Task

Implement Student record data structures, generate Quick Sort, Merge Sort, and Bubble Sort algorithms (with inline comments for Bubble Sort). Generate a large dataset of student records, compare the performance of Quick Sort and Merge Sort on this dataset, and display the top 10 students using a dedicated function. Finally, provide a time complexity analysis for Bubble Sort.

Implement Student Record Data Structure

Subtask:

Generate Python code to define a `Student` class or a dictionary-based structure to store student records with 'name', 'roll_number', and 'cgpa' attributes. Include code to create a sample list of these student records.

Reasoning: The subtask requires defining a `Student` class and creating a sample list of student records. This can be achieved with a single Python code block that includes the class definition, the `__init__` and `__repr__` methods, and the creation of the `sample_students` list.

```
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})"

# Create a sample list of student records
sample_students = [
    Student("Alice Smith", 101, 3.8),
    Student("Bob Johnson", 102, 3.5),
    Student("Charlie Brown", 103, 3.9),
    Student("Diana Prince", 104, 4.0),
    Student("Eve Adams", 105, 3.7)
]

print("Sample Student Records:")
for student in sample_students:
    print(student)
```

```
Sample Student Records:
Student(Name: Alice Smith, Roll: 101, CGPA: 3.8)
Student(Name: Bob Johnson, Roll: 102, CGPA: 3.5)
Student(Name: Charlie Brown, Roll: 103, CGPA: 3.9)
Student(Name: Diana Prince, Roll: 104, CGPA: 4.0)
Student(Name: Eve Adams, Roll: 105, CGPA: 3.7)
```

Implement Quick Sort Algorithm

Subtask:

Generate Python code for the Quick Sort algorithm. This function should be able to sort a list of `Student` objects in descending order based on their `cgpa` attribute.

Reasoning: I need to implement the Quick Sort algorithm, including a partition helper, to sort a list of `Student` objects by `cgpa` in descending order. Then, I will apply it to the `sample_students` list and print the result.

```
def quick_sort_students(students):
    if len(students) <= 1:
        return students
    pivot = students[len(students) // 2]
    less = [s for s in students if s.cgpa > pivot.cgpa]
    equal = [s for s in students if s.cgpa == pivot.cgpa]
    greater = [s for s in students if s.cgpa < pivot.cgpa]
```

```
    return quick_sort_students(less) + equal + quick_sort_students(greater)

# Test the Quick Sort function
sorted_students_quick_sort = quick_sort_students(list(sample_students))

print("\nStudent Records sorted by CGPA (Descending) using Quick Sort:")
for student in sorted_students_quick_sort:
    print(student)
```

```
Student Records sorted by CGPA (Descending) using Quick Sort:
Student(Name: Diana Prince, Roll: 104, CGPA: 4.0)
Student(Name: Charlie Brown, Roll: 103, CGPA: 3.9)
Student(Name: Alice Smith, Roll: 101, CGPA: 3.8)
Student(Name: Eve Adams, Roll: 105, CGPA: 3.7)
Student(Name: Bob Johnson, Roll: 102, CGPA: 3.5)
```

Implement Merge Sort Algorithm

Subtask:

Generate Python code for the Merge Sort algorithm. This function should be able to sort a list of `Student` objects in descending order based on their `cgpa` attribute.

Reasoning: The subtask requires implementing the Merge Sort algorithm, including a recursive function and a merge helper function, to sort `Student` objects by `cgpa` in descending order. After defining these, I will apply it to the `sample_students` list and print the result.

```
def merge_sort_students(students):
    if len(students) <= 1:
        return students

    mid = len(students) // 2
    left_half = students[:mid]
    right_half = students[mid:]

    left_half = merge_sort_students(left_half)
    right_half = merge_sort_students(right_half)

    return merge(left_half, right_half)

def merge(left, right):
    merged = []
    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:  # Descending order
            merged.append(left[left_idx])
            left_idx += 1
        else:
            merged.append(right[right_idx])
            right_idx += 1

    while left_idx < len(left):
        merged.append(left[left_idx])
        left_idx += 1

    while right_idx < len(right):
        merged.append(right[right_idx])
        right_idx += 1

    return merged

# Test the Merge Sort function
sorted_students_merge_sort = merge_sort_students(list(sample_students))

print("\nStudent Records sorted by CGPA (Descending) using Merge Sort:")
for student in sorted_students_merge_sort:
    print(student)
```

```
Student Records sorted by CGPA (Descending) using Merge Sort:
Student(Name: Diana Prince, Roll: 104, CGPA: 4.0)
```

```
Student(Name: Charlie Brown, Roll: 103, CGPA: 3.9)
Student(Name: Alice Smith, Roll: 101, CGPA: 3.8)
Student(Name: Eve Adams, Roll: 105, CGPA: 3.7)
Student(Name: Bob Johnson, Roll: 102, CGPA: 3.5)
```

Generate Large Dataset & Compare Performance

Subtask:

Generate a Python function to create a large dataset (e.g., 10,000 to 50,000 records) of `Student` objects with random CGPA values. Implement code to measure and compare the execution time of Quick Sort and Merge Sort on this large dataset, then display the runtimes clearly.

Reasoning: The subtask requires generating a large dataset of Student objects, then measuring and comparing the performance of Quick Sort and Merge Sort on this dataset. This involves importing necessary modules, defining a dataset generation function, and then timing the execution of both sorting algorithms.

```
import random
import time

def generate_large_students_dataset(num_students):
    students = []
    for i in range(num_students):
        name = f"Student {i+1}"
        roll_number = 1000 + i  # Ensure unique roll numbers
        cgpa = round(random.uniform(2.0, 4.0), 2)
        students.append(Student(name, roll_number, cgpa))
    return students

# Generate a large dataset
num_students_large = 20000
large_students_dataset = generate_large_students_dataset(num_students_large)
print(f"\nGenerated a large dataset of {len(large_students_dataset)} student records.")

# Measure execution time for Quick Sort
students_for_quick_sort = list(large_students_dataset) # Create a copy
start_time_quick = time.time()
sorted_students_quick_large = quick_sort_students(students_for_quick_sort)
end_time_quick = time.time()
time_quick_sort = end_time_quick - start_time_quick
print(f"Quick Sort execution time for {num_students_large} students: {time_quick_sort:.4f} seconds")

# Measure execution time for Merge Sort
students_for_merge_sort = list(large_students_dataset) # Create another copy
start_time_merge = time.time()
sorted_students_merge_large = merge_sort_students(students_for_merge_sort)
end_time_merge = time.time()
time_merge_sort = end_time_merge - start_time_merge
print(f"Merge Sort execution time for {num_students_large} students: {time_merge_sort:.4f} seconds")

print("\nPerformance comparison complete.")
```

```

Generated a large dataset of 20000 student records.
Quick Sort execution time for 20000 students: 0.0396 seconds
Merge Sort execution time for 20000 students: 0.1629 seconds

Performance comparison complete.
```

Implement Bubble Sort Algorithm

Subtask:

Generate Python code for the Bubble Sort algorithm. This function should be able to sort a list of `Student` objects in descending order based on their `cgpa` attribute. Include inline comments explaining the logic of the Bubble Sort algorithm.

Reasoning: I will implement the Bubble Sort algorithm, which involves iterating through the list multiple times, comparing adjacent elements, and swapping them if they are in the wrong order. For sorting `Student` objects by `cgpa` in descending order, I will compare

`cgpa` values and perform swaps such that students with higher CGPA appear earlier in the list. Inline comments will be added to explain each step of the Bubble Sort logic.

Reasoning: I need to implement the Bubble Sort algorithm for a list of Student objects, sorting them in descending order based on their CGPA, and add inline comments to explain the logic as requested.

```
def bubble_sort_students(students):
    n = len(students)
    # Outer loop for passes through the list
    for i in range(n - 1):
        # Inner loop for comparisons and swaps
        for j in range(0, n - i - 1):
            # Compare adjacent students based on CGPA
            # For descending order, swap if the current student's CGPA is less than the next student's CGPA
            if students[j].cgpa < students[j+1].cgpa:
                # Swap the elements if they are in the wrong order
                students[j], students[j+1] = students[j+1], students[j]
    return students

# Test the Bubble Sort function with a copy of sample_students
sorted_students_bubble_sort = bubble_sort_students(list(sample_students))

print("\nStudent Records sorted by CGPA (Descending) using Bubble Sort:")
for student in sorted_students_bubble_sort:
    print(student)
```

```
Student Records sorted by CGPA (Descending) using Bubble Sort:
Student(Name: Diana Prince, Roll: 104, CGPA: 4.0)
Student(Name: Charlie Brown, Roll: 103, CGPA: 3.9)
Student(Name: Alice Smith, Roll: 101, CGPA: 3.8)
Student(Name: Eve Adams, Roll: 105, CGPA: 3.7)
Student(Name: Bob Johnson, Roll: 102, CGPA: 3.5)
```

Display Top 10 Students

Subtask:

Generate a dedicated Python function to display the top 10 students from a given sorted list of `Student` objects. Apply this function to the large dataset sorted by CGPA (e.g., using the Quick Sort result) and print the top 10 students.

Reasoning: I need to define a function `display_top_n_students` that takes a list of `Student` objects (assumed to be already sorted by CGPA in descending order) and an integer `n` (defaulting to 10). The function will then iterate through the first `n` students in the list and print their details. Finally, I will call this function with the `sorted_students_quick_large` list to demonstrate its functionality.

Reasoning: I need to define the `display_top_n_students` function as outlined in the reasoning, apply it to the `sorted_students_quick_large` dataset, and print the top 10 students.

```
def display_top_n_students(students, n=10):
    print(f"\nTop {min(n, len(students))} Students by CGPA:")
    for i, student in enumerate(students[:n]):
        print(f"{i+1}. {student}")

# Apply the function to the large dataset sorted by Quick Sort
display_top_n_students(sorted_students_quick_large, n=10)
```

```
Top 10 Students by CGPA:
1. Student(Name: Student 78, Roll: 1077, CGPA: 4.0)
2. Student(Name: Student 531, Roll: 1530, CGPA: 4.0)
3. Student(Name: Student 1351, Roll: 2350, CGPA: 4.0)
4. Student(Name: Student 1593, Roll: 2592, CGPA: 4.0)
5. Student(Name: Student 1707, Roll: 2706, CGPA: 4.0)
6. Student(Name: Student 1877, Roll: 2876, CGPA: 4.0)
7. Student(Name: Student 1926, Roll: 2925, CGPA: 4.0)
8. Student(Name: Student 2114, Roll: 3113, CGPA: 4.0)
9. Student(Name: Student 2850, Roll: 3849, CGPA: 4.0)
10. Student(Name: Student 3310, Roll: 4309, CGPA: 4.0)
```

Time Complexity Analysis for Bubble Sort

Subtask:

Provide a time complexity analysis for the Bubble Sort algorithm, explaining its best, average, and worst-case scenarios.

```
**Reasoning**:
I need to explain the time complexity of the Bubble Sort algorithm, detailing its performance in best, average, and worst-case sc

**Time Complexity Analysis for Bubble Sort**:

*   **Best Case: O(n)**
    *   The best-case scenario occurs when the array is already sorted. In this situation, Bubble Sort will still iterate through

*   **Average Case: O(n^2)**
    *   In the average case, where elements are in a random order, Bubble Sort requires approximately `n/2` passes, and in each p

*   **Worst Case: O(n^2)**
    *   The worst-case scenario happens when the array is sorted in reverse order. In this situation, every comparison leads to a

**Summary**:
Bubble Sort is generally considered inefficient for large datasets due to its quadratic time complexity in average and worst-case
```


```
**Reasoning**:
I need to explain the time complexity of the Bubble Sort algorithm, detailing its performance in best, average, and worst-case sc

**Time Complexity Analysis for Bubble Sort**:

*   **Best Case: O(n)**
    *   The best-case scenario occurs when the array is already sorted. In this situation, Bubble Sort will still iterate through

*   **Average Case: O(n^2)**
    *   In the average case, where elements are in a random order, Bubble Sort requires approximately `n/2` passes, and in each p

*   **Worst Case: O(n^2)**
    *   The worst-case scenario happens when the array is sorted in reverse order. In this situation, every comparison leads to a

**Summary**:
Bubble Sort is generally considered inefficient for large datasets due to its quadratic time complexity in average and worst-case
```


Reasoning: I need to explain the time complexity of the Bubble Sort algorithm, detailing its performance in best, average, and worst-case scenarios. This analysis is crucial for understanding the efficiency of the algorithm, especially when compared to Quick Sort and Merge Sort.

Time Complexity Analysis for Bubble Sort:

- **Best Case: O(n)**
  - The best-case scenario occurs when the array is already sorted. In this situation, Bubble Sort will still iterate through the entire list once to confirm that no swaps are needed. The inner loop will execute `n-1` times in the first pass, but no swaps will occur. An optimized version of Bubble Sort can detect this state by checking if any swaps happened in a pass; if not, it can terminate early. Thus, only one full pass is required, making the time complexity linear, O(n).
- **Average Case: O(n^2)**
  - In the average case, where elements are in a random order, Bubble Sort requires approximately `n/2` passes, and in each pass, about `n/2` comparisons and swaps occur. This results in a quadratic time complexity. The number of comparisons is roughly proportional to `n^2`, specifically `(n * (n - 1)) / 2` comparisons.
- **Worst Case: O(n^2)**

- The worst-case scenario happens when the array is sorted in reverse order. In this situation, every comparison leads to a swap, and each element must 'bubble up' to its correct position through many passes. For an array of `n` elements, there will be `n-1` passes. In the first pass, `n-1` comparisons are made. In the second, `n-2`, and so on. The total number of comparisons and swaps is approximately `(n * (n - 1)) / 2`, which simplifies to O(n^2).

Summary: Bubble Sort is generally considered inefficient for large datasets due to its quadratic time complexity in average and worst-case scenarios. While its simplicity makes it easy to understand and implement, it is rarely used for practical sorting applications where performance is critical.

Reasoning: I need to explain the time complexity of the Bubble Sort algorithm, detailing its performance in best, average, and worst-case scenarios. This analysis is crucial for understanding the efficiency of the algorithm, especially when compared to Quick Sort and Merge Sort.

Time Complexity Analysis for Bubble Sort:

- **Best Case: O(n)**
  - The best-case scenario occurs when the array is already sorted. In this situation, Bubble Sort will still iterate through the entire list once to confirm that no swaps are needed. The inner loop will execute `n-1` times in the first pass, but no swaps will occur. An optimized version of Bubble Sort can detect this state by checking if any swaps happened in a pass; if not, it can terminate early. Thus, only one full pass is required, making the time complexity linear, O(n).
- **Average Case: O(n^2)**
  - In the average case, where elements are in a random order, Bubble Sort requires approximately `n/2` passes, and in each pass, about `n/2` comparisons and swaps occur. This results in a quadratic time complexity. The number of comparisons is roughly proportional to `n^2`, specifically `(n * (n - 1)) / 2` comparisons.
- **Worst Case: O(n^2)**
  - The worst-case scenario happens when the array is sorted in reverse order. In this situation, every comparison leads to a swap, and each element must 'bubble up' to its correct position through many passes. For an array of `n` elements, there will be `n-1` passes. In the first pass, `n-1` comparisons are made. In the second, `n-2`, and so on. The total number of comparisons and swaps is approximately `(n * (n - 1)) / 2`, which simplifies to O(n^2).

Summary: Bubble Sort is generally considered inefficient for large datasets due to its quadratic time complexity in average and worst-case scenarios. While its simplicity makes it easy to understand and implement, it is rarely used for practical sorting applications where performance is critical.

Final Task

Subtask:

Present the completed code for all sorting algorithms, the performance comparison results, the output of the top 10 students, and the Bubble Sort time complexity analysis.

Summary:

Data Analysis Key Findings

- The `Student` class was successfully defined, allowing the creation and representation of student records with 'name', 'roll_number', and 'cgpa'.
- Both Quick Sort and Merge Sort algorithms were correctly implemented to sort `Student` objects in descending order based on their CGPA.
- For a large dataset of 20,000 student records:
  - Quick Sort completed in approximately 0.0396 seconds.
  - Merge Sort completed in approximately 0.1629 seconds.
  - Quick Sort demonstrated significantly faster performance than Merge Sort for this dataset size.
- The Bubble Sort algorithm was successfully implemented and verified to sort student records by CGPA in descending order on a sample dataset.
- A dedicated function `display_top_n_students` was successfully implemented and used to present the top 10 students from the large dataset.

- The time complexity analysis for Bubble Sort confirmed its inefficiency for large datasets:
  - Best Case: O(n) (with optimization).
  - Average Case: O(n^2).
  - Worst Case: O(n^2).

Insights or Next Steps

- For sorting large datasets of student records, Quick Sort is the preferred algorithm over Merge Sort due to its superior performance observed in the comparison (0.0396s vs 0.1629s for 20,000 records).
- Given Bubble Sort's quadratic time complexity in average and worst-case scenarios, it should be avoided for sorting large student datasets where performance is critical.