Newton School of Technology

1

Selection Sort Algorithm

by Gladden Rumao

CSA121 : DSA

Sorting

The process of **arranging the elements** in a specific order.
Imagine sorting a list of names, numbers, or search results.

Motivation:

- For **the small data set** we can use selection sort.

- It is **space efficient** algorithm.

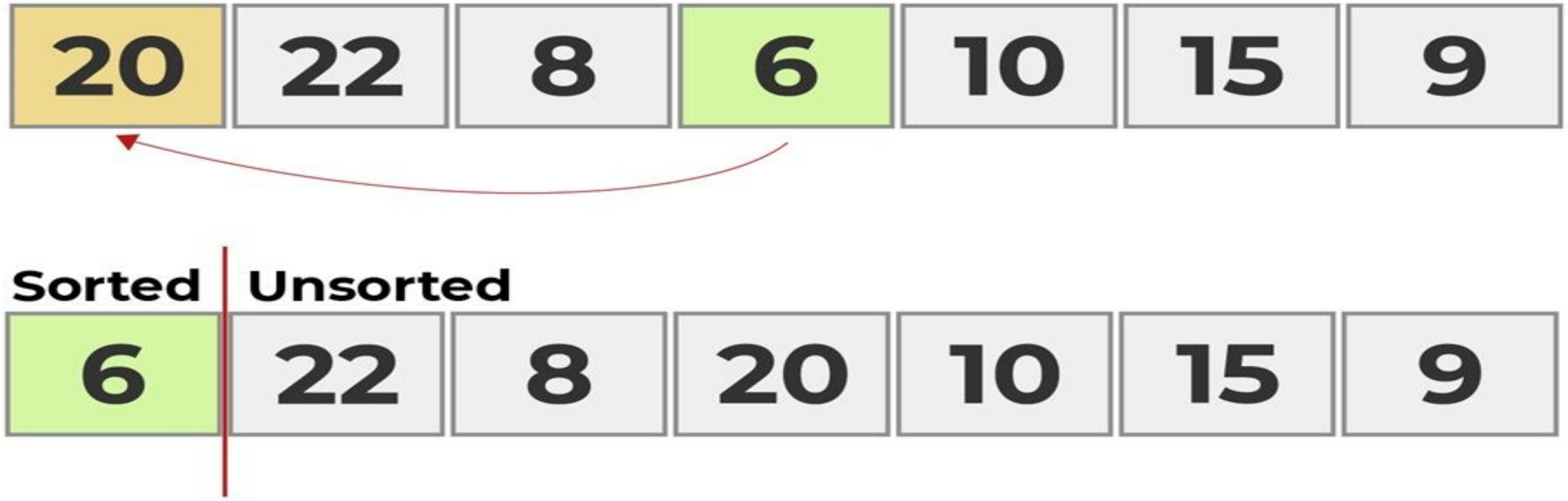
- Idea of selection sort can be **extended to more complex algorithms, like quicksort and heapsort**, that build on partitioning and selection concepts.

Newton School of Technology

Selection Sort

Why it is called Selection Sort:

Because this algorithm works by selecting the smallest element from a list of elements and placing it in its correct position.



Example - Imagine organizing a bookshelf

- You scan all the books to find the one that should come **first (alphabetically)**,
- Move it to the front by swapping it with the front position book.
- Repeat the process for the next position until all are in order.

Working principle

Step 1 :

Selection Process :
The algorithm repeatedly selects the smallest element from the unsorted portion of the list.

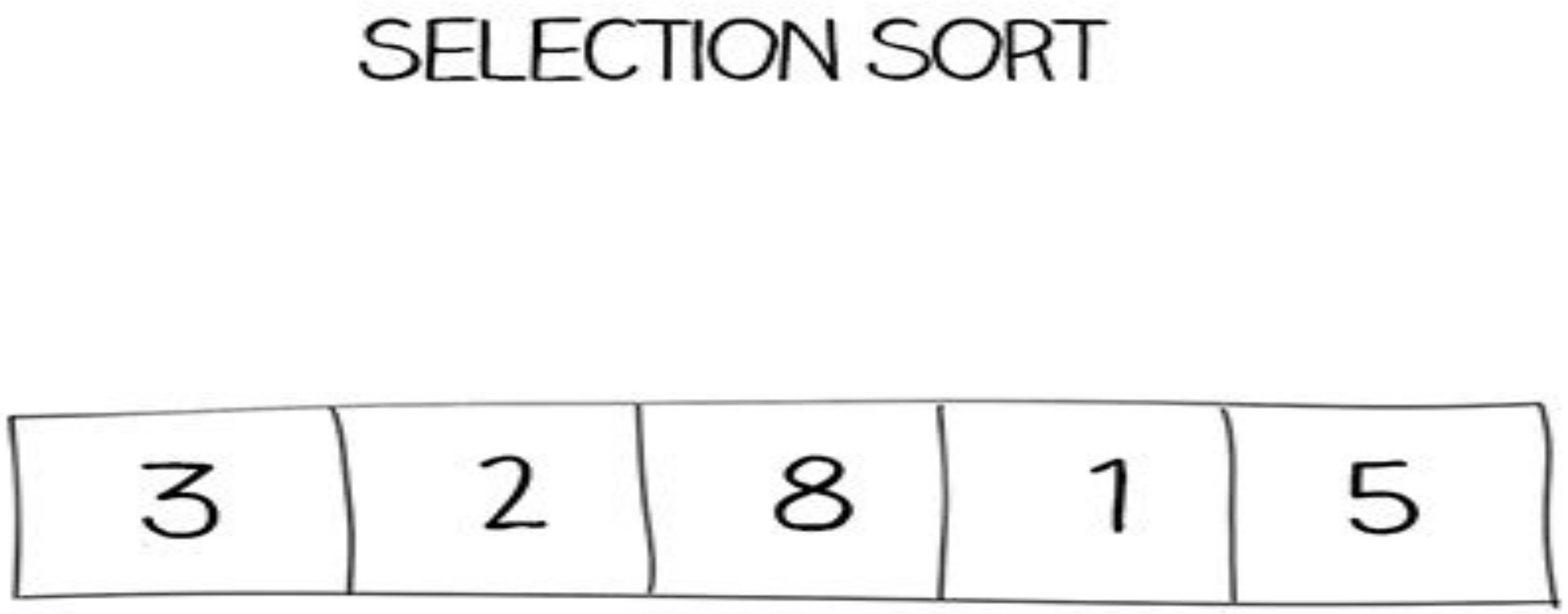
Step 2 :

Placement :
After selecting the smallest element, it is placed in its correct position in the sorted portion of the list.

Step 3 :

Step - by - Step Selection :
This process of selecting and placing elements continues until all elements are sorted.

Selection Sort Animation :



Selection Sort - Example

Example : Selection Sort

INDEX	0	1	2	3	4
ELEMENT	15	14	13	12	11

Example : Iteration – 1

INDEX	0	1	2	3	4
ELEMENT	15	14	13	12	11

INDEX	0	1	2	3	4
ELEMENT	15	14	13	12	11

Swap the elements at index **0 & 4**

After Swapping

INDEX	0	1	2	3	4
ELEMENT	11	14	13	12	15

Number of comparisons = 4

Number of Swaps = 1

Example : Iteration – 2

INDEX	1	2	3	4
ELEMENT	14	13	12	15

INDEX	1	2	3	4
ELEMENT	14	13	12	15

Swap the elements at index **1 & 3**

After Swapping

INDEX	0	1	2	3	4
ELEMENT	11	12	13	14	15

Number of comparisons = 3

Number of Swaps = 1

Example : Iteration – 3

INDEX	2	3	4
ELEMENT	13	14	15

INDEX	2	3	4
ELEMENT	13	14	15

No swap would be there

List is

INDEX	0	1	2	3	4
ELEMENT	11	12	13	14	15

Number of comparisons = 2

Number of Swaps = 0

Example : Iteration – 4

INDEX	3	4
ELEMENT	14	15

INDEX	3	4
ELEMENT	14	15

No swap would be there

List is

INDEX	0	1	2	3	4
ELEMENT	11	12	13	14	15

Number of comparisons = 1

Number of Swaps = 0

Post Iteration - 4 :

INDEX	4
ELEMENT	15

One element is sorted in itself

Sorted List is

INDEX	0	1	2	3	4
ELEMENT	11	12	13	14	15

Key Observation Are:

- **After the first pass**, the smallest element is placed at the beginning.
- **After the second pass**, the two smallest are in place. This continues until the entire array is sorted.
- It requires **exactly (n-1)** iterations to sort the list of elements.
- **Unlike Bubble Sort, Selection Sort does fewer swaps** (at most one swap per pass), but still does many comparisons.
- **Not Stable:** Selection Sort can reorder equal elements arbitrarily since you might swap identical elements from different positions. Thus, it is generally not considered stable.

Code Implementation :

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

        if i != min_index:
            arr[i], arr[min_index] =
                arr[min_index], arr[i]

    return arr
```

Performance Analysis - Selection Sort

	Number of Comparisons	Number of Swaps
1st Iteration	n-1	1
2nd Iteration	n-2	1
3rd Iteration	n-3	1
. .	. .	. .
(N-1)th Iteration	1	1

Performance Analysis – Selection Sort

Total number of comparisons = n (n-1) /2

Total number of swaps = n-1

Time Complexity = n (n-1) /2 + (n-1)
= $O(n)^2$

Performance Analysis - Best Case

Data is already in sorted order.

Performance Analysis - Best Case

- Selection sort does not take advantage of the already sorted array.

- It follows the same fixed approach for both sorted and unsorted array.

Observing

Selection Sort – Best Case Example

Newton School of Technology

Best Case Example : Iteration – 1

INDEX	0	1	2	3	4
ELEMENT	11	12	13	14	15

INDEX	0	1	2	3	4
ELEMENT	11	12	13	14	15

No swap would be there

After Iteration – 1

INDEX	0	1	2	3	4
ELEMENT	11	12	13	14	15

Number of comparisons = 4

Number of Swaps = 0

Best Case Example : Iteration – 2

INDEX	1	2	3	4
ELEMENT	12	13	14	15

INDEX	1	2	3	4
ELEMENT	12	13	14	15

No swap would be there

After Iteration – 2

INDEX	0	1	2	3	4
ELEMENT	11	12	13	14	15

Number of comparisons = 3

Number of Swaps = 0

Best Case Example : Iteration – 3

INDEX	2	3	4
ELEMENT	13	14	15

INDEX	2	3	4
ELEMENT	13	14	15

No swap would be there

After Iteration – 3

INDEX	0	1	2	3	4
ELEMENT	11	12	13	14	15

Number of comparisons = 2

Number of Swaps = 0

Best Case Example : Iteration – 4

INDEX	3	4
ELEMENT	14	15

INDEX	3	4
ELEMENT	14	15

No swap would be there

After Iteration – 4

INDEX	0	1	2	3	4
ELEMENT	11	12	13	14	15

Number of comparisons = 1

Number of Swaps = 0

Post Iteration-4 :

After Iteration 4, only one element remains unsorted in the array.

A **single element is inherently sorted**, as there are no other elements to compare or swap with.

Index	Data Items
4	15

Performance Analysis - Best Case

	Number of Comparisons	Number of Swaps
1^{st} Iteration	n-1	0
2^{nd} Iteration	n-2	0
3^{rd} Iteration	n-3	0
. .	. .	. .
$(N-1)^{th}$ Iteration	1	0

Newton School of Technology

Performance Analysis - Best Case

Total number of comparisons = n (n-1) /2

Total number of swaps = 0

Time Complexity = n (n-1) /2 + 0
= $O(n)^2$

Key points of Selection Sort :

- It does depend upon the input order of the data elements.

- Selection sort does not take advantage of the already sorted array.

- It follows the same fixed approach for both sorted and unsorted array.

- So both the best-case and worst-case scenarios for Selection Sort have the same time complexity, which is **$O(n^2)$**

Space Complexity of Selection Sort

Selection Sort has a space complexity of O(1)

It **does not require any additional space** proportional to the input size. Sorting is achieved by swapping adjacent elements directly within the input list.

END