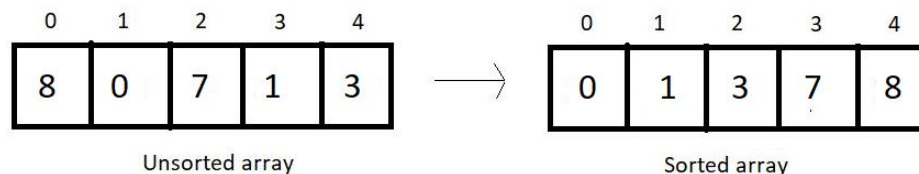


Selection Sort Algorithm

Suppose we are given an array of integers, and we are asked to sort them using the selection sort algorithm, then the array after being sorted would look something like this.



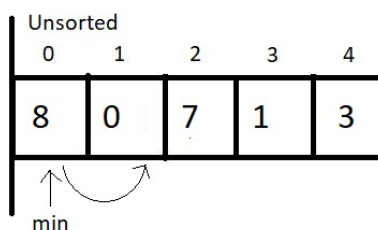
In selection sort, at each pass, we make sure that the smallest element of the current unsorted subarray reaches its final position. And this is pursued by finding the smallest element in the unsorted subarray and replacing it at the end with the element at the first index of the unsorted subarray. This algorithm reduces the size of the unsorted part by 1 and increases the size of the sorted part by 1 at each respective pass. Let's see how this work.

At each pass, we create a variable *min* to store the index of the minimum element. We start by assuming that the first element of the unsorted subarray is the minimum. We will iterate through the unsorted part of the array, and compare every element to this element at *min* index. If the element is less than the element at *min* index, we replace *min* by the current index and move ahead. Else, we keep going. And when we reach the end of the array, we replace the first element of the unsorted subarray with the element at *min* index. And doing this at every pass ensures that the smallest element of the unsorted part of the array reaches its final position at the end.

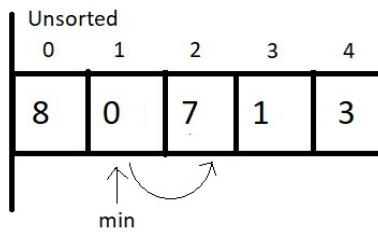
Since our array is of length 5, we will make 4 passes.

1st Pass:

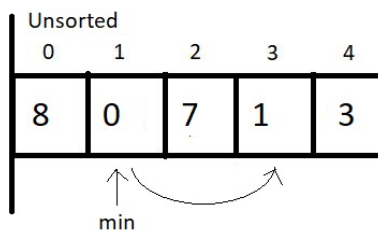
At first pass, our whole array comes under the unsorted part. We will start by assuming 0 as the *min* index. Now, we'll have to check among the remaining 4 elements if there is still a lesser element than the first one.



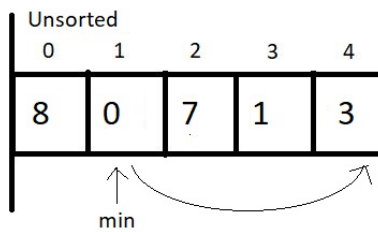
And when we compared the element at *min* index with the element at index 1, we found that 0 is less than 8 and hence we update our *min* index to 1.



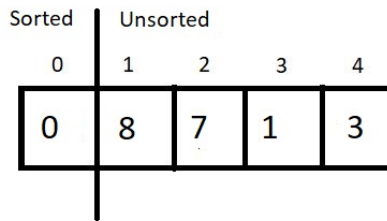
And now we keep checking with the updated *min*. Since 7 is not less than 0, we move ahead.



And now we compared the elements at index 1 and 3, and 0 is still lesser than 1, so we move ahead without making any changes.

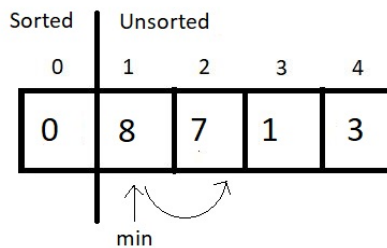


And now we compared the element at the *min* index with the last element. Since there is nothing to change, we end our 1st pass here. Now we simply replace the element at 0th index with the element at the *min* index. And this gives us our first sorted subarray of size 1. And this is where our first pass finishes. We should make an overview of what we received at the end of the first pass.

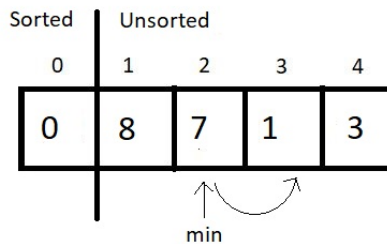


2nd Pass:

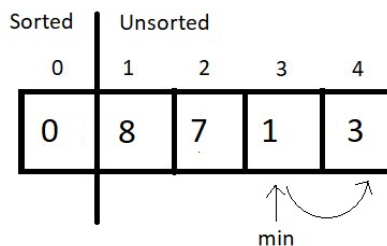
We now start from the beginning of the unsorted array, with a reduced unsorted part of length 4. Hence the number of comparisons would be just 3. We assume the element at index 1 is the one at the *min* index and start iterating to the right for finding the minimum element.



Since 7 is less than 8, we update our *min* index with 2. And move further.



Next, we compared the elements 7 and 1, and since 1 is still lesser than 7, we update the *min* index by 3. Then, we move ahead to the next comparison.




And since 3 is greater than 1, we don't make any changes here. And since we are finished with the array, we stop our pass here itself, and swap the element at index 1 with this element at *min* index. And that would be it for the second pass. Let's see how close we have reached to the sorted array.

Sorted		Unsorted		
0	1	2	3	4
0	1	7	8	3

3rd Pass:


We'll again start from the beginning of the unsorted subarray which is from the index 2, and make the *min* index equal to 2 for now. And this time our unsorted part has a length 3, hence no. of comparisons would be 2.

Sorted		Unsorted		
0	1	2	3	4
0	1	7	8	3

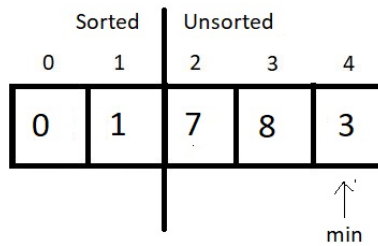


Since 8 is greater than 7, we would make no change, but move ahead.

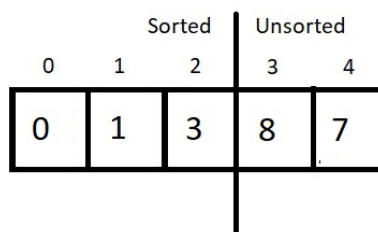
Sorted		Unsorted		
0	1	2	3	4
0	1	7	8	3



Comparing the elements at index *min* and 4, we found 3 to be smaller than 7 and hence an update is needed here. So, we update *min* to 4.

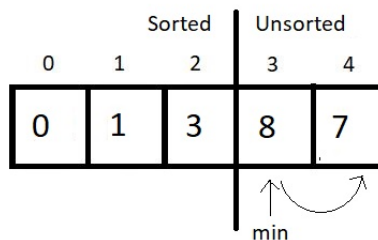


And since that was the last comparison of the third pass, we make a swap of the indices 2 and *min*. And the result at the end would be:

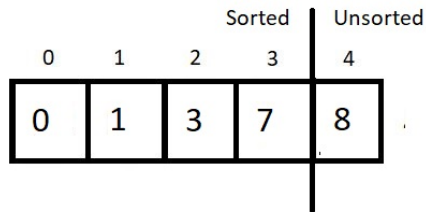


4th Pass:

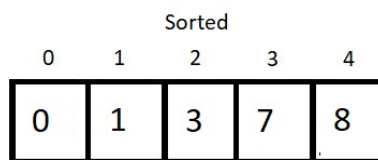
We now have the sorted subarray of length 3, hence the new *min* would be at the index 3. And for the unsorted part of length 2, we would make just a single comparison. So, let's see that.



And since 7 is less than 8, we update our *min* to 4. And since that was the only comparison in this pass, we finish our procedure here by swapping the elements at the indices *min* and 3. And see at the final results:



And since a subarray with a single element is always sorted, we ignore the only unsorted part and make it sorted too.



And this is why the Selection Sort algorithm got its name. We **select** the minimum element at each pass and give it its final position.

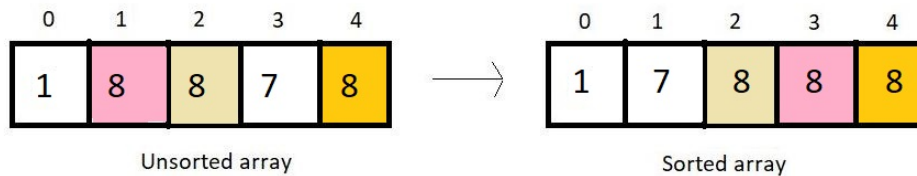
1. Time Complexity of Selection Sort:

We made 4 passes for an array of length 5. Therefore, for an array of length n we would have to make $n-1$ passes. And if you count the number of comparisons we made at each pass, there were $(4+3+2+1)$, that is, a total of 10 comparisons. And every time we compared; we had a fair possibility of updating our *min*. So, 10 comparisons are equivalent to making 10 updates.

So, for length 5, we had $4+3+2+1$ number of comparisons. Therefore, for an array of length n , we would have $(n-1) + (n-2) + (n-3) + (n-4) + \dots + 1$ comparisons.

Sum from 1 to $n-1$, we get , and hence the time complexity of the algorithm would be **$O(n^2)$** .

1. Selection sort algorithm is **not a stable algorithm**. Since the smallest element is replaced with the first element at each pass, it may jumble up positions of equal elements very easily. Hence, unstable. Refer to the example below:



1. It is not a recursive algorithm, since we didn't use recursion here.
2. Selection sort would anyways compare every element with the *min* element, regardless of the fact if the array is sorted or not, hence selection sort is **not an adaptive algorithm** by default.
3. This algorithm offers the benefit of making the least number of swaps to sort an array. We don't make any redundant swaps here.

```

for (int i=0, i < n, i++){
    min = i;
    for (int j = i+1, j < n, j++){
        if ( A[j] < A[min]){
            min = j
        } // end if
    } // end inner for
    swap(i, min)
} // end outer for

```

- Best Case [$O(N^2)$]. And $O(1)$ swaps.
 - Worst Case: Reversely sorted, and when the inner loop makes a maximum comparison. [$O(N^2)$]. Also, $O(N)$ swaps.
-
- Average Case: [$O(N^2)$]. Also $O(N)$ swaps.

Selection Sort vs Insertion Sort vs Bubble sort

Selection sort's advantage is that

- While insertion sort typically makes fewer comparisons than selection sort,
- In bubble sort more swaps than insertion sort.
- Bubble sort is slower than insertion sort.
- But bubble is simple while insertion is complex.
- Insertion sort requires more writes than the selection sort because the inner loop of the insertion sort can require shifting large sections of the sorted portion of the array.
 - In general, insertion sort will write to the array $O(n^2)$ times
 - Whereas selection sort will write/swap only $O(n)$ times
- For this reason, selection sort may be preferable in cases where writing to memory is significantly more expensive than reading,
- such as with EPROM or flash memory

Bubble Sort	Insertion Sort	Selection Sort
$\Theta(n^2)$ comparisons	$\Theta(n^2)$ comparisons	$\Theta(n^2)$ comparisons
$\Theta(n^2)$ swaps	$\Theta(n^2)$ writes	$\Theta(n)$ swaps
Adaptive: $O(n)$ running time when nearly sorted (Best case running time)	Adaptive: $O(n)$ running time when nearly sorted (Best case running time)	Not adaptive $\Theta(n^2)$ running time when nearly sorted (Best case running time)