Selection Sort

Varun Kumar

July 4, 2025

1. Logic

Selection Sort repeatedly finds the minimum element from the unsorted part of the array and puts it at the beginning.

Key Idea

At the i^{th} iteration, the smallest element from index i to $n-1$ is selected and swapped with the element at index i.

2. Number of Comparisons and Swaps

Let n be the number of elements.

Worst, Best, and Average Case

- Comparisons: $\frac{n(n-1)}{2}$
- Swaps: $n-1$

3. Optimal Behavior

Selection Sort does not adapt to the input data. It performs the same number of comparisons regardless of the input's order.

4. Pseudocode

```
function selectionSort(arr):
    n = length(arr)
    for i = 0 to n-2:
        min_idx = i
        for j = i+1 to n-1:
            if arr[j] < arr[min_idx]:
                min_idx = j
        swap arr[i] and arr[min_idx]
```

Stability Note

Selection Sort is not stable because it may change the relative order of equal elements due to swapping.

5. Example Walkthrough

Given: `[5, 1, 4, 2, 8]`

Pass 1

Find min from index 0 to 4 → 1 Swap 5 and 1 → `[1, 5, 4, 2, 8]`

Pass 2

Find min from index 1 to 4 → 2 Swap 5 and 2 → `[1, 2, 4, 5, 8]`

Pass 3

Find min from index 2 to 4 → 4 Already in position → No swap

Pass 4

Find min from index 3 to 4 → 5 Already in position → No swap

6. Mixed Case Behavior in Selection Sort

Input Array:

```
[(1, 'A'), (3, 'B'), (5, 'C'), (2, 'D'), (4, 'E')]
```

This example includes both:

- **Best-case behavior:** some elements (e.g., (1, 'A')) are already in the correct position.
- **Worst-case behavior:** some elements (e.g., (5, 'C')) need to move far.

Pass 1 (i = 0)

Find min from index 0 to 4 → (1, 'A') (already in correct position)

No shift performed.

```
[(1, 'A'), (3, 'B'), (5, 'C'), (2, 'D'), (4, 'E')]
```

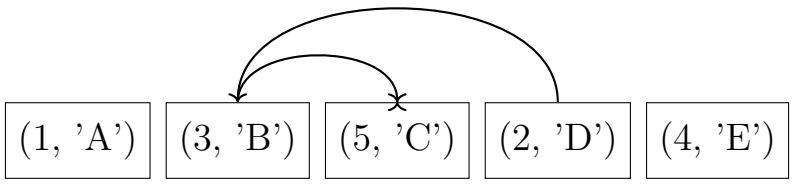
Pass 2 (i = 1)

Find min from index 1 to 4 → (2, 'D')

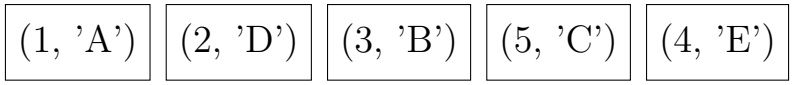
Insert it at index 1 by shifting right:

```
[(1, 'A'), (2, 'D'), (3, 'B'), (5, 'C'), (4, 'E')]
```

Visual for Pass 2 (Insert (2, 'D'))



After shifting:



What Happened

(2, 'D') was the minimum from index 1 to 4. Instead of swapping directly with (3, 'B'), we shifted (3, 'B') and (5, 'C') one step to the right, and inserted (2, 'D') at index 1. This preserved the relative order → **stable sort**.

Pass 3 (i = 2)

Find min from index 2 to 4 → (3, 'B') (already in correct position)
No shift performed.

```
[(1, 'A'), (2, 'D'), (3, 'B'), (5, 'C'), (4, 'E')]
```

Pass 4 (i = 3)

Find min from index 3 to 4 → (4, 'E')
Shift and insert at index 3:

```
[(1, 'A'), (2, 'D'), (3, 'B'), (4, 'E'), (5, 'C')]
```

Pass 5 (i = 4)

Only one element remains → Done.

Final Sorted Output

```
[(1, 'A'), (2, 'D'), (3, 'B'), (4, 'E'), (5, 'C')]
```

Summary of Mixed Behavior

- (1, 'A') and (3, 'B') were already in place → **Best-case behavior**.
- (5, 'C') moved from index 2 to 4 → **Worst-case behavior**.

This demonstrates both behaviors in the same execution.

7. Python Code with Explanation

```
def selection_sort(arr):
    n = len(arr) # Get the length of the array

    # Traverse the array from index 0 to n-2
    for i in range(n - 1):
        min_idx = i # Assume the minimum is at the current index

        # Find the index of the minimum element in the unsorted part
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                # Update min_idx if a smaller element is found
                min_idx = j

        # Swap the found minimum element with the element
        # at the current index i
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr # Return the sorted array
```

8. Why is Selection Sort Unstable?

Selection Sort is **unstable** because it swaps elements even when their keys are equal, which can change the original relative order of those elements.

Illustration with an Example

Consider the array of tuples where each tuple contains a key and a label:

```
arr = [(4, 'A'), (4, 'B'), (3, 'C')]
```

Step 1: Find the minimum element $\rightarrow (3,'C')$
Step 2: Swap it with the first element $\rightarrow (4,'A')$
Resulting array:

```
[(3, 'C'), (4, 'B'), (4, 'A')]
```

Now, the element $(4,'A')$ comes **after** $(4,'B')$, even though originally it appeared **before**. This violates the principle of *stability*, where equal keys must retain their input order.

Why This Happens in Logic

In the core logic of selection sort:

```
arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

This line swaps unconditionally, without checking if the elements are equal, thus potentially breaking the original order.

Summary

Selection Sort is not stable because it uses direct swapping, which may reorder equal elements and break their original sequence.

9. Making Selection Sort Stable

Selection Sort can be made **stable** by avoiding direct swapping. Instead of swapping the minimum element with the current index, we **shift** all elements to the right and **insert** the minimum element at its correct position.

Stable Selection Sort Logic

Key Idea

Instead of swapping the minimum element directly with the current index, store the minimum, shift the intermediate elements to the right, and insert the minimum value at the correct index.

Stable Selection Sort Code (Python)

```
def stable_selection_sort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j

        # Store the minimum element
        key = arr[min_idx]

        # Shift all elements between i and min_idx to the right
        while min_idx > i:
            arr[min_idx] = arr[min_idx - 1]
            min_idx -= 1

        # Insert the minimum element at its correct position
        arr[i] = key
```

Result

This version of Selection Sort is **stable** and maintains the relative order of equal elements.

10. Example Walkthrough of Stable Selection Sort

Initial Array:

```
[(4, 'A'), (5, 'B'), (3, 'C'), (4, 'D'), (1, 'E'), (2, 'F'), (6, 'G')]
```

Pass 1 (i = 0)

Find minimum from index 0 to 6 → (1, 'E')
Insert it at index 0 by shifting others right:

```
[(1, 'E'), (4, 'A'), (5, 'B'), (3, 'C'), (4, 'D'), (2, 'F'), (6, 'G')]
```

Pass 2 (i = 1)

Find minimum from index 1 to 6 → (2, 'F')
Shift and insert:

```
[(1, 'E'), (2, 'F'), (4, 'A'), (5, 'B'), (3, 'C'), (4, 'D'), (6, 'G')]
```

Pass 3 (i = 2)

Find minimum from index 2 to 6 → (3, 'C')
Shift and insert:

```
[(1, 'E'), (2, 'F'), (3, 'C'), (4, 'A'), (5, 'B'), (4, 'D'), (6, 'G')]
```

Pass 4 (i = 3)

Find minimum from index 3 to 6 → (4, 'A') (already in place)
No shift needed.

```
[(1, 'E'), (2, 'F'), (3, 'C'), (4, 'A'), (5, 'B'), (4, 'D'), (6, 'G')]
```

Pass 5 (i = 4)

Find minimum from index 4 to 6 → (4, 'D')

Shift and insert:

```
[(1, 'E'), (2, 'F'), (3, 'C'), (4, 'A'), (4, 'D'), (5, 'B'), (6, 'G')]
```

Pass 6 (i = 5)

Find minimum from index 5 to 6 → (5, 'B') (already in place)

```
[(1, 'E'), (2, 'F'), (3, 'C'), (4, 'A'), (4, 'D'), (5, 'B'), (6, 'G')]
```

Pass 7 (i = 6)

Only one element left → Done.

Final Output

```
[(1, 'E'), (2, 'F'), (3, 'C'), (4, 'A'), (4, 'D'), (5, 'B'), (6, 'G')]
```

Stability Observed

(4, 'A') came before (4, 'D') and remains in that order after sorting. Hence, the algorithm is stable.

11. Best and Worst Case of Selection Sort

Selection Sort is **not adaptive**, which means it performs the same number of comparisons regardless of the input order.

Best Case

- **Input:** Already sorted array `[1, 2, 3, 4, 5]`
- **Comparisons:** Always $\frac{n(n-1)}{2}$
- **Swaps:** Exactly $n-1$, even if elements are already in place (can be reduced with optimization to skip unnecessary swaps)

Time Complexity: $O(n^2)$
Swaps: $O(n)$

Worst Case

- **Input:** Reverse sorted array `[5, 4, 3, 2, 1]`
- **Comparisons:** Still $\frac{n(n-1)}{2}$
- **Swaps:** $n-1$

Time Complexity: $O(n^2)$
Swaps: $O(n)$

Summary Table

Case	Comparisons	Swaps	Time	Adaptive	Stable
Best Case	$\frac{n(n-1)}{2}$	$n-1$	$O(n^2)$	No	No
Worst Case	$\frac{n(n-1)}{2}$	$n-1$	$O(n^2)$	No	No

Note

- The number of comparisons is always the same: $\frac{n(n-1)}{2}$.
- Selection Sort is useful when:
  - Swaps are expensive (since it does at most $n-1$ swaps).
  - Simplicity or small input size is acceptable.

12. Time & Space Complexity and Its Properties

Case	**Complexity**	**Property**	**Value**
Best Case	$O(n^2)$	Stable	No
Average Case	$O(n^2)$	In-place	Yes
Worst Case	$O(n^2)$	Adaptive	No
Space Complexity	$O(1)$ (in-place)	Recursive	No