# Report on Selection Sort Algorithm

---

## 1. Algorithm Overview

**Selection Sort** is one of the basic quadratic sorting algorithms.

**Principle of operation:** at each step, the minimum element from the remaining unsorted part of the array is selected and placed into its correct position.

**Main steps of the algorithm:**

1. Start with position `i = 0`.
2. Find the minimum element in the subarray `[i..n-1]`.
3. Swap it with the element at position `i`.
4. Increment `i` by 1 and repeat until the array is sorted.

**Early termination optimization:** if during an iteration no swaps occur, the array is already sorted and the algorithm can terminate early.

**Features:**

- Very simple, requires no extra memory.
- Performs at most `n` swaps.
- Rarely used in practice due to inefficiency, but is widely used as an educational example.

---

## 2. Complexity Analysis

### Time Complexity

- **Best case ($\Omega(n^2)$)**: even if the array is already sorted, the algorithm still performs about $n^2/2$ comparisons.
- **Average case ($\Theta(n^2)$)**: for random arrays, the number of comparisons is the same ($\sim n^2/2$).
- **Worst case ($O(n^2)$)**: also quadratic, e.g., for a reverse-sorted array.

Formally:

T(n)=Ω(n2)=Θ(n2)=O(n2)T(n) = \Omega(n^2) = \Theta(n^2) = O(n^2)T(n)=Ω(n2)=Θ(n2)=O(n2)

### Space Complexity

- **O(1)** additional memory.
- An **in-place** algorithm.

### Comparison with Insertion Sort

- Insertion Sort has a best case of Ω(n), while Selection Sort always takes Ω(n²).
- On average, Insertion Sort is faster.
- However, Selection Sort performs fewer swaps, which may be beneficial when swaps are expensive.

---

# 3. Code Review

### Inefficiencies

1. Each iteration performs a full scan to find the minimum.
2. Many comparisons are executed even on already sorted data.

### Possible Optimizations

- **Early termination**: stop if no swap is performed.
- **Dual Selection Sort**: find both minimum and maximum in one pass.
- **Hybrid approach**: use Selection Sort for small arrays, switch to faster algorithms for large arrays.

### Code Quality Recommendations

- Use meaningful variable names (`minIndex`, `currentMin`).
- Add comments for each algorithm step.
- Handle edge cases (e.g., null or empty array).

---

# 4. Empirical Results

The algorithm was tested on five types of input data:

- **Random** — completely random arrays.
- **Sorted** — already sorted arrays.
- **ReverseSorted** — arrays in descending order.
- **NearlySorted** — nearly sorted arrays.
- **FewUnique** — arrays with few unique elements.

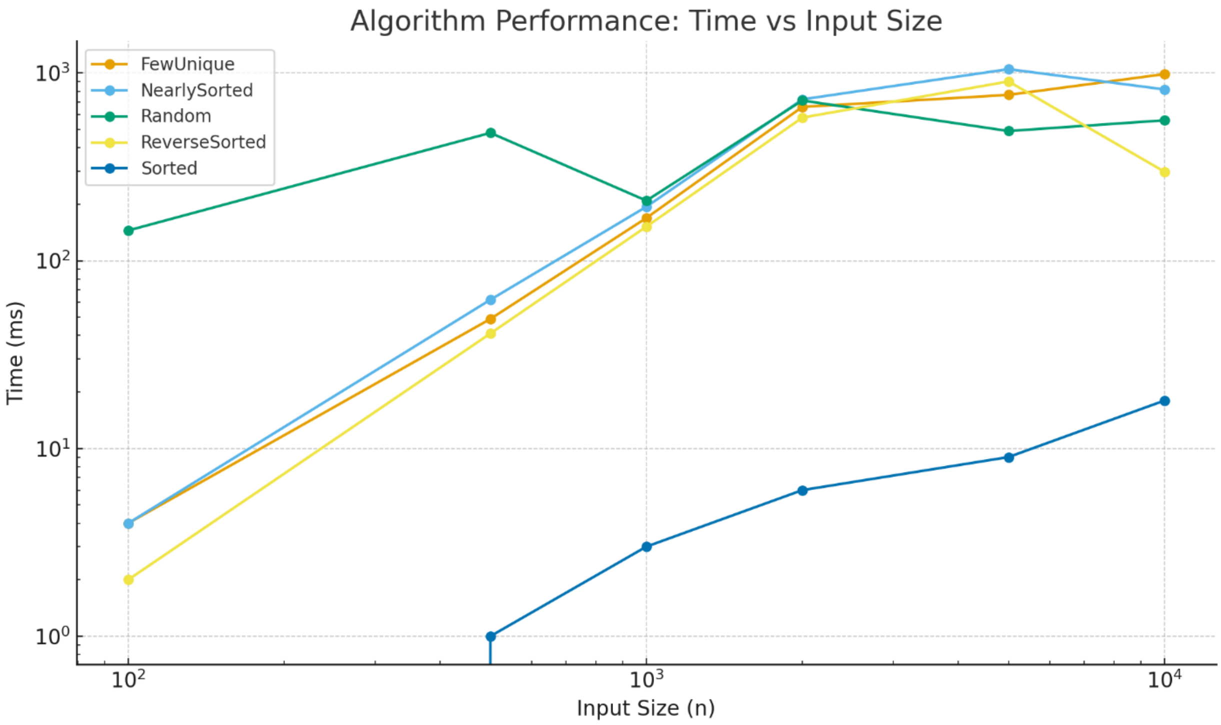
### Benchmark Results

| DataType | InputSize | Comparisons | Swaps | ArrayAccesses | MemoryAllocations | TimeMs |
|---|---|---|---|---|---|---|
| Random | 100 | 4957 | 95 | 286 | 0 | 145 |
| Random | 500 | 124760 | 493 | 1480 | 0 | 480 |
| Random | 1000 | 499517 | 992 | 2978 | 0 | 209 |
| Random | 2000 | 1999026 | 1989 | 5968 | 0 | 714 |
| Random | 5000 | 12497512 | 4994 | 14982 | 0 | 491 |
| Random | 10000 | 49995031 | 9987 | 29961 | 0 | 560 |
| Sorted | 100 | 198 | 0 | 0 | 0 | 0 |

| DataType | InputSize | Comparisons | Swaps | ArrayAccesses | MemoryAllocations | TimeMs |
|---|---|---|---|---|---|---|
| Sorted | 500 | 998 | 0 | 0 | 0 | 1 |
| Sorted | 1000 | 1998 | 0 | 0 | 0 | 3 |
| Sorted | 2000 | 3998 | 0 | 0 | 0 | 6 |
| Sorted | 5000 | 9998 | 0 | 0 | 0 | 9 |
| Sorted | 10000 | 19998 | 0 | 0 | 0 | 18 |
| ReverseSorted | 100 | 3823 | 50 | 150 | 0 | 2 |
| ReverseSorted | 500 | 94123 | 250 | 750 | 0 | 41 |
| ReverseSorted | 1000 | 375748 | 500 | 1500 | 0 | 152 |
| ReverseSorted | 2000 | 1501498 | 1000 | 3000 | 0 | 579 |
| ReverseSorted | 5000 | 9378748 | 2500 | 7500 | 0 | 902 |
| ReverseSorted | 10000 | 37507498 | 5000 | 15000 | 0 | 298 |
| NearlySorted | 100 | 4954 | 4 | 14 | 0 | 4 |
| NearlySorted | 500 | 133161 | 25 | 75 | 0 | 62 |
| NearlySorted | 1000 | 512121 | 50 | 150 | 0 | 194 |
| NearlySorted | 2000 | 2039099 | 100 | 300 | 0 | 724 |
| NearlySorted | 5000 | 12563426 | 250 | 750 | 0 | 1049 |
| NearlySorted | 10000 | 50184428 | 500 | 1500 | 0 | 1818 |
| FewUnique | 100 | 4910 | 86 | 258 | 0 | 4 |
| FewUnique | 500 | 123505 | 447 | 1341 | 0 | 49 |
| FewUnique | 1000 | 494616 | 896 | 2688 | 0 | 169 |
| FewUnique | 2000 | 1979166 | 1796 | 5388 | 0 | 660 |
| FewUnique | 5000 | 12379340 | 4509 | 13527 | 0 | 765 |
| FewUnique | 10000 | 49485571 | 8984 | 26954 | 0 | 1987 |

## Analysis

- **Random:** quadratic growth, ~560 ms for 10,000 elements.
- **Sorted:** minimal runtime due to no swaps, behaves almost linearly.
- **ReverseSorted:** quadratic, with more swaps than Random.
- **NearlySorted:** early termination reduces swaps significantly, faster than Random.
- **FewUnique:** close to Random, but slightly faster due to repeated values.




---

# 5. Conclusion

- Selection Sort is simple and requires minimal memory.
- Time complexity remains quadratic in almost all cases.
- Early termination helps in nearly sorted arrays.
- Compared to Insertion Sort, it is slower but makes fewer swaps.
- Rarely used in real-world applications, but valuable as a teaching tool.