

## Selection\_Sort

Given an array, we find the lowest element value and swap it in order.

### Steps

- we loop through array, from left to right.
- we keep a track of the lowest\_index\_val.
- for each pass through, if lowest value is not in position, then we swap the pass through index with lowest value.
- We repeat the above steps till the second last index is sorted.

Eg:

0	1	2	3	4
4	2	7	1	3

1st pass through,

Initially, assuming the first element to be least, so lowest\_index\_value has the '0' index.

we loop through array, and find the lowest value and if found we will update the lowest\_index\_value.

Here in this case, index '3' is the lowest, we will compare it with the

passthrough index ('0' in this case, outer loop index), if both are different, then swap the value.

So now the array becomes,

1	2	7	4	3
---	---	---	---	---

we know that, first index of the array is sorted now, so 2<sup>nd</sup> passthrough begins from 2<sup>nd</sup> index (index 1).

We loop through and find the lowest\_index\_value, if both are different we swap.

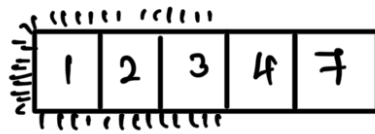
So after 2<sup>nd</sup> passthrough, array becomes

1	2	7	4	3
---	---	---	---	---

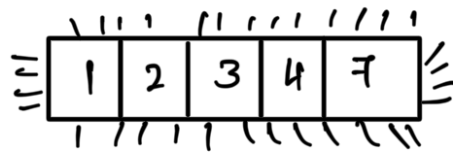
Since no value was lesser than the lowest\_index\_value and lowest\_index\_value was equal to passthrough index, swapping isn't done, element is in correct order.

In 3<sup>rd</sup> passthrough, passthrough index is '2', assuming it to be the lowest value, looping through the remaining elements of

the array, finding the lowest and swapping.  
so array becomes,



in 4<sup>th</sup> pass through, we follow the above steps, after which, 2<sup>nd</sup> last element is sorted,



Since all the elements till 2<sup>nd</sup> last has been sorted, we can say that last element is also sorted, and end the program.

### Implementation :

```
func selectionSort (arr)
```

```
{
```

```
    for (i = 0 → len(arr) - 1)
```

```
        lowest_ind_val = i
```

Finding  
lowest value  
index

```
        for (j = i + 1 → len(arr) - 1)
```

```
            if (arr[j] < arr[lowest_ind_val])
```

```
                lowest_ind_val = j
```

```
    if (lowest_ind_val != i)
```

swap(arr, lowest\_index\_val, i)

return arr;  
}

### Efficiency :

Selection sort has two kinds of steps,

- (i). Comparison - where we compare and find the lowest index value.
- (ii). Swap - Swap the element, swapping is done either one or zero, for each pass through.

for  $N$  elements,

(i). Comparison is done for  $\frac{N(N-1)}{2}$  steps.

(ii). Swapping is done, either one or zero.

Since Swapping is done maximum one per pass through, if lowest-index-value is not in correct position else no swapping is done.

Compare to Bubble sort, in worst-case, we have to swap for each comparison.

This makes Selection sort faster than

Bubble sort.

mathematically, Selection sort takes  $\left(\frac{N^2}{2}\right)$  steps in worst case and Bubble sort takes  $(N^2)$ .

Selection sort is twice as fast as Bubble sort but one of the major rule of Big O is that Ignore constants, because of which both selection sort and bubble sort becomes same  $O(N^2)$ .

Big O Notation Ignores constants
----------------------------------

Role of Big O :

Purpose of Big O is that for different classification, there will be a point at which one classification supersedes the other in speed, and will remain faster forever. When that point occurs exactly, However, it is not the concern of Big O.

If two algorithm falls under 2 different classification, we can easily say which algorithm will be faster with large

amount of data.

However, when it falls under some classification, further analysis is required to determine which algorithm is faster.

As falling under the same classification doesn't necessarily means both process the same speed.

eg: selection sort and Bubble sort.