Project Title: Selection Sort Algorithm Implementation

Overview

This project demonstrates a Java-based implementation of the selection sort algorithm. It is designed to sort a collection of strings provided via standard input. The implementation showcases the basic principles of the selection sort, emphasizing clarity and educational value. The goal is to illustrate how the algorithm operates and how it can be used to sort data in ascending order.

Algorithm Description in Plain English

1. **Dividing the Array:**
   The algorithm conceptually splits the list of strings into two sections—a sorted section and an unsorted section. At the start, the sorted section is empty, and all elements belong to the unsorted section.

2. **Finding the Minimum Element:**
   The process begins at the first element of the unsorted section. The algorithm scans the unsorted portion of the list to locate the smallest element based on the natural ordering (or a provided custom comparator).

3. **Swapping Elements:**
   Once the smallest element is identified, it is swapped with the first element of the unsorted section. This action effectively moves that element into the sorted section, increasing its size by one.

4. **Iterative Process:**
   The algorithm then moves to the next element in the unsorted section and repeats the process: find the minimum element in the remaining unsorted section and swap it with the current element. This cycle continues until every element has been moved to the sorted section.

5. **Completing the Sort:**
   By the end of the iterations, the entire list is sorted in ascending order. Each step ensures that a part of the list is correctly ordered, and with each swap, the algorithm progressively builds the sorted portion of the array.

Key Characteristics

- **Simplicity:** The algorithm is straightforward, making it an excellent tool for educational purposes to understand basic sorting techniques.
- **Fixed Number of Swaps:** It performs a predetermined number of exchanges, exactly one swap per iteration, regardless of the initial order of the elements.
- **Performance Considerations:**
  - The algorithm makes roughly half the square of the number of comparisons, resulting in a time complexity of $O(n^2)$.

    - This quadratic time complexity means that while selection sort is efficient for small datasets, it is not the best choice for larger collections of data.
- **Flexibility:**
    - The implementation supports sorting using the natural order of elements (as defined by their inherent comparison method).
    - It also allows for custom ordering by utilizing a comparator, making it adaptable to different types of sorting criteria.
- **Debugging Aids:**
    - The project incorporates checks (assertions) that verify the list is partially sorted after each iteration and fully sorted at the end. This ensures the reliability and correctness of the algorithm during execution.