**Algorithms in DSA**

**Sorting:**

Bubble sort [O(n²)]:

* Iterate for (n-1) times.
* Compare with adjacent elements and move the largest element to the end.
* After each **pass** (one complete iteration of the inner loop), the largest unsorted element is moved to its correct position at the end.
* Therefore, after the **first pass**, the largest element is at the end. After the **second pass**, the second largest element is in the correct position, and so on. This means that after n - 1 passes, all elements will be sorted.

A screenshot of a computer code

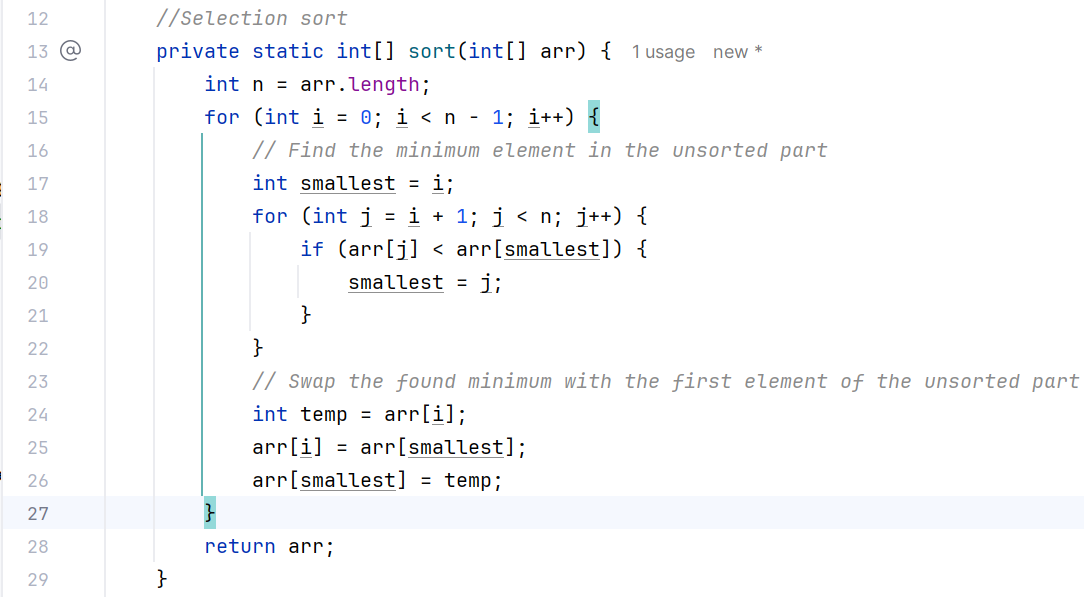
AI-generated content may be incorrect.

Explanation:

1. The **outer loop** runs from 0 to n-1 to ensure all elements are sorted.
2. The **inner loop** compares adjacent elements and swaps them if they are in the wrong order.
3. The **swapped** flag is used to break out early if the array becomes sorted before completing all passes.

Selection sort [O(n²)]:

* Divide the array into two parts: Sorted part (left) and Unsorted part (right)
* **Repeat for each element:**
* Find the **minimum element** from the unsorted part.
* Swap it with the **first unsorted element**.
* Move the boundary between sorted and unsorted parts, one element to the right.



Explanation:

* The **outer loop** iterates from the first element to the second-to-last element (n-1 times).
* Each iteration places the next smallest element at position i.
* The **inner loop** searches for the **remaining unsorted part** of the array (from i+1 to n).
* If a smaller element is found, it updates minIndex to that element's index.
* After finding the minimum element, it swaps the value at minIndex with the value at position i.
* Better than Bubble Sort when the number of swaps is a concern, as it minimizes swaps.

Insertion sort:

* Start with the **first element** (assumed sorted).
* Pick the **next element** and compare it with the elements on its left.
* **Shift elements** to the right to make space and **insert** the picked element at its correct position.
* Repeat until the entire array is sorted.

A computer screen shot of a program

AI-generated content may be incorrect.

Explanation

* **Outer Loop:** Starts from the **second element** (index 1) because the first element is considered sorted.
* **Key:** Holds the current element to be placed correctly in the sorted part.
* **Inner Loop:** Compares the key with elements on the left. **Shifts elements** one position to the right if they are greater than the key. Decrements j to move left.
* **Placement:** Places the key in the correct position after shifting.

Quick sort:

* We need to understand 2 concepts -> pivot and partition

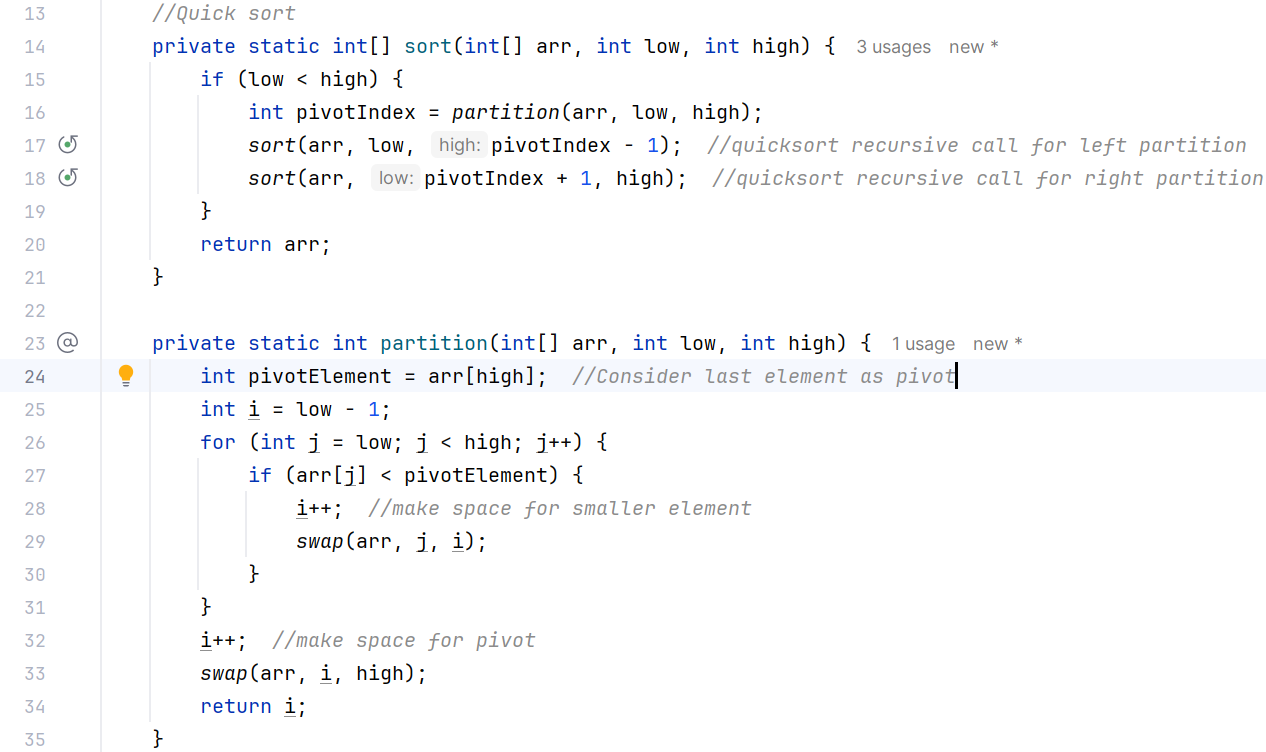
Pivot -> fixed point for which left side contains lower elements while right side contains higher elements.

Partition -> array is divided into left side of pivot and right side of pivot

* We need 2 vars -> low(0) and high(arr.length-1)
* Write a method to take arr, low and high as params
* Find the correct pivot index and then Call the sort method recursively 2 times.

First -> on left partition i.e. from low to (pivot-1)

Second -> on right partition i.e. from (pivot+1) to high.



Merge sort:

A computer screen shot of a program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

Merge Sort is a **Divide and Conquer** algorithm that:

1. Divides the array into two halves.
2. Recursively sorts both halves.
3. Merges the sorted halves.

**Key Takeaways:**

1. **Bubble Sort:** Simple but inefficient for large datasets. Best when the list is almost sorted.
2. **Insertion Sort:** Efficient for small or partially sorted lists. Uses minimal swaps.
3. **Selection Sort:** Always takes O(n2)O(n^2)O(n2) time, making it inefficient for large lists.
4. **Quick Sort:** Highly efficient on average; performance degrades if the pivot selection is poor.
5. **Merge Sort:** Always O(nlog⁡n)O(n \log n)O(nlogn), stable, but requires additional space.

**Java’s Built-in Sorts (Arrays.sort() and Collections.sort()) and How do Java's built-in sorting methods work?:**

Java provides two commonly used methods for sorting:

1. **Arrays.sort()** - for sorting arrays.
2. **Collections.sort()** - for sorting collections (like lists).

**1. How do Java's Built-in Sorting Methods Work?**

* **Arrays.sort() for Primitives:**
  + Uses **Dual-Pivot Quicksort** (since Java 7).
  + Time Complexity:
    - **Best/Average Case:** O(nlog⁡n)O(n \log n)O(nlogn)
    - **Worst Case:** O(n2)O(n^2)O(n2) (rare due to dual pivot).
  + Space Complexity: O(log⁡n)O(\log n)O(logn) due to recursion.
  + **Why Dual-Pivot Quicksort?**
    - Faster than traditional Quicksort.
    - Efficient for primitive data types like int, long, char, etc.
* **Arrays.sort() for Objects:**
  + Uses **Timsort** (a hybrid of Merge Sort and Insertion Sort).
  + Time Complexity:
    - **Best/Average/Worst Case:** O(nlog⁡n)O(n \log n)O(nlogn)
  + Space Complexity: O(n)O(n)O(n) due to merging.
  + **Why Timsort?**
    - Efficient on real-world data, especially when the data is partially sorted.
    - Stable sort (maintains relative order of equal elements).
* **Collections.sort():**
  + Uses **Timsort** internally, as it converts the list to an array and sorts it.
  + Time Complexity: O(nlog⁡n)O(n \log n)O(nlogn).
  + Space Complexity: O(n)O(n)O(n).
  + **Why Timsort?**
    - Provides stable sorting and efficient performance on partially sorted data.

Why does Arrays.sort() have different implementations for primitives and objects?

* Primitives can be sorted in place without boxing/unboxing, making Quicksort faster.
* Objects require comparisons using compareTo() or Comparator, so Timsort's efficiency on partially sorted data is preferable.

When would you use Collections.sort() vs Arrays.sort()?

* Use **Arrays.sort()** for primitive arrays and object arrays.
* Use **Collections.sort()** for sorting List or other collections.
* **Dual-Pivot Quicksort** is fast for primitives, while **Timsort** is stable and efficient for objects and collections.
* Choose based on data type and whether stability is required.