TORIGAN

Data Structures and Algorithms

Principles and Applications for Modern Computing

Viraj Shakya Samaranayake

Table of Contents

Table of Contents

7.1 Bubble Sort

Bubble Sort is one of the earliest and simplest sorting algorithms introduced in the field of computer science. Although it is not efficient for handling large datasets, it holds significant pedagogical value in illustrating the fundamentals of comparison-based sorting. It provides a clear foundation for understanding basic principles such as iteration, element swapping, and algorithmic optimization.

The name Bubble Sort is derived from the characteristic behavior of elements during the sorting process: larger elements progressively "bubble up" to their correct positions at the end of the array, much like bubbles rising to the surface of water.

The algorithm operates by repeatedly traversing the list of elements and comparing each pair of adjacent items. If the first element in the pair is greater than the second, the two elements are swapped. This comparison and swapping process is carried out across the entire list. With each complete pass through the list, the next largest element is placed in its final, sorted position at the end. The algorithm continues to make successive passes until a complete iteration occurs without any swaps, indicating that the list is fully sorted.

Scientific Definition

Bubble Sort is a **stable, in-place, comparison-based sorting algorithm**. It operates by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. This process continues until no more swaps are needed, indicating that the list is sorted. Its **worst-case and average-case time complexity is $O(n^2)$**, and the **best-case time complexity is $O(n)$**, which occurs when the input is already sorted. The algorithm is **stable** because it preserves the relative order of equal elements, and it performs **$O(n^2)$** comparisons and up to **$O(n^2)$** swaps in the worst case.

One Sentence Description

Bubble Sort is a simple sorting algorithm that repeatedly compares and swaps adjacent elements until the list is sorted.

How It Works (Detailed Explanation)

Assume we are sorting an array of integers in ascending order.

1. First Pass:
   - Start from the first index.
   - Compare the first and second elements.
   - If the first is greater, swap them.
   - Move to the next pair and repeat until the end of the array.
   - After this pass, the largest element is at the end.
2. Subsequent Passes:
   - Repeat the process for the remaining elements (excluding the already sorted last elements).
   - With each pass, the next largest element is moved to its correct position.

Note on Optimization:
- If during a complete pass no swaps are made, the array is already sorted, and the algorithm terminates early.
- This optimization improves the best-case performance to O(n).

Let's take an example array: **[5, 3, 8, 4, 2]**

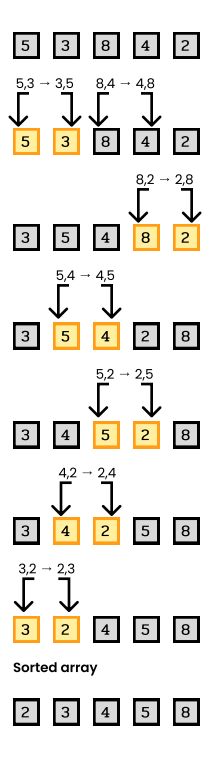
Pass 1
Compare 5 & 3 → swap → [3, 5, 8, 4, 2]
Compare 5 & 8 → no swap
Compare 8 & 4 → swap → [3, 5, 4, 8, 2]
Compare 8 & 2 → swap → [3, 5, 4, 2, 8]

Pass 2
Compare 3 & 5 → no swap
Compare 5 & 4 → swap → [3, 4, 5, 2, 8]
Compare 5 & 2 → swap → [3, 4, 2, 5, 8]

Pass 3
Compare 3 & 4 → no swap
Compare 4 & 2 → swap → [3, 2, 4, 5, 8]

Pass 4
Compare 3 & 2 → swap → [2, 3, 4, 5, 8]

Final Result: [2, 3, 4, 5, 8]



Java Implementation

```java
import java.util.Arrays;

public class BubbleSort {

    static int[] sort(int[] arr) {
        int n = arr.length;
        // Create a copy to avoid modifying the original array
        int[] result = Arrays.copyOf(arr, n);
        boolean swapped;

        for (int i = 0; i < n - 1; i++) {
            swapped = false;

            for (int j = 0; j < n - i - 1; j++) {
                if (result[j] > result[j + 1]) {
                    // Swap result[j] and result[j+1]
                    int temp = result[j];
                    result[j] = result[j + 1];
                    result[j + 1] = temp;
                    swapped = true;
                }
            }

            // If no two elements were swapped, the array is sorted
            if (!swapped)
                break;
        }

        return result;
    }

    public static void main(String[] args) {
        int[] arr = {5, 3, 8, 4, 2};
        System.out.println("Before: " + Arrays.toString(arr));
        int[] sortedArr = sort(arr);
        System.out.println("After: " + Arrays.toString(sortedArr));
    }
}
```

Output

```
Before: [5, 3, 8, 4, 2]
After:  [2, 3, 4, 5, 8]
```

Explanation

The program starts by calculating the length of the array. The outer loop runs n-1 times because we need at most n-1 passes to sort n elements. In each iteration of the inner loop, two adjacent values are compared. If the first value is greater than the second, they are swapped. The process repeats until no swaps are required, which means the list is sorted. A boolean flag "swapped" is used to optimize the algorithm by exiting early if the array becomes sorted before completing all the passes.

Best Practices

- **Always implement the swap flag** (swapped) to detect if the array is already sorted. This allows the algorithm to terminate early, improving best-case performance to O(n).
- **Use Bubble Sort only on small or nearly sorted datasets**, such as in educational contexts or environments with minimal performance constraints.
- **Add clear comments** and maintain readable loop structures when implementing Bubble Sort to help students understand the iterative process.
- **Prefer using Bubble Sort where memory write operations are inexpensive**, as it involves frequent element swapping.
- **Use it as a learning tool** for teaching algorithmic concepts like iteration, comparison-based sorting, and optimization through early termination.

Common Mistakes

- **Failing to use the "swapped" condition**, which results in unnecessary passes over an already sorted array, wasting time.
- **Incorrect loop boundaries,** such as going out of bounds or missing edge cases—can lead to *ArrayIndexOutOfBoundsException* or missed comparisons.
- **Overusing Bubble Sort in practical systems**, despite its known inefficiency, especially for large or randomly ordered datasets.
- **Not understanding its simplicity can be a drawback**, as it may give learners a false sense of efficiency if used beyond its intended scope.
- **Assuming stability is always preserved**, but poor implementation may lead to bugs that break this property if not carefully handled.

Usage Limitations

- **Inefficient for large-scale datasets** due to its average and worst-case time complexity of $O(n^2)$, making it unsuitable for performance-critical applications.
- **Involves a high number of swap operations**, leading to increased memory write overhead, which is problematic in low-end or resource-constrained systems.
- **Not suited for modern real-time applications** or systems that require predictable or optimized performance (e.g., operating systems, large databases).
- **Rarely used in production code,** as more efficient sorting algorithms (like Merge Sort, Quick Sort, or Timsort) are available and widely implemented in standard libraries.
- **Limited adaptability to parallelization**, making it a poor choice in multi-threaded or GPU-accelerated environments.

Real-World Applications

Sorting patient files in a small clinic
In a rural clinic with 10-15 patient records for the day, a nurse needs to arrange them by appointment time. They compare adjacent files and swap them if the earlier appointment is listed after a later one, repeating until the list is ordered. Bubble sort is practical here because the dataset is small, and the clinic may lack digital tools, making manual sorting with a simple algorithm ideal.

Organizing a small batch of invoices in a freelance business
A freelancer with 8-12 invoices from the past month needs to sort them by payment amount for tax reporting. They compare adjacent invoices and swap them if a higher amount precedes a lower one, iterating until sorted. Bubble sort works well due to the limited number of invoices and the ease of implementing this process manually or in a basic spreadsheet.

Prioritizing tasks in a small team's daily stand-up
During a meeting, a team of 5-10 members lists their tasks on a whiteboard by estimated completion time. They compare adjacent tasks and swap them if a shorter task is listed after a longer one, repeating until the list is sorted. Bubble sort is effective here because the small task list and collaborative, visual nature of the process favor a simple, transparent sorting method.

Sorting sensor data in a low-power IoT device
A basic environmental monitor (e.g., a temperature logger) collects 10-20 readings hourly and needs to display them in ascending order. The device, with limited memory and processing power, uses bubble sort to compare and swap adjacent readings. Bubble sort is suitable because its low memory footprint and simplicity align with the constraints of embedded systems.

Arranging student presentations in a classroom activity
In a class of 10 students, each prepares a short presentation, and the teacher wants to order them by duration (shortest to longest). The teacher compares adjacent students' estimated times and swaps their slots if needed, repeating until the schedule is sorted. Bubble sort is appropriate due to the small group size and the straightforward, manual process that doesn't require complex software.

Practice Tasks

1. Sort a small shop's 6 product prices (in dollars) in ascending order using bubble sort.
2. Arrange a list of 8 daily task duration (in minutes) for a team in descending order using bubble sort.
3. Order a classroom's 10 student quiz scores (out of 100) from lowest to highest using bubble sort.
4. Sort a list of 7 temperature readings (in Celsius) from a low-power IoT sensor in ascending order using bubble sort.
5. Arrange a small event's 9 guest arrival times (in minutes past noon) from earliest to latest using bubble sort.

Sample Questions

1. When and why would you choose Bubble Sort in a real-world application, even though it's inefficient?
2. What makes Bubble Sort a stable sorting algorithm, and why is stability important in some applications?
3. How does Bubble Sort's performance compare to Insertion Sort and Selection Sort, and in what situations is it better or worse?
4. How can you modify Bubble Sort to sort in descending order, and what specific line of code would change?
5. How does the "swapped" flag improve Bubble Sort's efficiency, and in what case does it reduce time complexity?

Use-Case Challenges

A rural clinic handles 10–15 patient appointments daily. Due to limited digital infrastructure, appointment times are manually recorded and need to be organized before consultations begin.

Objective:
Implement Bubble Sort to arrange appointment times (in minutes past 9:00 AM) in ascending order to establish a proper consultation schedule.

Sample Input:

[5, 10, 15, 20, 25, 30, 35, 45, 50, 60]

Output

[45, 5, 30, 15, 60, 10, 20, 50, 25, 35]

Summary

Bubble Sort is one of the simplest sorting algorithms. Although it lacks efficiency on large datasets, it plays an important educational role. It helps students and beginners understand the basics of sorting, array manipulation, and iterative logic. Its simplicity makes it ideal for explaining fundamental algorithmic processes and stepping stones toward more advanced sorting techniques.

4.1 Bubble Sort

Practice Tasks – Answers

1. Sort a small shop's 6 product prices (in dollars) in ascending order using Bubble Sort

```java
int[] prices = {25, 10, 30, 15, 20, 5};

for (int i = 0; i < prices.length - 1; i++) {
    for (int j = 0; j < prices.length - i - 1; j++) {
        if (prices[j] > prices[j + 1]) {  // ascending order
            int temp = prices[j];
            prices[j] = prices[j + 1];
            prices[j + 1] = temp;
        }
    }
}

System.out.println(java.util.Arrays.toString(prices));
```

2. Arrange a list of 8 daily task durations (in minutes) for a team in descending order using Bubble Sort

```java
int[] durations = {20, 35, 10, 25, 30, 15, 40, 18};

for (int i = 0; i < durations.length - 1; i++) {
    for (int j = 0; j < durations.length - i - 1; j++) {
        if (durations[j] < durations[j + 1]) {  // descending order
            int temp = durations[j];
            durations[j] = durations[j + 1];
            durations[j + 1] = temp;
        }
    }
}

System.out.println(java.util.Arrays.toString(durations));
```

3. Order a classroom's 10 student quiz scores (out of 100) from lowest to highest

```java
int[] quizScores = {78, 65, 89, 92, 55, 74, 83, 68, 90, 60};

for (int i = 0; i < quizScores.length - 1; i++) {
    for (int j = 0; j < quizScores.length - i - 1; j++) {
        if (quizScores[j] > quizScores[j + 1]) {  // ascending order check
            int temp = quizScores[j];
            quizScores[j] = quizScores[j + 1];
            quizScores[j + 1] = temp;
        }
    }
}

System.out.println(java.util.Arrays.toString(quizScores));
```

4. Sort a list of 7 temperature readings (in Celsius) from a low-power IoT sensor in ascending order using bubble sort.

```java
int[] temps = {22, 19, 25, 18, 21, 20, 23};

for (int i = 0; i < temps.length - 1; i++) {
    for (int j = 0; j < temps.length - i - 1; j++) {
        if (temps[j] > temps[j + 1]) {
            int temp = temps[j];
            temps[j] = temps[j + 1];
            temps[j + 1] = temp;
        }
    }
}

System.out.println(java.util.Arrays.toString(temps));
```

5. How does the "swapped" flag improve Bubble Sort's efficiency, and in what case does it reduce time complexity?

```java
int[] arrivalTimes = {25, 5, 30, 10, 15, 35, 20, 40, 45};

for (int i = 0; i < arrivalTimes.length - 1; i++) {
    for (int j = 0; j < arrivalTimes.length - i - 1; j++) {
        if (arrivalTimes[j] > arrivalTimes[j + 1]) {
            int temp = arrivalTimes[j];
            arrivalTimes[j] = arrivalTimes[j + 1];
            arrivalTimes[j + 1] = temp;
        }
    }
}

System.out.println(java.util.Arrays.toString(arrivalTimes));
```

Sample Questions – Answers

1. When and why would you choose Bubble Sort in a real-world application, even though it's inefficient?

   > Bubble Sort is suitable in scenarios where the dataset is small, system memory is limited, or where algorithm simplicity is preferred—such as embedded systems, classroom exercises, or manually sorted data lists.

2. What makes Bubble Sort a stable sorting algorithm, and why is stability important in some applications?

   > Bubble Sort is stable because it preserves the relative order of equal elements. This is important in multi-key sorting (e.g., sorting students by score while preserving name order).

3. How does Bubble Sort's performance compare to Insertion Sort and Selection Sort, and in what situations is it better or worse?

   > Bubble Sort generally performs more swaps than Insertion Sort, making it slower in practice. Insertion Sort is better for nearly sorted lists. Bubble Sort is simpler but not suitable for large or random datasets.

4. How can you modify Bubble Sort to sort in descending order, and what specific line of code would change?

   > Change the comparison from if (arr[j] > arr[j + 1]) to if (arr[j] < arr[j + 1]).

5. How does the "swapped" flag improve Bubble Sort's efficiency, and in what case does it reduce time complexity?

   > The swapped flag checks whether any swap occurred in a pass. If no swaps happen, the array is already sorted, allowing early termination and improving best-case performance from $O(n^2)$ to O(n).

Use-Case Challenge - Answer

```java
import java.util.Arrays;

public class ClinicAppointmentSorter {

    public static int[] bubbleSort(int[] arr) {
        int n = arr.length;
        boolean swapped;

        for (int i = 0; i < n - 1; i++) {
            swapped = false;
            for (int j = 0; j < n - i - 1; j++) {
                if (arr[j] > arr[j + 1]) {
                    // Swap elements
                    int temp = arr[j];
                    arr[j] = arr[j + 1];
                    arr[j + 1] = temp;
                    swapped = true;
                }
            }
            if (!swapped) break; // Exit early if no swaps
        }

        return arr; // Return the sorted array
    }

    public static void main(String[] args) {
        int[] times = {45, 5, 30, 15, 60, 10, 20, 50, 25, 35};

        int[] sortedTimes = bubbleSort(times); // Get sorted result

        System.out.println("Sorted Appointment Times: " + Arrays.toString(sortedTimes));
    }
}
```

```
Sorted Appointment Times: [5, 10, 15, 20, 25, 30, 35, 45, 50, 60]
```

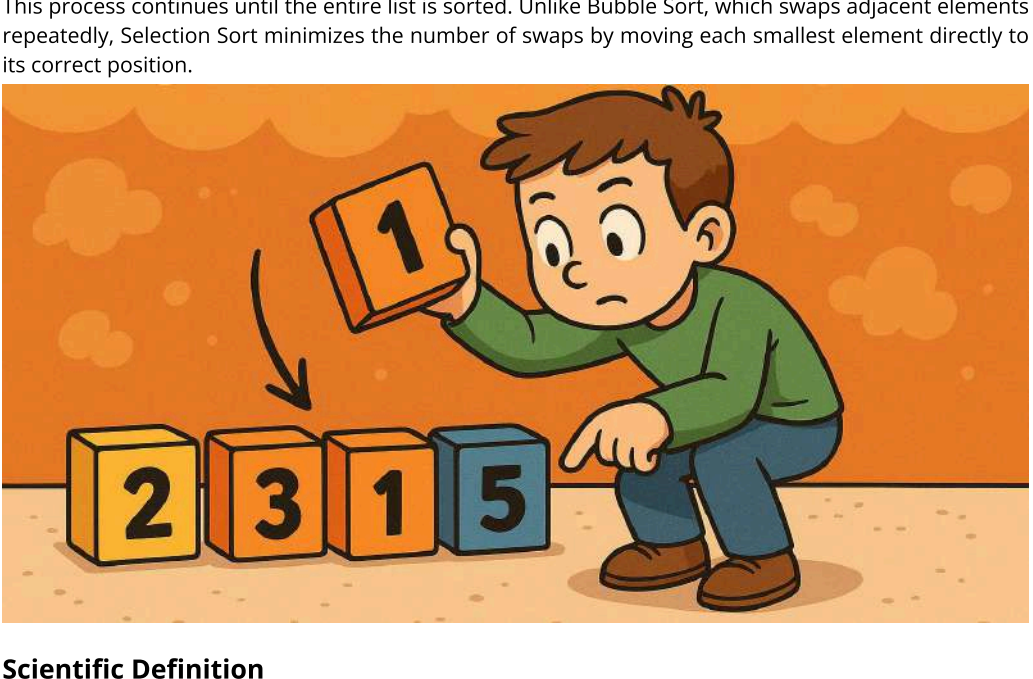
7.2 Selection Sort

Selection Sort is another fundamental and intuitive sorting algorithm introduced early in the study of computer science. Like Bubble Sort, it is not suitable for large datasets due to its quadratic time complexity, but it plays an important role in teaching the core concepts of sorting. It helps learners grasp essential ideas such as element comparison, selection, and in-place data manipulation.

The name Selection Sort is derived from the algorithm's key operation: during each iteration, it selects the smallest (or largest, depending on sorting order) element from the unsorted portion of the list and places it in its correct position within the sorted portion.

The algorithm functions by dividing the list into two parts: the sorted section at the beginning, and the unsorted section that occupies the remainder. It repeatedly searches for the smallest element in the unsorted section and swaps it with the first unsorted element, effectively growing the sorted section by one element with each pass.

This process continues until the entire list is sorted. Unlike Bubble Sort, which swaps adjacent elements repeatedly, Selection Sort minimizes the number of swaps by moving each smallest element directly to its correct position.



Scientific Definition

Selection Sort is an **in-place, comparison-based sorting algorithm**. It has a **worst-case, average-case, and best-case time complexity of O(n²)**, where n is the number of elements in the input array. Unlike stable sorting algorithms, **Selection Sort is not stable** in its typical implementation because it may change the relative order of equal elements. The algorithm repeatedly selects the minimum (or maximum) element from the unsorted portion of the array and swaps it with the first unsorted element, gradually growing the sorted portion of the array from the front. It performs **O(n²)** comparisons and **O(n)** swaps in total, making it inefficient on large datasets compared to more advanced algorithms like Merge Sort or Quick Sort.

One Sentence Description

Selection Sort is a simple sorting algorithm that repeatedly selects the minimum element from the unsorted portion and swaps it with the first unsorted element.

How It Works (Detailed Explanation)

Assume we are sorting an array of integers in ascending order.

1. First Pass:
   - Start from the first index.
   - Compare the first and second elements.
   - If the first is greater, swap them.
   - Move to the next pair and repeat until the end of the array.
   - After this pass, the largest element is at the end.

2. Subsequent Passes:
   - Repeat the process for the remaining elements (excluding the already sorted last elements).
   - With each pass, the next largest element is moved to its correct position.

Note on Optimization:
- If during a complete pass no swaps are made, the array is already sorted, and the algorithm terminates early.
- This optimization improves the best-case performance to O(n).

Visual Illustration

Let's take an example array: [5, 3, 8, 4, 2]

Pass 1
Find the minimum from index 0 to 4 → min = 2 at index 4
Swap 2 with element at index 0 → [2, 3, 8, 4, 5]

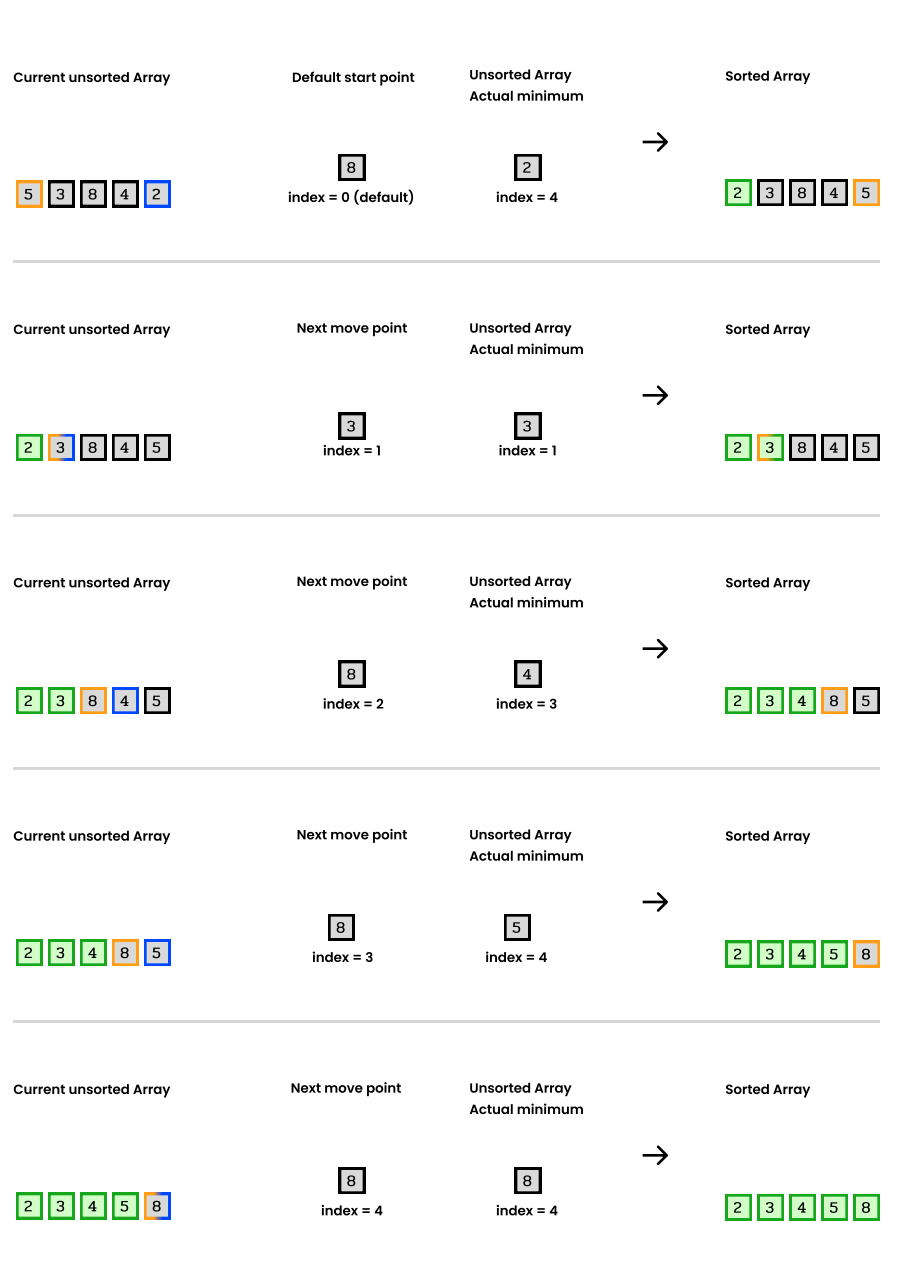
Pass 2
Find the minimum from index 1 to 4 → min = 3 at index 1
No swap needed → [2, 3, 8, 4, 5]

Pass 3
Find the minimum from index 2 to 4 → min = 4 at index 3
Swap 4 with element at index 2 → [2, 3, 4, 8, 5]

Pass 4
Find the minimum from index 3 to 4 → min = 5 at index 4
Swap 5 with element at index 3 → [2, 3, 4, 5, 8]

Pass 5
Only one element left at index 4 → already sorted

Final Result: [2, 3, 4, 5, 8]



Java Implementation

```java
import java.util.Arrays;

public class SelectionSort {

    static int[] sort(int[] arr) {
        int n = arr.length;
        // Copy the original array to avoid in-place modification
        int[] result = Arrays.copyOf(arr, n);

        for (int i = 0; i < n - 1; i++) {
            // Assume the minimum is at the current position
            int minIndex = i;

            // Find the index of the minimum element in the unsorted portion
            for (int j = i + 1; j < n; j++) {
                if (result[j] < result[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap the found minimum element with the first unsorted element
            if (minIndex != i) {
                int temp = result[i];
                result[i] = result[minIndex];
                result[minIndex] = temp;
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[] arr = {5, 3, 8, 4, 2};
        System.out.println("Before: " + Arrays.toString(arr));
        int[] sortedArr = sort(arr);
        System.out.println("After: " + Arrays.toString(sortedArr));
    }
}
```

Output

```
Before: [5, 3, 8, 4, 2]
After:  [2, 3, 4, 5, 8]
```

Explanation

The program begins by **copying the input array** to avoid modifying the original. It calculates the length **n** of the array. The outer loop runs **n-1 times**, representing passes over the unsorted portion of the array. For each pass, the algorithm assumes the first unsorted element is the minimum **(minIndex)**. Then, the inner loop scans the remaining unsorted elements to **find the actual minimum value's index**. After completing the inner loop, if the minimum element is not already in correct position, it **swaps** that minimum element with the first unsorted element. This process gradually **grows the sorted portion of the array from the front** until the entire array is sorted. Unlike Bubble Sort,

Selection Sort does **not use a swapped flag** and always performs all passes, meaning it doesn't terminate early even if the array is already sorted. The time complexity remains O(n²) in all cases.

Best Practices

- **Copy the array if you want to preserve the original data,** because Selection Sort modifies the array in place. This is crucial when the original dataset must remain unchanged.
- **Minimize swaps by only swapping when necessary,** i.e., when the minimum element is not already in the correct position. This helps reduce memory write operations, which can be important in systems with limited write cycles (like flash memory).
- **Use Selection Sort primarily on small datasets** (typically fewer than a few hundred elements). Its simplicity is ideal for such cases where algorithmic overhead should be minimal.
- **Understand that Selection Sort is stable only if implemented with care** — typically it is not stable by default since swapping might reorder equal elements.
- **Prefer other algorithms for larger datasets,** since Selection Sort's O(n²) time complexity leads to poor scalability.

Common Mistakes

- **Failing to update the minimum index during the inner loop** leads to incorrect minimum selection and wrong sorting results.
- **Incorrect loop boundaries** can cause index out of bounds errors or skipped elements, e.g., starting the inner loop at the wrong index.
- Performing unnecessary swaps, such as swapping an element with itself when the minimum is at the current position.
- **Assuming Selection Sort is stable** without modification — it typically is not stable because it swaps elements indiscriminately.
- **Not considering input validation or empty arrays**, which can cause exceptions if not handled.

Usage Limitations

- **Always runs in O(n²) time complexity** for comparisons, regardless of the input's initial state — so it doesn't improve on nearly sorted data.
- **Number of swaps is O(n), fewer than Bubble Sort or Insertion Sort**, but the total number of comparisons is still high, making it inefficient on large datasets.
- **Not suitable for real-time systems or large-scale data processing** where performance is critical.
- **Less cache-friendly** compared to algorithms like Quick Sort or Merge Sort, which use divide-and-conquer and better memory locality.
- **Not stable by default**, which may be a problem when the relative order of equal elements must be preserved.

Real-World Applications

Organizing Your Study Schedule
Imagine you have a list of assignments with different deadlines, and you want to sort them from earliest to latest to plan your study time efficiently. Selection Sort can help you pick the closest deadline step-by-step and put it in order.

Sorting Your Music Playlist by Song Length
If you want to arrange songs from shortest to longest for a quick music session, you can use Selection Sort to find the shortest song remaining and add it to your playlist order one by one.

Arranging Books on Your Shelf by Height
You want your books to go from shortest to tallest. Selection Sort helps by picking the shortest book left and placing it at the correct spot on the shelf, repeating until all books are arranged nicely.

Ranking Your Test Scores
After exams, you want to list your scores in ascending or descending order to see where you stand. Selection Sort lets you find the lowest or highest score left and place it in the list until all scores are sorted.

Sorting Your Friends' Names Alphabetically
Say you have a small list of friends' names and want to arrange them alphabetically for a group chat or event invitation. Selection Sort helps by selecting the alphabetically smallest name each time and ordering the list.

Practice Tasks

1. Sort a list of 8 product prices in ascending order using Selection Sort.
2. Arrange 12 employee ID numbers from lowest to highest.
3. Sort 10 task durations in descending order for a small team project.
4. Organize a classroom's 15 quiz scores in ascending order.
5. Sort temperature readings from a sensor device with limited computational power.

Sample Questions

1. How does Selection Sort minimize the number of swaps compared to Bubble Sort? Why is this important in some applications?
2. Why does Selection Sort perform the same number of comparisons regardless of whether the input array is sorted or not?
3. What changes would you make to Selection Sort to sort in descending order?
4. In what cases is Selection Sort preferred over more efficient algorithms like Quick Sort or Merge Sort?
5. How can Selection Sort be modified to make it a stable sorting algorithm?

Use-Case Challenges

A student living alone does weekly grocery shopping and logs expiry dates of food items (measured in days until expiration). To minimize food waste, they want to arrange their groceries so items that expire sooner are used first.

The student notes the days until each item expires and decides to sort them in ascending order using Selection Sort. This way, the earliest expiring items come first on their "Use Me First" shelf.

Objective:
Sort a list of grocery items by their days until expiration using the Selection Sort algorithm so that the items that expire soonest appear first. This helps the student reduce food waste by consuming perishables in the right order.

Sample Input	Output
[12, 5, 3, 14, 2, 9, 1]	[1, 2, 3, 5, 9, 12, 14]

Summary

Selection Sort is a straightforward comparison-based sorting algorithm best suited for small datasets or scenarios where swap minimization is critical. It works by repeatedly selecting the smallest unsorted element and swapping it into its correct position, growing a sorted section from the front of the array. While simple to implement and understand, its O(n²) time complexity makes it inefficient on large inputs. Selection Sort offers consistent performance regardless of the initial order of data but does not benefit from partially sorted input. Its predictable behavior and simplicity make it an excellent educational tool and a practical choice for specific use cases involving small or memory-constrained datasets.

4.2 Selection Sort

Practice Tasks – Answers

1. Sort a list of 8 product prices in ascending order using Selection Sort.

```java
int[] prices = {25, 10, 30, 15, 20, 5, 12, 18};

for (int i = 0; i < prices.length - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < prices.length; j++) {
        if (prices[j] < prices[minIndex]) {
            minIndex = j;
        }
    }
    int temp = prices[i];
    prices[i] = prices[minIndex];
    prices[minIndex] = temp;
}

System.out.println(java.util.Arrays.toString(prices));
```

2. Arrange 12 employee ID numbers from lowest to highest.

```java
int[] employeeIDs = {1023, 1001, 1045, 1012, 1034, 1007,
                     1050, 1029, 1015, 1038, 1003, 1040};

for (int i = 0; i < employeeIDs.length - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < employeeIDs.length; j++) {
        if (employeeIDs[j] < employeeIDs[minIndex]) {
            minIndex = j;
        }
    }
    int temp = employeeIDs[i];
    employeeIDs[i] = employeeIDs[minIndex];
    employeeIDs[minIndex] = temp;
}

System.out.println(java.util.Arrays.toString(employeeIDs));
```

3. Sort 10 task durations in descending order for a small team project.

```java
int[] durations = {45, 20, 35, 10, 50, 30, 25, 60, 15, 40};

for (int i = 0; i < durations.length - 1; i++) {
    int maxIndex = i;
    for (int j = i + 1; j < durations.length; j++) {
        if (durations[j] > durations[maxIndex]) {
            maxIndex = j;
        }
    }
    int temp = durations[i];
    durations[i] = durations[maxIndex];
    durations[maxIndex] = temp;
}

System.out.println(java.util.Arrays.toString(durations));
```

4. Organize a classroom's 15 quiz scores in ascending order.

```java
int[] quizScores = {72, 85, 90, 66, 58, 93, 77, 81, 69, 88, 74, 61, 80, 95, 70};

for (int i = 0; i < quizScores.length - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < quizScores.length; j++) {
        if (quizScores[j] < quizScores[minIndex]) {
            minIndex = j;
        }
    }
    int temp = quizScores[i];
    quizScores[i] = quizScores[minIndex];
    quizScores[minIndex] = temp;
}

System.out.println(java.util.Arrays.toString(quizScores));
```

5. Sort temperature readings from a sensor device with limited computational power.

```java
int[] temperatures = {32, 28, 35, 30, 26, 29, 31, 27, 33, 25};

for (int i = 0; i < temperatures.length - 1; i++) {
    int minIndex = i;
    for (int j = i + 1; j < temperatures.length; j++) {
        if (temperatures[j] < temperatures[minIndex]) {
            minIndex = j;
        }
    }
    int temp = temperatures[i];
    temperatures[i] = temperatures[minIndex];
    temperatures[minIndex] = temp;
}

System.out.println(java.util.Arrays.toString(temperatures));
```

Sample Questions – Answers

1. How does Selection Sort minimize the number of swaps compared to Bubble Sort? Why is this important in some applications?

> Selection Sort minimizes swaps by performing only one swap per iteration of the outer loop, placing the minimum element in its correct position. Bubble Sort, on the other hand, may perform multiple swaps in a single pass. This is important in applications where write operations are costly or limited, such as in embedded systems or flash memory, because fewer swaps reduce wear and resource usage.

2. Why does Selection Sort perform the same number of comparisons regardless of whether the input array is sorted or not?

> Selection Sort always compares each element in the unsorted portion of the array to find the minimum, regardless of the current order. It does not check whether the array is already sorted, so it performs a fixed number of comparisons determined by the array size. This means the algorithm lacks early termination and behaves the same in best, average, and worst cases in terms of comparisons.

3. What changes would you make to Selection Sort to sort in descending order?

> To sort in descending order, you would change the comparison in the inner loop to find the maximum element instead of the minimum. Specifically, replace the condition if (arr[j] < arr[minIndex]) with if (arr[j] > arr[maxIndex]) and update the variable names accordingly. This will move the largest elements to the beginning of the array.

4. In what cases is Selection Sort preferred over more efficient algorithms like Quick Sort or Merge Sort?

> Selection Sort is preferred when working with small datasets, systems with tight memory constraints, or when minimizing the number of swaps is more critical than execution speed. It is also useful in educational settings to teach sorting fundamentals due to its simple and intuitive logic. In such cases, the simplicity of Selection Sort outweighs the performance benefits of more complex algorithms.

5. How can Selection Sort be modified to make it a stable sorting algorithm?

> To make Selection Sort stable, avoid direct swapping. Instead, when the minimum element is found, shift all elements between its position and the current position to the right by one, and then insert the minimum element into its correct place. This preserves the relative order of equal elements, ensuring the algorithm's stability.

Use-Case Challenge - Answer

```java
import java.util.Arrays;

public class GroceryExpirySort {

    // Selection Sort to sort expiration days in ascending order
    static int[] selectionSort(int[] expiryDays) {
        int n = expiryDays.length;

        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;

            // Find index of the minimum expiry day in unsorted portion
            for (int j = i + 1; j < n; j++) {
                if (expiryDays[j] < expiryDays[minIndex]) {
                    minIndex = j;
                }
            }

            // Swap if minIndex is different from current position i
            if (minIndex != i) {
                int temp = expiryDays[i];
                expiryDays[i] = expiryDays[minIndex];
                expiryDays[minIndex] = temp;
            }
        }

        return expiryDays;
    }

    public static void main(String[] args) {
        // Sample input: days until expiration for grocery items
        int[] expiryDays = {7, 2, 14, 1, 10, 5, 3};

        System.out.println("Before sorting: " + Arrays.toString(expiryDays));

        int[] sortedExpiry = selectionSort(expiryDays);

        System.out.println(Arrays.toString(sortedExpiry));
    }
}
```

```
Before sorting: [7, 2, 14, 1, 10, 5, 3]
After sorting (soonest expiration first): [1, 2, 3, 5, 7, 10, 14]
```

7.3 Insertion Sort

Insertion Sort is a straightforward sorting algorithm that builds the final sorted list one element at a time. It is often used as an introductory example because of its simplicity and ease of understanding.
The algorithm works by dividing the list into two parts: a **sorted** section at the beginning, and an **unsorted** section comprising the remaining elements. Starting from the second element, Insertion Sort takes each element from the unsorted section and inserts it into the correct position within the sorted section.

To insert an element, the algorithm compares it with elements in the sorted section, shifting those that are larger one position to the right to create space. Once the correct spot is found, the element is placed in that position. This process repeats until all elements are sorted.

Unlike Selection Sort, which searches for the smallest element and swaps it, Insertion Sort inserts elements directly where they belong, often making fewer movements when the list is already partially sorted.
Insertion Sort is efficient for small or nearly sorted lists, but its performance decreases with larger, randomly ordered datasets due to its quadratic time complexity.

Scientific Definition

Insertion Sort is an in-place, comparison-based sorting algorithm. It has a worst-case and average-case time complexity of $O(n^2)$, where n is the number of elements in the input array, and a best-case time complexity of $O(n)$ when the array is already sorted. Unlike some sorting algorithms, Insertion Sort is stable because it does not change the relative order of equal elements. The algorithm builds the sorted portion of the array by repeatedly taking the next unsorted element and inserting it into its correct position within the sorted portion, shifting larger elements one position to the right as needed. It performs $O(n^2)$ comparisons and $O(n^2)$ movements in the worst case, making it inefficient on large datasets but efficient for small or nearly sorted datasets.

One sentence description

Insertion Sort is a sorting algorithm that iteratively takes each element from the unsorted portion and inserts it into the correct position within the sorted portion of the list.

How It Works (Detailed Explanation)

Assume we are sorting an array of integers in ascending order.

1. First Pass:
   - Start from the second element (index 1).
   - Compare this element with the one before it.
   - If the current element is smaller, shift the larger element one position to the right.
   - Continue comparing and shifting elements in the sorted portion until the correct position for the current element is found.
   - Insert the current element into this position.
2. Subsequent Passes:
   - Move to the next unsorted element and repeat the process:
   - Compare it with elements in the sorted portion to its left, shifting larger elements right.
   - Insert the element into its correct position.
   - With each pass, the sorted portion grows by one element.

Note on Optimization:
- Since Insertion Sort only shifts elements when necessary, it performs very efficiently on nearly sorted arrays.
- If the array is already sorted, the algorithm only compares adjacent elements without shifting, resulting in a best-case time complexity of O(n)O(n)O(n).

Visual Illustration

Let's take an example array: **[5, 3, 8, 4, 2]**

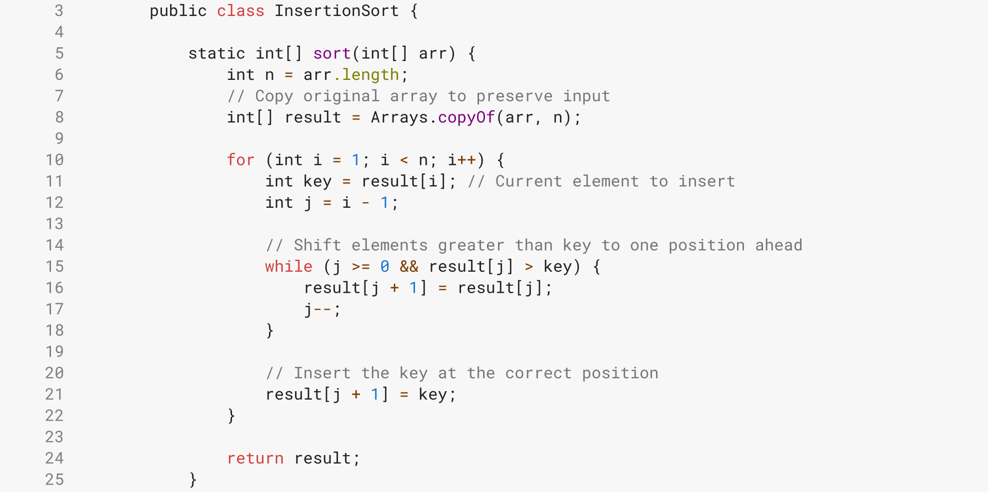
Pass 1
Take element at index 1 (3). Compare with 5 at index 0.
Since 3 < 5, shift 5 right and insert 3 at index 0.
Array becomes: [3, 5, 8, 4, 2]

Pass 2
Take element at index 2 (8). Compare with 5.
8 > 5, so no shifting needed. Insert 8 at index 2.
Array remains: [3, 5, 8, 4, 2]

Pass 3
Take element at index 3 (4). Compare with 8, shift 8 right.
Compare with 5, shift 5 right.
Compare with 3, 4 > 3, insert 4 at index 1.
Array becomes: [3, 4, 5, 8, 2]

Pass 4
Take element at index 4 (2). Compare with 8, shift 8 right.
Compare with 5, shift 5 right.
Compare with 4, shift 4 right.
Compare with 3, shift 3 right.
Insert 2 at index 0.
Array becomes: [2, 3, 4, 5, 8]

Final Result: [2, 3, 4, 5, 8]



Java Implementation

```java
import java.util.Arrays;

public class InsertionSort {

    static int[] sort(int[] arr) {
        int n = arr.length;
        // Copy original array to preserve input
        int[] result = Arrays.copyOf(arr, n);

        for (int i = 1; i < n; i++) {
            int key = result[i]; // Current element to insert
            int j = i - 1;

            // Shift elements greater than key to one position ahead
            while (j >= 0 && result[j] > key) {
                result[j + 1] = result[j];
                j--;
            }

            // Insert the key at the correct position
            result[j + 1] = key;
        }

        return result;
    }

    public static void main(String[] args) {
        int[] arr = {5, 3, 8, 4, 2};
        System.out.println("Before: " + Arrays.toString(arr));
        int[] sortedArr = sort(arr);
        System.out.println("After: " + Arrays.toString(sortedArr));
    }
}
```

Output

```
Before: [5, 3, 8, 4, 2]
After:  [2, 3, 4, 5, 8]
```

Explanation

The program begins by **copying the input array** to avoid modifying the original data. It calculates the length n of the array. The sorting process starts from the **second element (index 1)**, assuming the **first element is already "sorted."** For each iteration, the **current element (key)** is compared with the elements in the **sorted portion (from right to left).**

The algorithm **shifts all larger elements one position to the right** to make space for the key. Once it finds the correct position where the key is **greater than or equal** to the compared element (or the start of the array is reached), it **inserts the key** into that position.

This process repeats until **all elements are placed in their correct positions.** As a result, the **sorted portion of the array grows from left to right.** Insertion Sort is **adaptive**, meaning it performs **fewer operations if the array is already or nearly sorted**. It is also **stable**, preserving the **order of equal elements**. While its **worst-case and average-case time complexity is $O(n^2)$**, its **best-case performance is $O(n)$** when the array is already sorted.

Best Practices

- **Start from the second element** (index 1), assuming the first element is already sorted. This forms the basis of the algorithm's logic — building a sorted section one element at a time.
- **Use a temporary variable (often called key)** to store the current element before shifting other elements. This ensures data integrity while making room for insertion.
- **Shift larger elements to the right** instead of swapping. Insertion Sort relies on element shifting, which is more efficient and reflects how insertion logically works in real scenarios (e.g., sorting cards in hand).
- Use Insertion Sort on **small datasets or nearly sorted data**, where it performs very efficiently — even achieving **linear time $O(n)$** in the best case.
- **Avoid unnecessary comparisons by breaking early** from the inner loop once the correct position is found, improving performance.
- If needed, **enhance with Binary Search** to find the insertion position faster — a technique used in more advanced versions like **Binary Insertion Sort**.
- Although **stable by nature**, maintain stability by ensuring equal elements are **not moved past each other**, preserving their original order.

Common Mistakes

- **Not starting from the second element** (index 1), causing incorrect assumptions about the sorted portion and leading to errors.
- **Failing to store the current element in a temporary variable (key) before shifting**, which results in **overwriting values** and losing data.
- **Swapping elements instead of shifting**, which is less efficient and deviates from the correct insertion sort logic.
- **Incorrect loop boundaries or conditions** in the inner loop, causing **index out-of-bounds errors** or infinite loops.
- **Not breaking early** from the inner loop once the correct insertion point is found, leading to unnecessary comparisons and reduced performance.
- **Assuming Insertion Sort is unstable** — it is **stable by nature**, so moving equal elements past each other is a common mistake to avoid.
- **Ignoring edge cases**, such as empty arrays or arrays with one element, which should be handled gracefully.

Usage Limitations

- **Runs in $O(n^2)$ time complexity** in the worst and average cases; however, it performs in **$O(n)$** time on nearly sorted data, making it better than Selection Sort in some scenarios.
- The number of **element shifts can be high** for large or randomly ordered datasets, making it inefficient compared to advanced algorithms like **Quick Sort** or **Merge Sort**.
- **Not suitable for large-scale or real-time systems** where performance and speed are critical.
- While **stable and in-place**, it is **less cache-efficient** compared to divide-and-conquer algorithms that exploit better memory locality.
- Best suited for **small datasets or nearly sorted arrays**, where it can outperform more complex sorts due to low overhead.

Real-World Applications

Maintaining a Sorted Playlist
When adding new songs one at a time to a playlist sorted by duration or popularity, Insertion Sort efficiently places each new song in the right position without reordering the entire list.

Sorting Patient Arrival Times in Small Clinics
New patients arriving at irregular intervals can be inserted into an already sorted schedule quickly using Insertion Sort, ensuring the appointment list remains ordered by arrival time.

Organizing Library Books by Call Number
When new books arrive intermittently, librarians can insert each book into the correct spot in a sorted shelf list without needing to re-sort the entire collection.

Real-Time Event Logs in Embedded Systems
For small, continuously updating logs, such as error events or sensor readings, Insertion Sort can quickly insert new entries while maintaining chronological order.

Ranking Players' Scores in Casual Games
As players finish rounds, their scores are added and sorted instantly in a leaderboard that is mostly sorted already, allowing Insertion Sort to keep the rankings updated smoothly.

Practice Tasks

1. Sort a list of 10 student test scores from lowest to highest using Insertion Sort.
2. Arrange 8 daily task durations (in minutes) in descending order.
3. Insert new transaction amounts into an existing sorted list of bank deposits.
4. Organize a small dataset of 12 product weights in ascending order.
5. Sort sensor readings from an IoT device collected over several hours.

Sample Questions

1. Why does Insertion Sort perform better than Selection Sort on nearly sorted data?
2. How does the shifting process in Insertion Sort differ from swapping elements?
3. In what ways does Insertion Sort maintain stability during sorting?
4. How would you modify Insertion Sort to sort elements in descending order?
5. What are the advantages and disadvantages of using Insertion Sort on large datasets?

Use-Case Challenges

A small bookstore keeps track of the prices of its books on a shelf. New books arrive frequently, and the store manager wants to keep the price list sorted from cheapest to most expensive to help customers quickly find affordable options.

Since the price list is already mostly sorted, the manager chooses **Insertion Sort** to efficiently insert each new book price into the correct position without re-sorting the entire list.

Objective:
Use the Insertion Sort algorithm to maintain an ascending order of book prices as new books are added, minimizing sorting time and effort.

Sample Input:

```
[15, 22, 18, 25, 10]
```

Output:

```
[10, 15, 18, 22, 25, ]
```

Summary

Insertion Sort is a simple, comparison-based sorting algorithm that builds a sorted portion of the array one element at a time by inserting each new element into its correct position. It works efficiently on small or nearly sorted datasets, with an average and worst-case time complexity of $O(n^2)$. Unlike Selection Sort, Insertion Sort adapts well to partially sorted data, often performing fewer comparisons and shifts in such cases. Its stable nature preserves the relative order of equal elements, making it suitable for applications where stability is important. While not ideal for large datasets, Insertion Sort's straightforward implementation and good performance on small or nearly sorted inputs make it a valuable teaching tool and a practical choice in specific scenarios.

Answers

4.3 Insertion Sort

Practice Tasks – Answers

1. Sort a list of 10 student test scores from lowest to highest using Insertion Sort.

```java
int[] scores = {75, 62, 88, 90, 55, 70, 85, 60, 95, 68};

for (int i = 1; i < scores.length; i++) {
    int key = scores[i];
    int j = i - 1;
    while (j >= 0 && scores[j] > key) {
        scores[j + 1] = scores[j];
        j--;
    }
    scores[j + 1] = key;
}

System.out.println(java.util.Arrays.toString(scores));
```

2. Arrange 8 daily task durations (in minutes) in descending order.

```java
int[] durations = {20, 35, 10, 25, 30, 15, 40, 18};

for (int i = 1; i < durations.length; i++) {
    int key = durations[i];
    int j = i - 1;
    while (j >= 0 && durations[j] < key) {
        durations[j + 1] = durations[j];
        j--;
    }
    durations[j + 1] = key;
}

System.out.println(java.util.Arrays.toString(durations));
```

3. Insert new transaction amounts into an existing sorted list of bank deposits.

```java
int[] deposits = {100, 150, 200, 250, 300, 350, 400};
int newDeposit = 275;

// Expand array by 1
int[] updated = new int[deposits.length + 1];
System.arraycopy(deposits, 0, updated, 0, deposits.length);
updated[updated.length - 1] = newDeposit;

// Apply insertion sort to keep list sorted
for (int i = 1; i < updated.length; i++) {
    int key = updated[i];
    int j = i - 1;
    while (j >= 0 && updated[j] > key) {
        updated[j + 1] = updated[j];
        j--;
    }
    updated[j + 1] = key;
}

System.out.println(java.util.Arrays.toString(updated));
```

4. Organize a small dataset of 12 product weights in ascending order.

```java
int[] weights = {3, 7, 2, 9, 4, 6, 1, 5, 8, 10, 11, 0};

for (int i = 1; i < weights.length; i++) {
    int key = weights[i];
    int j = i - 1;
    while (j >= 0 && weights[j] > key) {
        weights[j + 1] = weights[j];
        j--;
    }
    weights[j + 1] = key;
}

System.out.println(java.util.Arrays.toString(weights));
```

5. Sort sensor readings from an IoT device collected over several hours.

```java
int[] readings = {22, 19, 25, 18, 21, 24, 20, 23, 26, 17};

for (int i = 1; i < readings.length; i++) {
    int key = readings[i];
    int j = i - 1;
    while (j >= 0 && readings[j] > key) {
        readings[j + 1] = readings[j];
        j--;
    }
    readings[j + 1] = key;
}

System.out.println(java.util.Arrays.toString(readings));
```

Sample Questions – Answers

1. Why does Insertion Sort perform better than Selection Sort on nearly sorted data?

> Insertion Sort performs fewer operations when data is nearly sorted because it only shifts elements as needed rather than making full comparisons across the remaining list. In the best case (fully sorted input), it runs in $O(n)$ time because the inner loop rarely executes.
> In contrast, Selection Sort always performs $O(n^2)$ comparisons regardless of input order, because it searches the entire unsorted portion for the minimum every time.

2. How does the shifting process in Insertion Sort differ from swapping elements?

> In Insertion Sort, elements are shifted one position to the right until the correct location for the current value is found. This avoids multiple swap operations.
> In contrast, swapping involves exchanging the positions of two elements directly, which often leads to more memory writes.
> Shifting is typically more efficient and causes less wear in memory-constrained or write-sensitive environments (e.g., flash storage).

3. In what ways does Insertion Sort maintain stability during sorting?

> Insertion Sort is stable by nature because it only moves elements that are strictly greater (or less in descending order), not equal.
> So, if two equal elements appear in the input, their relative order is preserved in the output — a key requirement in sorting datasets where secondary attributes (like timestamps or IDs) must be respected.

4. How can you insert a new element into an already sorted array using Insertion Sort logic?

> To insert a new element into an already sorted array using Insertion Sort logic:
>
> - Append the new element at the end of the array.
> - Shift larger elements (for ascending order) one position to the right until the correct spot is found.
> - Place the new element in its proper position.

```java
int key = newElement;
int j = sortedArray.length - 2;
while (j >= 0 && sortedArray[j] > key) {
    sortedArray[j + 1] = sortedArray[j];
    j--;
}
sortedArray[j + 1] = key;
```

This approach is efficient, especially when the array is already sorted, and minimizes unnecessary operations.

5. What are the advantages and disadvantages of using Insertion Sort on large datasets?

> Advantages:
> - Simple to implement and understand.
> - Performs well on small or nearly sorted datasets.
> - Stable and requires no additional memory (in-place).
> - Low overhead compared to more complex algorithms.
>
> Disadvantages:
> - Has $O(n^2)$ time complexity in the worst case.
> - Poor performance on large, unsorted datasets.
> - Not as efficient as algorithms like Merge Sort or Quick Sort for large-scale data.

Use-Case Challenge - Answer

```java
import java.util.Arrays;

public class BookPriceSort {

    // Insertion Sort to keep book prices sorted (ascending order)
    static int[] insertionSort(int[] prices) {
        int n = prices.length;

        for (int i = 1; i < n; i++) {
            int key = prices[i];
            int j = i - 1;

            // Shift elements greater than key to the right
            while (j >= 0 && prices[j] > key) {
                prices[j + 1] = prices[j];
                j--;
            }

            prices[j + 1] = key;
        }

        return prices;
    }

    public static void main(String[] args) {
        // Sample input: book prices in dollars
        int[] bookPrices = {15, 10, 25, 5, 20, 12, 30};

        System.out.println("Before sorting: " + Arrays.toString(bookPrices));

        int[] sortedPrices = insertionSort(bookPrices);

        System.out.println("After sorting : " + Arrays.toString(sortedPrices));
    }
}
```

```
Before sorting: [15, 10, 25, 5, 20, 12, 30]
After sorting : [5, 10, 12, 15, 20, 25, 30]
```

7.4 Merge Sort

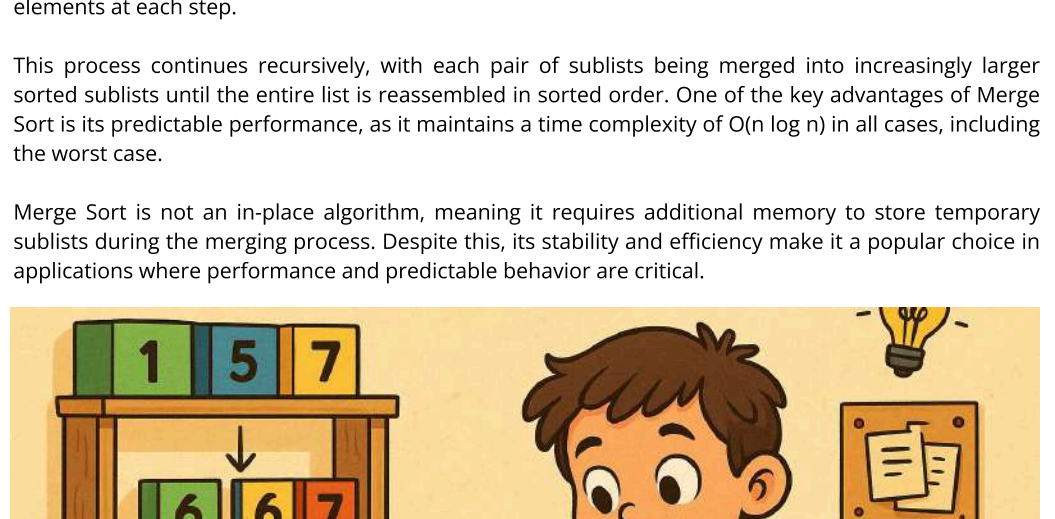
Merge Sort is a classic and highly efficient sorting algorithm that is frequently introduced in the study of algorithm design and analysis. Unlike simpler sorting techniques such as Bubble Sort or Selection Sort, Merge Sort is well suited for handling large datasets due to its favorable time complexity and consistent performance.

The name Merge Sort reflects the central operation of the algorithm: the merging of two sorted sublists into a single sorted list. It employs a divide-and-conquer approach, which involves breaking down a problem into smaller, more manageable subproblems and then combining their solutions to solve the original problem.

The algorithm begins by recursively dividing the list into halves until each sublist contains only a single element or is empty. Since a list of one element is inherently sorted, the algorithm then proceeds to merge these sublists in a manner that results in a sorted sequence. During the merge phase, it compares the smallest elements of the sublists and builds a new list by selecting the smaller of the two elements at each step.

This process continues recursively, with each pair of sublists being merged into increasingly larger sorted sublists until the entire list is reassembled in sorted order. One of the key advantages of Merge Sort is its predictable performance, as it maintains a time complexity of O(n log n) in all cases, including the worst case.

Merge Sort is not an in-place algorithm, meaning it requires additional memory to store temporary sublists during the merging process. Despite this, its stability and efficiency make it a popular choice in applications where performance and predictable behavior are critical.



Scientific Definition

Merge Sort is a **divide-and-conquer**, comparison-based sorting algorithm. It has a **worst-case, average-case, and best-case time complexity of O(n log n)**, where n is the number of elements in the input array. Unlike many other sorting algorithms, **Merge Sort is stable by design**, meaning it preserves the relative order of equal elements. The algorithm recursively divides the input array into two halves, sorts each half, and then **merges the sorted halves** to produce a fully sorted array. It requires **O(n) additional space** for the temporary arrays used during the merge process, making it **not an in-place algorithm**. Due to its predictable time complexity and stability, Merge Sort is well-suited for applications involving large datasets or linked lists.

One Sentence Description

Merge Sort is an efficient, divide-and-conquer algorithm that recursively splits the array into halves, sorts them, and then merges the sorted halves back together.

How It Works (Detailed Explanation)

Merge Sort is a **Divide and Conquer** algorithm.
It **divides** the array into two halves, **recursively sorts** each half, and then **merges** the sorted halves into a single sorted array.

Step-by-Step Process

Divide:

- Split the array into two halves.
- Continue dividing each half recursively until each subarray contains only one element.
- An array with one element is already sorted.

Conquer (Recursive Sort):

- Once the base case is reached (subarrays of one element), begin the merging process.
- **Sort** and **merge** the divided subarrays by comparing elements from both halves.

Combine (Merge):

- Use two pointers to compare elements from the left and right subarrays.
- **Select the smaller element** and move it to a temporary array.
- Continue until all elements from both subarrays are placed into the temporary array in sorted order.
- Copy the sorted temporary array back to the original array.

Visual Illustration

Let's take an example array:
[7, 2, 6, 3, 8, 5, 1, 4]

Step 1: Divide
Divide into: [7, 2, 6, 3] and [8, 5, 1, 4]

Further divide [7, 2, 6, 3] → [7, 2] and [6, 3]
Further divide [7, 2] → [7] and [2]
Further divide [6, 3] → [6] and [3]

Further divide [8, 5, 1, 4] → [8, 5] and [1, 4]
Further divide [8, 5] → [8] and [5]
Further divide [1, 4] → [1] and [4]

At this point, we have subarrays:
[7], [2], [6], [3], [8], [5], [1], [4]

Step 2: Conquer and Merge

Merge [7] and [2] → [2, 7]
Merge [6] and [3] → [3, 6]
Merge [2, 7] and [3, 6] →
Compare 2 with 3 → [2]
Compare 7 with 3 → [2, 3]
Compare 7 with 6 → [2, 3, 6]

Remaining: 7 → [2, 3, 6, 7]

Merge [8] and [5] → [5, 8]
Merge [1] and [4] → [1, 4]
Merge [5, 8] and [1, 4] →
Compare 5 with 1 → [1]
Compare 5 with 4 → [1, 4]

Remaining: 5, 8 → [1, 4, 5, 8]

Final Merge: [2, 3, 6, 7] and [1, 4, 5, 8]
Compare 2 with 1 → [1]
Compare 2 with 4 → [1, 2]
Compare 3 with 4 → [1, 2, 3]
Compare 6 with 4 → [1, 2, 3, 4]
Compare 6 with 5 → [1, 2, 3, 4, 5]
Compare 6 with 8 → [1, 2, 3, 4, 5, 6]
Compare 7 with 8 → [1, 2, 3, 4, 5, 6, 7]
Remaining: 8 → [1, 2, 3, 4, 5, 6, 7, 8]

Final Result:
[1, 2, 3, 4, 5, 6, 7, 8]

Java Implementation

```java
import java.util.Arrays;

public class MergeSort {

    static int[] sort(int[] arr) {
        // Copy the original array to avoid in-place modification
        int[] result = Arrays.copyOf(arr, arr.length);
        mergeSort(result, 0, result.length - 1);
        return result;
    }

    // Recursive function to divide the array
    private static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            // Find the middle point
            int mid = left + (right - left) / 2;

            // Recursively sort the first and second halves
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);

            // Merge the sorted halves
            merge(arr, left, mid, right);
        }
    }

    // Function to merge two sorted subarrays
    private static void merge(int[] arr, int left, int mid, int right) {
        // Sizes of two subarrays to merge
        int n1 = mid - left + 1;
        int n2 = right - mid;

        // Create temp arrays
        int[] leftArray = new int[n1];
        int[] rightArray = new int[n2];

        // Copy data to temp arrays
        for (int i = 0; i < n1; ++i)
            leftArray[i] = arr[left + i];
        for (int j = 0; j < n2; ++j)
            rightArray[j] = arr[mid + 1 + j];

        // Initial indexes of subarrays
        int i = 0, j = 0;
        int k = left;

        // Merge the temp arrays back into arr
        while (i < n1 && j < n2) {
            if (leftArray[i] <= rightArray[j]) {
                arr[k] = leftArray[i];
                i++;
            } else {
                arr[k] = rightArray[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements of leftArray
        while (i < n1) {
            arr[k] = leftArray[i];
            i++;
            k++;
        }

        // Copy remaining elements of rightArray
        while (j < n2) {
            arr[k] = rightArray[j];
            j++;
            k++;
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 3, 8, 4, 2};
        System.out.println("Before: " + Arrays.toString(arr));
        int[] sortedArr = sort(arr);
        System.out.println("After: " + Arrays.toString(sortedArr));
    }
}
```

Output

```
Before: [5, 3, 8, 4, 2]
After: [2, 3, 4, 5, 8]
```

Explanation

The program begins by copying the input array to avoid modifying the original. This ensures that the original data remains intact, and sorting is performed on a duplicate.

The sorting logic is handled by a **recursive function** called *mergeSort()*. It accepts three parameters: the array, the starting index (*left*), and the ending index (*right*). The array is **repeatedly divided in half** until each subarray contains only one element. At this point, each subarray is trivially considered sorted.

After division, the merge() function is called to **combine two sorted halves** into a single sorted segment. This merging process uses **two temporary arrays** to hold the elements of the left and right halves.

Within the **merge()** function:

- Two pointers (one for each temporary array) are used to compare elements.
- The smaller of the two compared elements is placed into the original array.
- This process continues until all elements from both temporary arrays are merged in sorted order.
- Any remaining elements (if one side finishes early) are then copied into the original array.

The **main sorting work is performed during the merge step**, not during the division. The recursion continues until the entire array has been broken down and merged back together in sorted order.

Finally, the main() method prints the array before and after sorting to demonstrate the result.

This implementation of Merge Sort is more efficient than simpler algorithms like Selection Sort or Bubble Sort, particularly for large datasets, due to its consistent **O(n log n)** time complexity.

Best Practices

- Use Merge Sort when **stability is required**, as it preserves the relative order of equal elements — crucial for applications like record sorting.
- Always ensure there is enough **auxiliary space**, since Merge Sort needs **O(n) extra memory**. Avoid it on memory-limited systems unless using an optimized variant.
- Rely on Merge Sort for consistent performance, as its **O(n log n)** time complexity holds even for sorted or reverse-sorted data.
- Prefer Merge Sort in **external sorting**, especially when dealing with data too large to fit in RAM. It works efficiently with **file-based datasets.**
- Improve performance by switching to **Insertion Sort** on small subarrays (typically fewer than 32 elements). This hybrid strategy is widely used in production-grade sorting libraries.
- Avoid using Merge Sort in **real-time or embedded systems** unless memory usage and latency are carefully managed.

Common Mistakes

- Forgetting that Merge Sort requires **additional memory** for merging. It is not an in-place sort by default, so failing to account for this can cause performance issues in memory-constrained environments.
- Not handling the **base case correctly** in recursion. Make sure the recursion stops when the array length is 1 or less, otherwise it may lead to infinite recursion or stack overflow.
- Overlooking the need to **copy subarrays properly** during the merge step. Incorrect slicing or referencing can lead to data corruption or unexpected results.
- Assuming Merge Sort is always faster. While it has a consistent **O(n log n)** time complexity, for small datasets, **simpler algorithms like Insertion Sort** may outperform it due to lower overhead.
- Ignoring stability in a custom implementation. If stability is required (i.e., preserving the order of equal elements), ensure your merge function does not rearrange equal items unnecessarily.
- Using Merge Sort blindly on linked lists without adapting it. The array-based implementation differs significantly from the more efficient version designed for **linked lists**, which avoids repeated array slicing.

Common Mistakes

- Forgetting that Merge Sort requires **additional memory** for merging. It is not an in-place sort by default, so failing to account for this can cause performance issues in memory-constrained environments.
- Not handling the **base case correctly** in recursion. Make sure the recursion stops when the array length is 1 or less, otherwise it may lead to infinite recursion or stack overflow.
- Overlooking the need to **copy subarrays properly** during the merge step. Incorrect slicing or referencing can lead to data corruption or unexpected results.
- Assuming Merge Sort is always faster. While it has a consistent **O(n log n)** time complexity, for small datasets, **simpler algorithms like Insertion Sort** may outperform it due to lower overhead.
- Ignoring stability in a custom implementation. If stability is required (i.e., preserving the order of equal elements), ensure your merge function does not rearrange equal items unnecessarily.
- Using Merge Sort blindly on linked lists without adapting it. The array-based implementation differs significantly from the more efficient version designed for **linked lists**, which avoids repeated array slicing.

Usage Limitations

- **Always runs in O(n log n) time complexity, regardless of the input's order, so it does not benefit from nearly sorted data.**
- **Requires additional space of O(n)**, which can be a disadvantage for large datasets or memory-constrained environments.
- **Not ideal for small datasets,** as the overhead from recursive calls and memory allocation can outweigh its performance benefits.
- **Recursive implementation can lead to stack overflow** on systems with limited stack size, especially if not optimized for tail recursion.
- **Less efficient than in-place algorithms like Quick Sort** in terms of memory usage and practical speed, particularly for simple, average-case scenarios.

Real-World Applications

Managing Large Contact Lists
Imagine you have thousands of names in your phonebook and want to arrange them alphabetically. Merge Sort can divide the list into smaller groups, sort them, and then combine them into one neatly ordered list.

Sorting Library Books by Title
A library system needs to organize thousands of book titles quickly. Merge Sort efficiently splits the titles into smaller chunks, sorts each one, and merges them into a single, sorted shelf list.

Organizing Online Store Products by Price
In an online store, users often want to view products from lowest to highest price. Merge Sort can handle large inventories by breaking the product list into smaller parts, sorting each part, and then merging them back together in the correct price order.

Combining Sorted Event Lists
Imagine two different departments created separate event calendars for the same month. Merge Sort can take both already-sorted calendars and merge them into a single, date-ordered list without losing any event.

Practice Tasks

1. Sort a list of 20 student names alphabetically using Merge Sort.
2. Arrange 16 shipment weights from lightest to heaviest.
3. Organize 24 book titles in a library database in alphabetical order.
4. Sort daily sales figures from the last 14 days in descending order.
5. Arrange 18 customer feedback scores to identify the top-rated experiences.

Sample Questions

1. What is the significance of dividing the array into halves in Merge Sort, and how does this impact its overall time complexity?
2. How does Merge Sort maintain stability, and why is this beneficial in certain applications like sorting records with multiple keys?
3. Why is Merge Sort considered more suitable than other algorithms for sorting linked lists or working with large datasets on disk?
4. What modifications would be required to implement Merge Sort iteratively instead of using recursion?
5. In what scenarios does Merge Sort's consistent performance (O(n log n)) provide an advantage over algorithms like Quick Sort that have a worst-case time of O(n²)?

Use-Case Challenges

A small online bookstore needs to organize its inventory of thousands of book titles alphabetically by author name. Since the inventory is large and frequently updated, they want a sorting method that efficiently handles big data without slowing down their system.

The bookstore decides to use Merge Sort because it reliably sorts large lists quickly by dividing the list into smaller parts, sorting those parts, and then merging them back together in order.

Objective:
Sort a large list of book titles by author name using the Merge Sort algorithm to ensure fast, stable, and efficient sorting. This helps the bookstore maintain an organized catalog for smooth searching and updating, even as the inventory grows.

Sample Input	Output
["Smith", "Adams", "Brown", "Clark", "Davis", "Evans", "Ford"]	["Adams", "Brown", "Clark", "Davis", "Evans", "Ford", "Smith"]

Summary

Merge Sort is a powerful divide-and-conquer sorting algorithm ideal for large datasets and scenarios requiring stable sorting. It works by recursively splitting the array into halves until single-element subarrays are reached, then merges these subarrays in a sorted manner to produce the final sorted array. Merge Sort guarantees a time complexity of **O(n log n)** in all cases, making it efficient and predictable regardless of the initial data order. Although it requires additional memory for merging, its stability and performance on large or partially sorted data make it a preferred choice in many practical applications, especially where consistent speed and reliability are important.

Merge Sort

Practice Tasks – Answers

1. Sort a list of 20 student names alphabetically using Merge Sort.

```java
public class MergeSortNames {

    // Merge two sorted subarrays into one sorted array
    public static void merge(String[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        String[] leftArr = new String[n1];
        String[] rightArr = new String[n2];

        for (int i = 0; i < n1; i++) {
            leftArr[i] = arr[left + i];
        }
        for (int j = 0; j < n2; j++) {
            rightArr[j] = arr[mid + 1 + j];
        }

        int i = 0, j = 0;
        int k = left;

        while (i < n1 && j < n2) {
            if (leftArr[i].compareTo(rightArr[j]) <= 0) {
                arr[k] = leftArr[i];
                i++;
            } else {
                arr[k] = rightArr[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = leftArr[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = rightArr[j];
            j++;
            k++;
        }
    }

    // Recursive Merge Sort function
    public static void mergeSort(String[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;

            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);

            merge(arr, left, mid, right);
        }
    }

    public static void main(String[] args) {
        String[] students = {
            "Olivia", "Liam", "Emma", "Noah", "Ava", "Oliver", "Sophia", "Elijah",
            "Isabella", "Lucas", "Mia", "Mason", "Amelia", "Logan", "Harper", "Ethan",
            "Evelyn", "James", "Abigail", "Benjamin"
        };

        mergeSort(students, 0, students.length - 1);

        System.out.println(java.util.Arrays.toString(students));
    }
}
```

2. Arrange 16 shipment weights from lightest to heaviest.

```java
public class ShipmentWeightsMergeSort {

    // Merge two sorted subarrays into one sorted array
    public static void merge(int[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int[] leftArr = new int[n1];
        int[] rightArr = new int[n2];

        for (int i = 0; i < n1; i++) {
            leftArr[i] = arr[left + i];
        }
        for (int j = 0; j < n2; j++) {
            rightArr[j] = arr[mid + 1 + j];
        }

        int i = 0, j = 0;
        int k = left;

        while (i < n1 && j < n2) {
            if (leftArr[i] <= rightArr[j]) {
                arr[k] = leftArr[i];
                i++;
            } else {
                arr[k] = rightArr[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements of leftArr, if any
        while (i < n1) {
            arr[k] = leftArr[i];
            i++;
            k++;
        }

        // Copy remaining elements of rightArr, if any
        while (j < n2) {
            arr[k] = rightArr[j];
            j++;
            k++;
        }
    }

    // Recursive Merge Sort function
    public static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;

            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);

            merge(arr, left, mid, right);
        }
    }

    public static void main(String[] args) {
        int[] shipmentWeights = {45, 12, 67, 34, 23, 89, 10, 55,
                                 38, 41, 29, 60, 72, 15, 50, 27};

        mergeSort(shipmentWeights, 0, shipmentWeights.length - 1);

        System.out.println(java.util.Arrays.toString(shipmentWeights));
    }
}
```

3. Organize 24 book titles in a library database in alphabetical order.

```java
public class BookTitlesMergeSort {

    // Merge two sorted subarrays into one sorted array
    public static void merge(String[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        String[] leftArr = new String[n1];
        String[] rightArr = new String[n2];

        for (int i = 0; i < n1; i++) {
            leftArr[i] = arr[left + i];
        }
        for (int j = 0; j < n2; j++) {
            rightArr[j] = arr[mid + 1 + j];
        }

        int i = 0, j = 0;
        int k = left;

        while (i < n1 && j < n2) {
            if (leftArr[i].compareTo(rightArr[j]) <= 0) {
                arr[k] = leftArr[i];
                i++;
            } else {
                arr[k] = rightArr[j];
                j++;
            }
            k++;
        }

        while (i < n1) {
            arr[k] = leftArr[i];
            i++;
            k++;
        }

        while (j < n2) {
            arr[k] = rightArr[j];
            j++;
            k++;
        }
    }

    // Recursive Merge Sort function
    public static void mergeSort(String[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;

            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);

            merge(arr, left, mid, right);
        }
    }

    public static void main(String[] args) {
        String[] bookTitles = {
            "To Kill a Mockingbird", "1984", "The Great Gatsby",
            "One Hundred Years of Solitude",, "Jane Eyre",
            "Moby Dick", "War and Peace", "The Odyssey", "Crime and Punishment",
            "The Catcher in the Rye", "Pride and Prejudice", "The Lord of the Rings"
            "The Hobbit", "Brave New World", "Animal Farm", "Wuthering Heights",
            "Great Expectations", "The Divine Comedy", "Hamlet", "The Brothers Karamazov",
            "Les Misérables", "Don Quixote", "Ulysses", "The Iliad"
        };

        mergeSort(bookTitles, 0, bookTitles.length - 1);

        System.out.println(java.util.Arrays.toString(bookTitles));
    }
}
```

1. How does dividing the array into halves in Merge Sort impact its overall time complexity?

> Dividing the array into halves breaks the problem into smaller subproblems, each half being sorted independently. This division happens recursively, resulting in a depth of about $\log_2(n)$. At each level, merging the halves takes O(n) time, so overall time complexity becomes O(n log n). This ensures efficient and predictable sorting performance.

2. How does Merge Sort maintain stability, and why is this beneficial in sorting records with multiple keys?

> Merge Sort maintains stability by always choosing the element from the left subarray first when two elements are equal during merging. Stability preserves the original order of equal elements, which is crucial when sorting records by multiple keys—allowing you to sort first by one key, then by another without losing the previous ordering.

3. Why is Merge Sort more suitable than other algorithms for sorting linked lists or large datasets on disk?

> Merge Sort works well on linked lists because it requires only sequential access, which matches linked list structure. It also merges efficiently without extra space. For large datasets on disk, Merge Sort's sequential read/write pattern minimizes slow random disk access, making it ideal for external sorting where data doesn't fit in memory.

4. What changes are needed to implement Merge Sort iteratively instead of recursively?

> To implement Merge Sort iteratively, use a bottom-up approach by merging subarrays of increasing size (1, 2, 4, 8, ...). Start by merging pairs of single-element arrays, then merge pairs of sorted subarrays of size two, then four, and so on until the whole array is merged. This avoids recursion by using loops to control the merging process.

5. When does Merge Sort's consistent O(n log n) performance outperform Quick Sort despite Quick Sort's faster average case?

> Merge Sort's guaranteed O(n log n) worst-case is preferable when predictable timing is critical, such as in real-time systems or when dealing with data that can cause Quick Sort's worst-case $O(n^2)$ behavior. It's also better when stability is needed or when sorting large external datasets where consistent performance matters.

Use-Case Challenge - Answer

```java
import java.util.Arrays;

public class BookAuthorSorter {

    // Merge two sorted subarrays into one sorted array
    public static void merge(String[] arr, int left, int mid, int right)
    {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        String[] leftArr = new String[n1];
        String[] rightArr = new String[n2];

        for (int i = 0; i < n1; i++) {
            leftArr[i] = arr[left + i];
        }
        for (int j = 0; j < n2; j++) {
            rightArr[j] = arr[mid + 1 + j];
        }

        int i = 0, j = 0, k = left;

        // Merge the temporary arrays back into arr[left..right]
        while (i < n1 && j < n2) {
            if (leftArr[i].compareTo(rightArr[j]) <= 0) {
                arr[k] = leftArr[i];
                i++;
            } else {
                arr[k] = rightArr[j];
                j++;
            }
            k++;
        }

        // Copy remaining elements of leftArr[] if any
        while (i < n1) {
            arr[k] = leftArr[i];
            i++;
            k++;
        }

        // Copy remaining elements of rightArr[] if any
        while (j < n2) {
            arr[k] = rightArr[j];
            j++;
            k++;
        }
    }

    // Recursively divide and sort the array
    public static void mergeSort(String[] arr, int left, int right) {
        if (left < right) {
            int mid = left + (right - left) / 2;

            // Sort first and second halves
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);

            // Merge the sorted halves
            merge(arr, left, mid, right);
        }
    }

    // Main method to run the program
    public static void main(String[] args) {
        String[] authors = { "Smith", "Adams", "Brown", "Clark",
                             "Davis", "Evans", "Ford" };

        System.out.println("Before sorting: " + Arrays.toString(authors));

        mergeSort(authors, 0, authors.length - 1);

        System.out.println("After sorting:  " + Arrays.toString(authors));
    }
}
```