

Selection Sort

Selection Sort works on placing smallest element to its correct position until every element is not in its correct position.

→ Smallest element leake, usko right jagah par rakha hai.

arr[] = 64, 25, 12, 22, 11
 0 1 2 3 4
 ↑
 min

$n = 5$

Note:

Swap smallest element in the array to its 1st position. And second smallest to its 2nd position and so on.

Round 1: 64 25 12 22 11
 i=0 ↑ swap

Round 2: 11 25 12 22 64
 i=1 ↑ swap

Round 3: 11 12 25 22 64
 i=2 ↑ swap

Round 4: 11 12 22 25 64
 i=3 ↑ No swapping.

Total Round = $(n-1)$

Note:

Stable Algorithm: If the two element have the same value and appear in the input in a particular order a stable sorting algorithm will always arrange them in same order.

Selection Sort is not stable because it does not guarantee the relative order of equal element in the input list after sorting.

For Example:

5, 3, 5, 2 $n=4$
_{0 1 2 3}

R1: 5 3 5 2
_{i=0} ↗ swap ↘

R2: 3 5 2 5 (5 is moved after 5 but its original 5 is after it)
 This situation is not stable.

→ When to use selection sort:

→ For small list (100-200)

→ When stability not required

→ Memory efficient

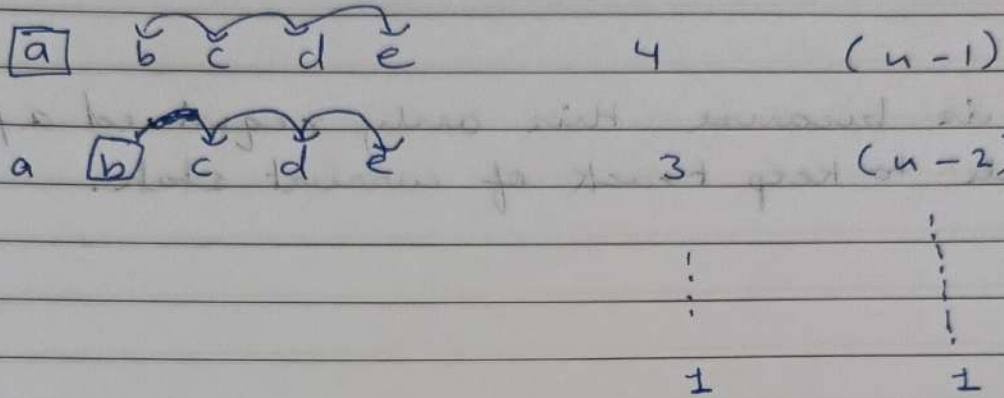
→ When Not to use selection sort:

→ For large list

→ Unstable

Time Complexity:

size = n



$$= (n-1) + (n-2) + \dots + 2 + 1$$

$$= \frac{n(n-1)}{2}$$

$$= \frac{n^2 - n}{2}$$

$$\text{Time Complexity} = O(n^2)$$

Best Case (Already Sorted) = $O(n^2)$

Worst Case (Totally Unsorted) = $O(n^2)$

During Sorted arr we still have to go for all rounds.

Space Complexity:

Space Complexity = $O(1)$

This is because this only required a few variables to keep track of current state.

Code:

```
void selection(int *arr, int size)
{
    for (int i = 0; i < size; i++)
    {
        int min = i;
        for (int j = i; j < size; j++)
        {
            if (arr[j] < arr[min])
            {
                min = j;
            }
        }
        swap(arr[min], arr[i]);
    }
}
```