data") print("3. Sort and display stocks by percentage change (Heap Sort)") print("4. Search for a stock by symbol") print("5. Compare sorting algorithm performance") print("6. Compare search algorithm performance") print("7. Exit") print("-------------------------------")  while True:  display\_menu() choice = input("Enter your choice (1-7): ")  if choice == '1': try:  num\_stocks = int(input("Enter the number of stocks to simulate: ")) if num\_stocks <= 0:  print("Please enter a positive number.") continue simulated\_stocks = simulate\_stock\_data(num\_stocks) stock\_lookup\_map = build\_hash\_map(simulated\_stocks) # Rebuild map for new data print(f"Generated {num\_stocks} stocks.") except ValueError:  print("Invalid input. Please enter a number.")  elif choice == '2': if simulated\_stocks:  print("\n--- Current Stock Data ---") for stock in simulated\_stocks:  print(stock) else:  print("No stock data available. Please generate data first (Option 1).")  elif choice == '3': if simulated\_stocks:  print("\n--- Stocks Sorted by Percentage Change (Descending using Heap Sort) --")  # Heap sort modifies the list in place, so we pass a copy for consistent behavior.  # The heap\_sort function returns a reversed list, which is the desired descending order.  sorted\_stocks\_heap = heap\_sort(list(simulated\_stocks)) for stock in sorted\_stocks\_heap:  print(stock) else:  print("No stock data available to sort. Please generate data first (Option  1).") elif choice == '4':  if simulated\_stocks and stock\_lookup\_map:  symbol = input("Enter stock symbol to search: ").upper() found\_stock = hash\_map\_lookup(stock\_lookup\_map, symbol) if found\_stock:  print(f"Found stock: {found\_stock}") |
| else:  print(f"Stock with symbol '{symbol}' not found.") else:  print("No stock data available for search. Please generate data first (Option  1).") elif choice == '5':  measure\_sorting\_performance(simulated\_stocks)  elif choice == '6':  measure\_search\_performance(simulated\_stocks, stock\_lookup\_map)  elif choice == '7':  print("Exiting Stock Analysis Tool. Goodbye!") break  else:  print("Invalid choice. Please enter a number between 1 and 7.") |

Output:



Explana on:

The code generates stock data, ranks stocks based on percentage change using Heap Sort, and retrieves stock details quickly using a hash map, then compares custom methods with built-in Python func ons.