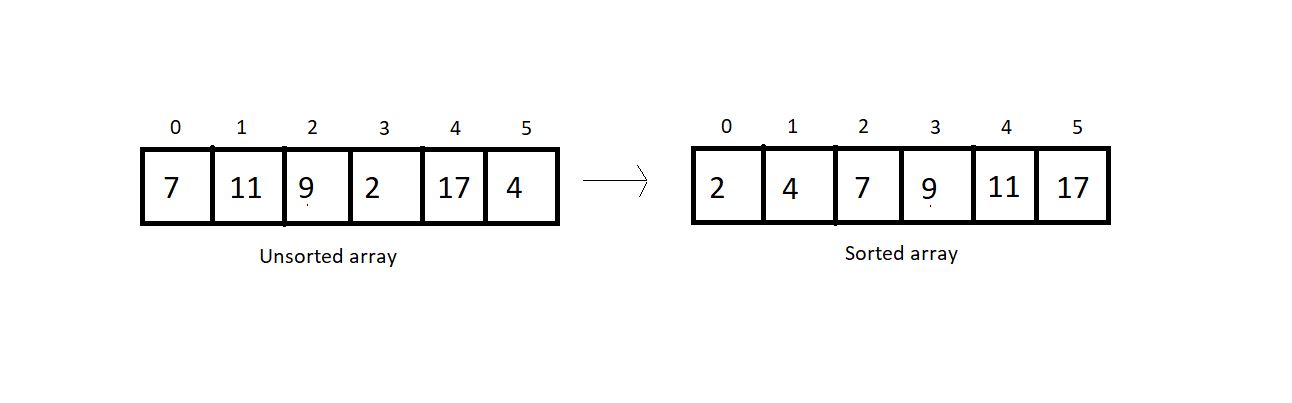
# Bubble Sort Algorithm

In the last tutorial, we discussed different criteria to analyze our sorting algorithms. We made our basis for judging the efficiency of different sorting algorithms for different situations. Today, we are starting all these different sorting algorithms, and we will start with the Bubble Sort Algorithm.

Suppose we are given an array of integers and are asked to sort them using the bubble sort algorithm, then it is not difficult to generate the resultant array, which is just the sorted form of the given array. In fact, whichever algorithm you follow, the result would be the same. The below figure shows the same.



The difference lies in the algorithm we follow. With bubble sort, we intend to ensure that the largest element of the segment reaches the last position at each iteration.  It's important for us to know how that will be pursued.

Bubble sort intends to sort an array using (n-1) passes where n is the array's length. And in one pass, the largest element of the current unsorted part reaches its final position, and our unsorted part of the array reduces by 1, and the sorted part increases by 1. Take a look at the unsorted array above, and I'll walk you through each pass one by one, so you can feel how it gets sorted.

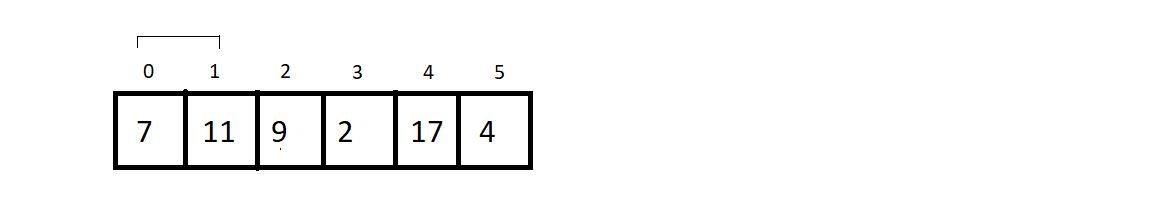
At each pass, we will iterate through the unsorted part of the array and compare every adjacent pair. We move ahead if the adjacent pair is sorted; otherwise, we make it sorted by swapping their positions. And doing this at every pass ensures that the largest element of the unsorted part of the array reaches its final position at the end.

Since our array is of length 6, we will make 5 passes. It wouldn't take long for you to understand why.

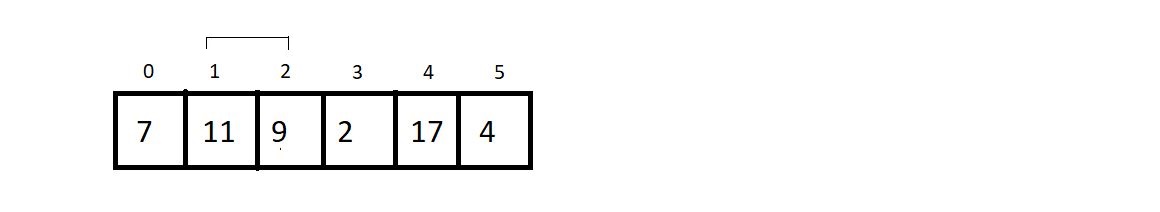
#### 1st Pass:

At first pass, our whole array comes under the unsorted part. We will start by comparing each adjacent pair. Since our array is of length 6, we have 5 pairs to compare.

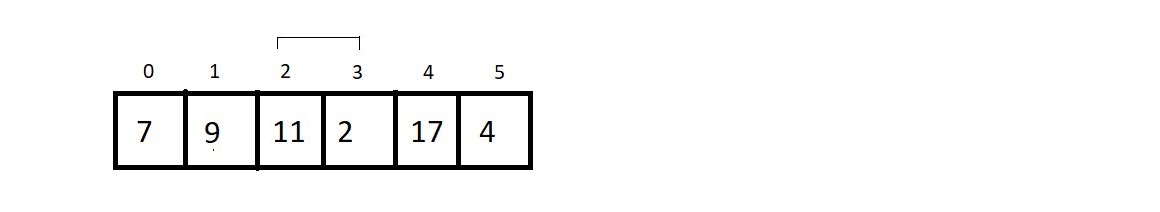
Let’s start with the first one.



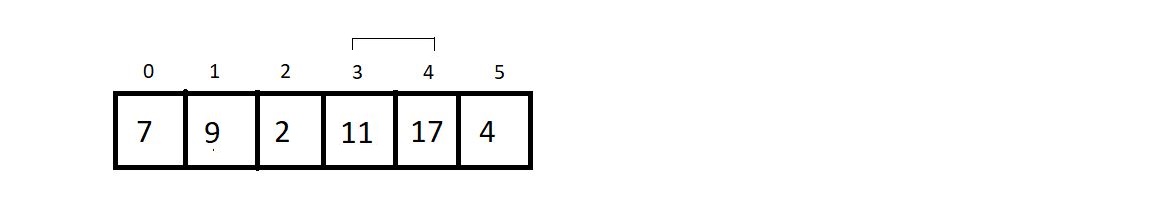
Since these two are already sorted, we move ahead without making any changes.



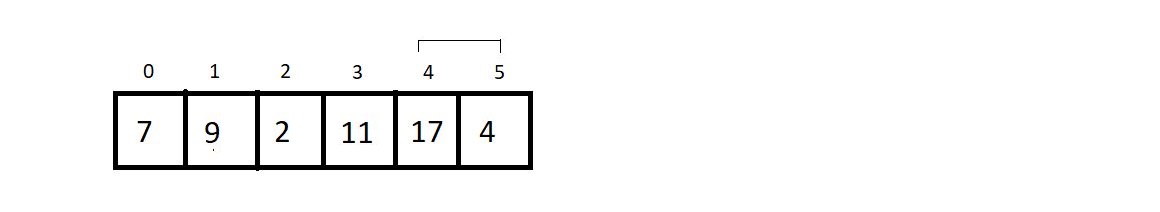
Now since 9 is less than 11, we swap their positions to make them sorted.



Again, we swap the positions of 11 and 2.

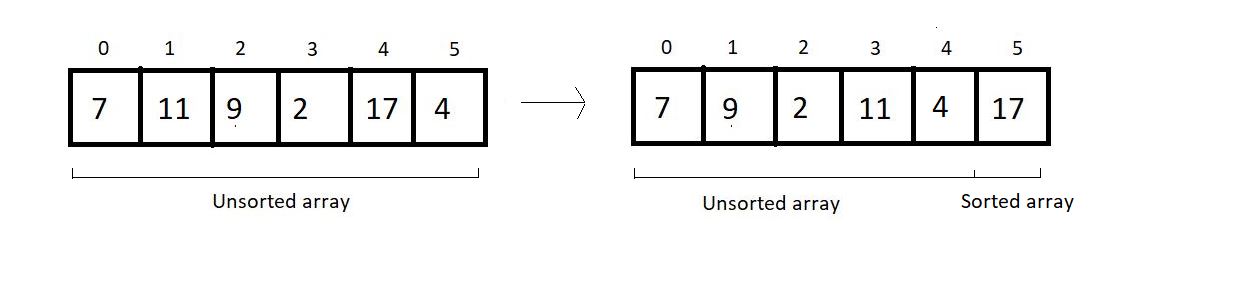


We move ahead without changing anything since they are already sorted.



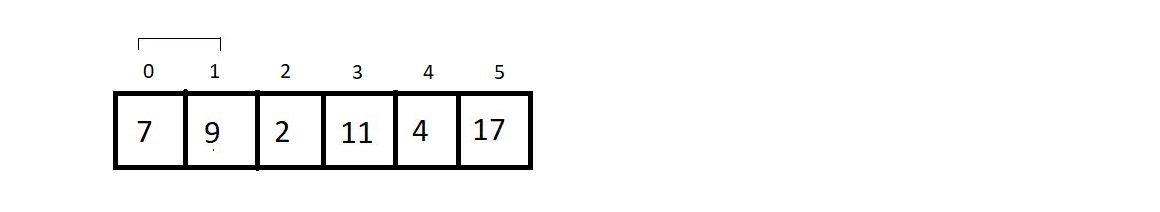
Here, we make a swap since 17 is greater than 4.

And this is where our first pass finishes. We should make an overview of what we received at the end of the first pass.

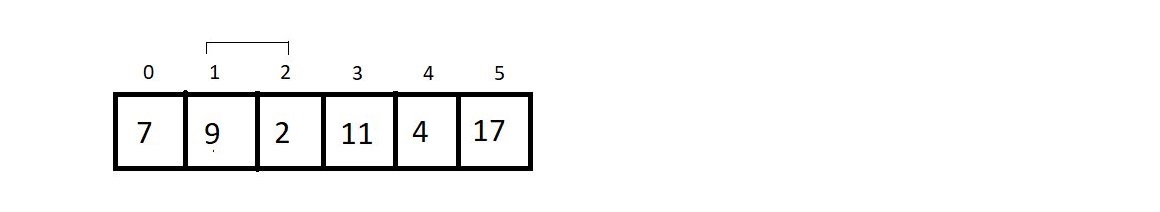


#### 2nd Pass:

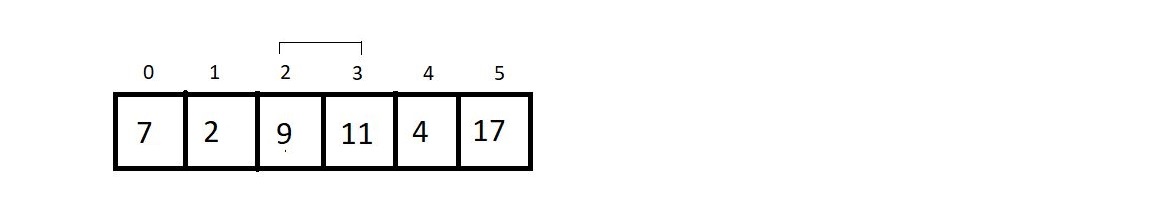
We again start from the beginning, with a reduced unsorted part of length 5. Hence the number of comparisons would be just 4.



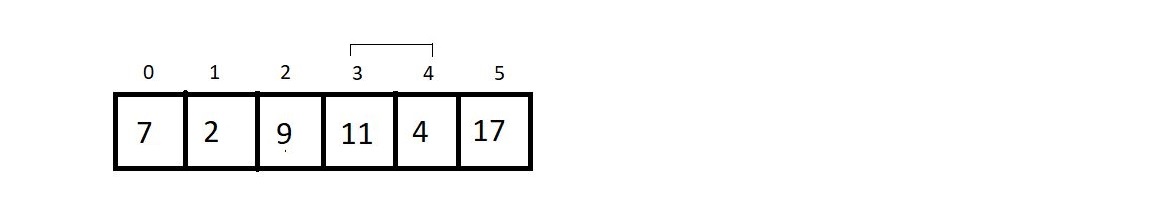
No changes to make.



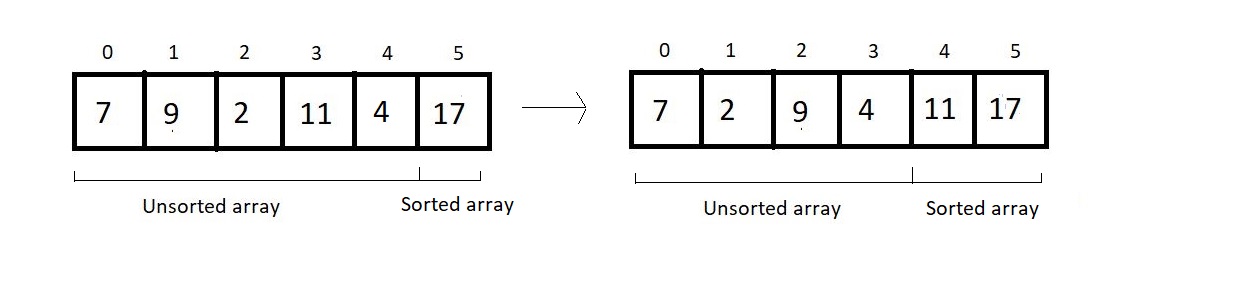
Yes, here we make a swap, since 9>2.



Since 9 < 11, we move further.

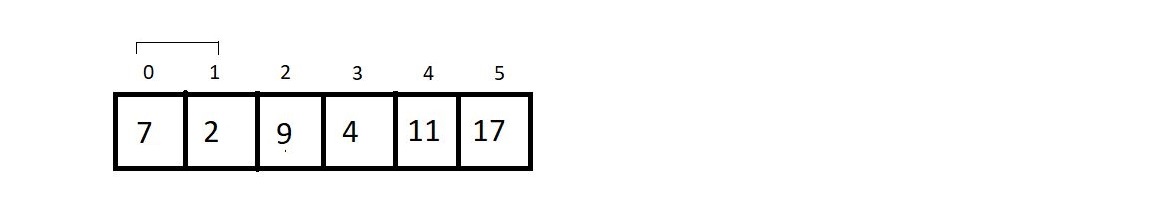


And since 11 is greater than 4, we make a swap again. And that would be it for the second pass. Let’s see how close we have reached to the sorted array.

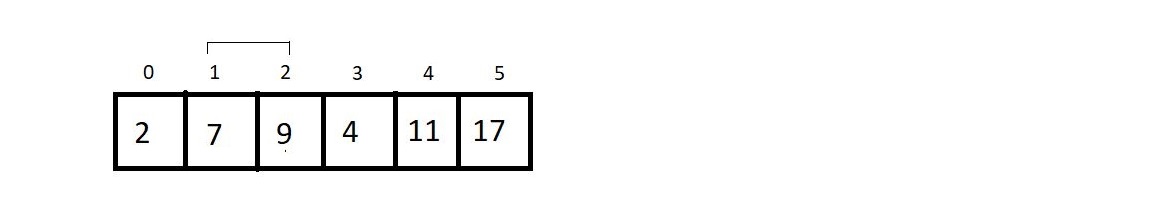


#### 3rd Pass:

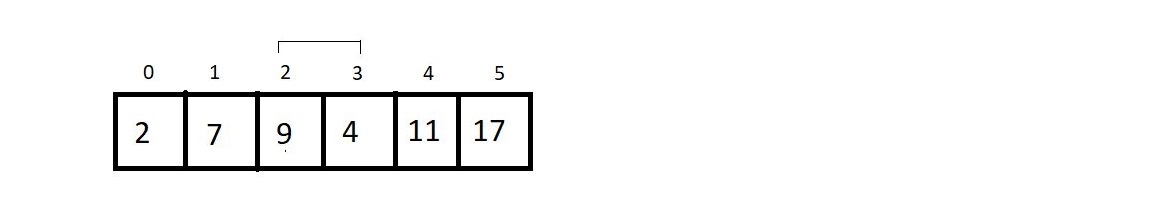
We’ll again start from the beginning, and this time our unsorted part has a length of 4; hence no. of comparisons would be 3.



Since 7 is greater than 2, we make a swap here.

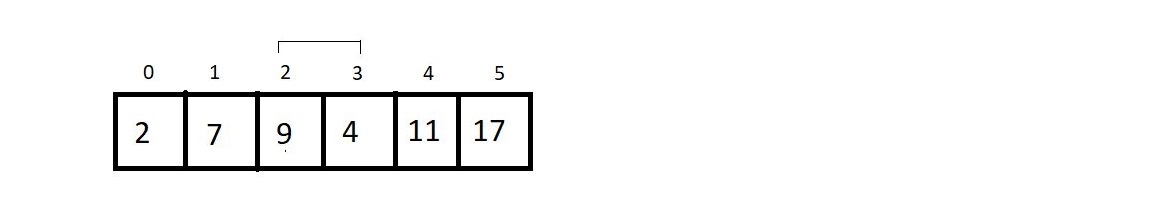


We move ahead without making any change.



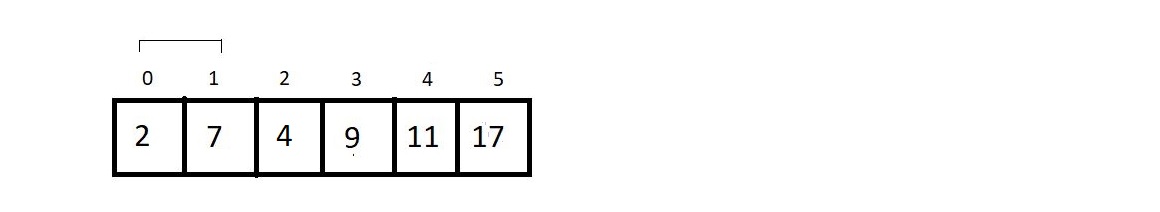
In this final comparison, we make a swap, since 9 > 4.

And that was our third pass. And the result at the end was:

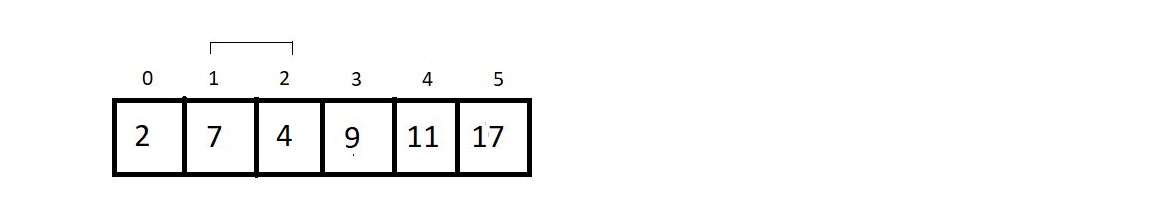


#### 4th Pass:

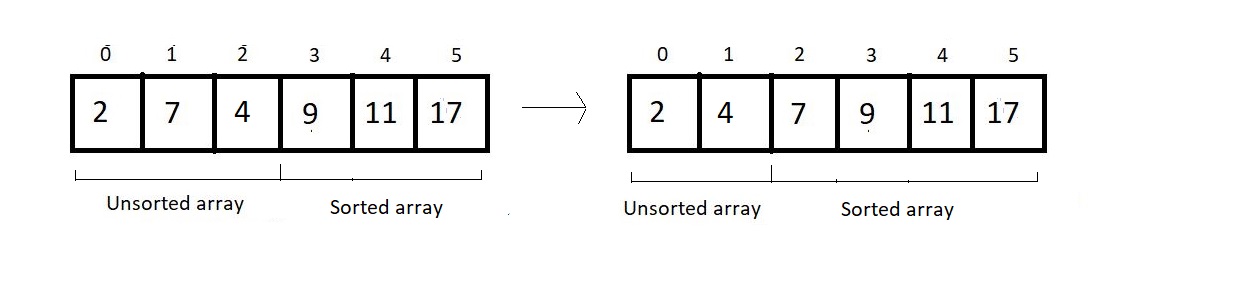
We just have the unsorted part of length 3, and that would cause just 2 comparisons. So, let’s see them.



No changes here.

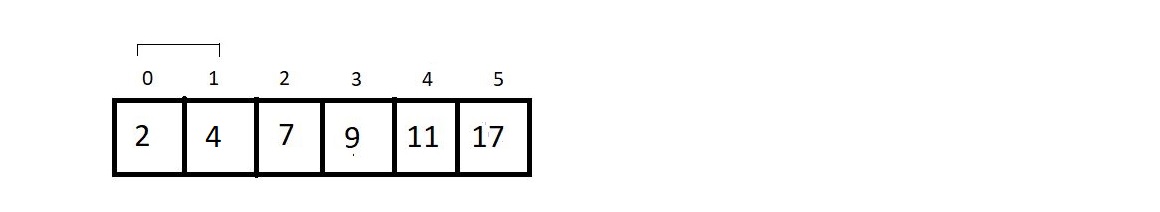


We swap their positions. And that is all in the 4th pass. The resultant array after the 4th pass is:

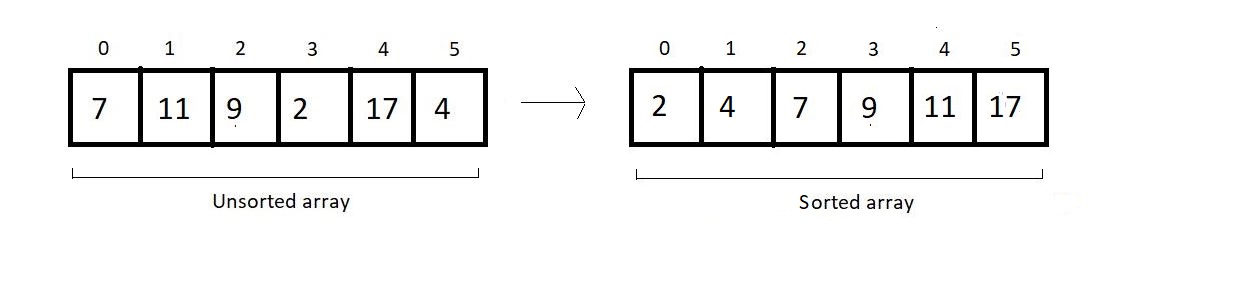


#### 5th (last) pass:

We have only one comparison to make here.



And since these are already sorted, we finish our procedure here. And see the final results:



And this is what the Bubble Sort algorithm looks like. We have a few things to conclude and few calculations regarding the complexity of the algorithm to make.

Time Complexity of Bubble Sort:

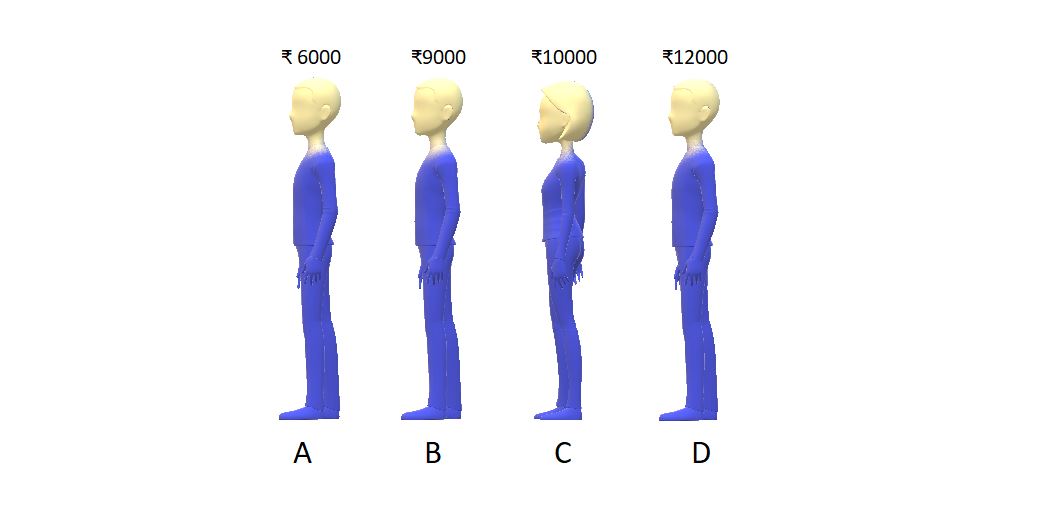
1. If you count the number of comparisons we made, there were (5+4+3+2+1), that is, a total of 15 comparisons. And every time we compared, we had a fair probability of making a swap. So, 15 comparisons intend to make 15 possible swaps.  Let us quickly generalize this sum. For length 6, we had 5+4+3+2+1 number of comparisons and possible swaps. Therefore, for an array of length n, we would have (n-1) + (n-2) + (n-3) + (n-4) + . . . . . + 1 comparison and possible swaps.
2. This is a high school thing to find the sum from 1 to n-1, which is n(n-1)/2, and hence our complexity of runtime becomes O(n^2).
3. And if you could observe, we never made a swap when two elements of a pair become equal. Hence the algorithm is a stable algorithm.
4. It is not a recursive algorithm since we didn’t use recursion here.
5. This algorithm has no adaptive aspect since every pair will be compared, even if the array given has already been sorted. So, no adaptiveness. Although it can be modified to make it adaptive, it's not adaptive by default. We’ll see in the next lecture how it can be made adaptive.

Before we wrap up, bubble sort is called bubble because it bubbles up lighter elements to the left and stores larger elements towards the right. I expect you all to take your own unsorted array and use the bubble sort algorithm to sort it. That would make you confident about using bubble sort.

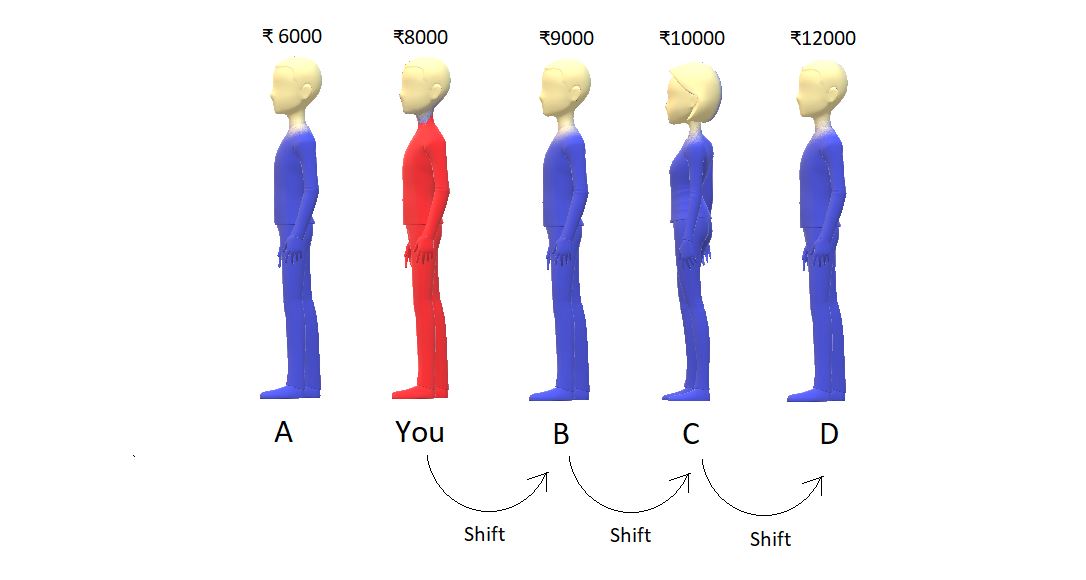
# Insertion Sort Algorithm

In the last lecture, we finished learning the Bubble Sort Algorithm. We learned how to write a program for the same in C language. We saw its characteristics as well. Today, we will learn about a new sorting method, called the Insertion Sort Algorithm. I will make it very intuitive for you to understand. I will use some real-life applications to make the process easy and will steadily dive into the technical part.

Suppose you were to stand in a queue where people are already sorted on the basis of the amount of money they have. Person with the least amount is standing in the front and the person with the largest sum in his pocket stands last. The below illustration describes the given situation.



Problem arises when you suppose you have ₹8000 in your pocket, and you want to be a part of this queue. You don’t know where to stand. So, now you start from the last and keep asking the person standing there whether he has more money than you or less money than you. If you find someone with more money, you simply ask him/her to shift backward. And the moment you find a person having less money than you, you stand just behind him/her. So, after doing all this, you find a position in the 2nd place in the queue. The final situation is:



So, this was one of the examples I had in mind. Now, suppose these were not the people but the numbers in an array. It would have been as simple as it is right now. We would keep comparing two numbers, and if we find a number greater than the number we want to insert, we shift it backward. And the moment we find a number smaller, we insert the element at the vacant space just behind the smaller number.

And basically, what did we learn? We learned to insert an element in a sorted array. Although it felt very intuitive to just put yourself in the second position, what would you do if the queue had a thousand people? Not easy, right? And this is where we need a proper algorithm.

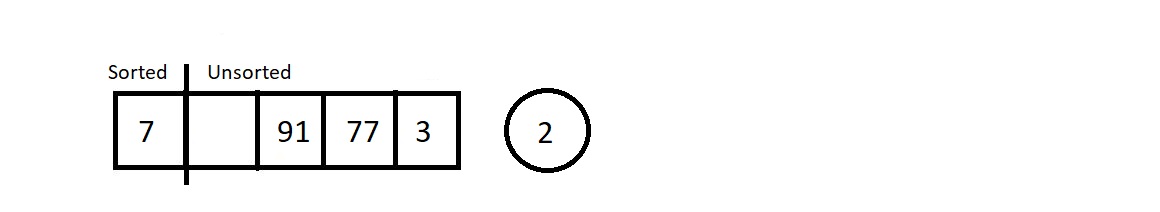
Insert Sort Algorithm:

Let’s just take an array, and use the insertion sort algorithm to sort its elements in increasing order.

Consider the given array below:

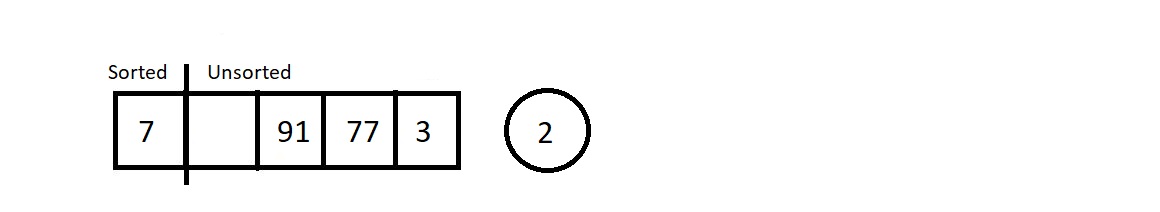


And what have we already learned? We have learned to put an arbitrary element inside a sorted array, using the insertion method we saw above. And an array of a single element is always sorted. So, what we have now is an array of length 5 with a subarray of length 1 already sorted.

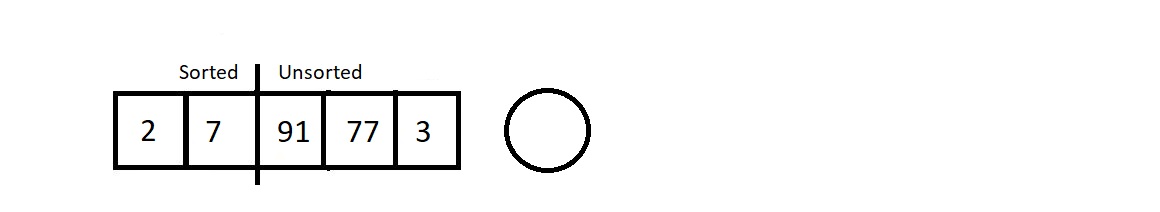


Moving from the left to the right, we will pluck the first element from the unsorted part, and insert it in the sorted subarray. This way at each insertion, our sorted subarray length would increase by 1 and unsorted subarray length decreases by 1. Let’s call each of these insertions and the traversal of the sorted subarray to find the best position, a pass.

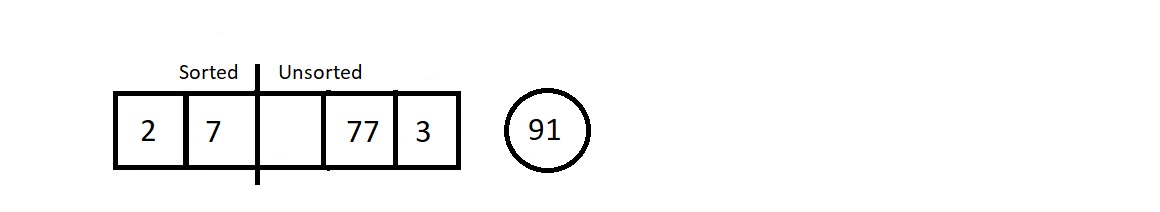
So, let’s start with pass 1, which is to insert 2 in the sorted array of length 1.



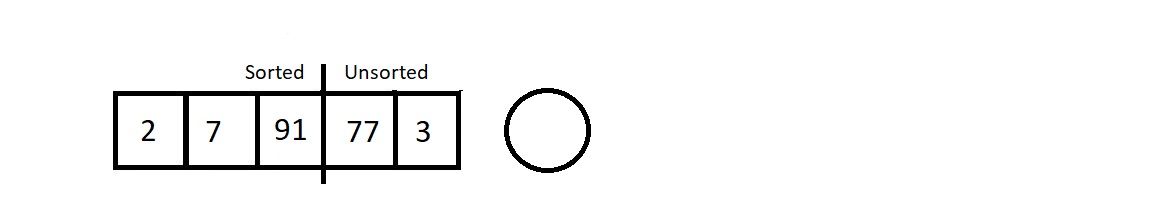
So, we plucked the first element from the unsorted part. Let’s insert element 2 at its correct position, which is before 7. And this increases the size of our sorted array.



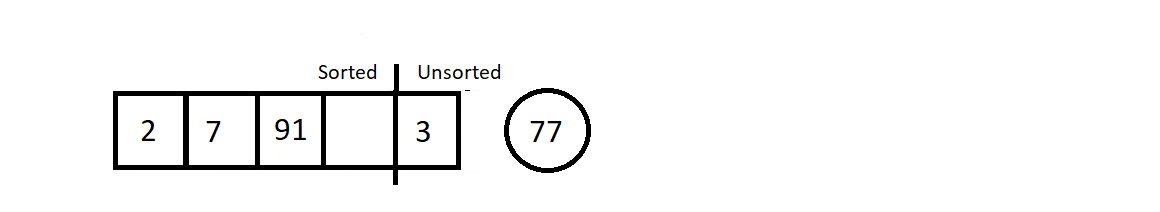
Let’s proceed to the next pass.



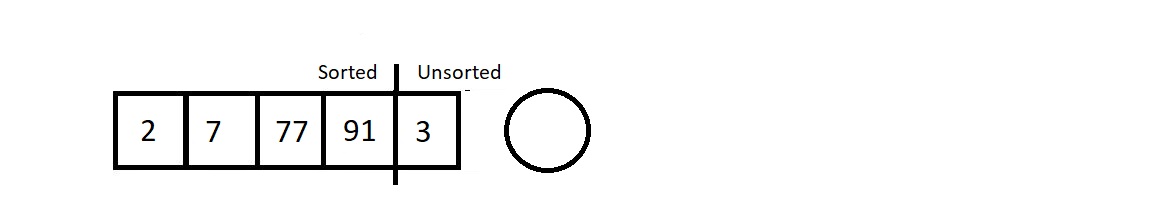
The next element we plucked out was 91. And its position in the sorted array is at the last. So that would cause zero shifting. And our array would look like this.



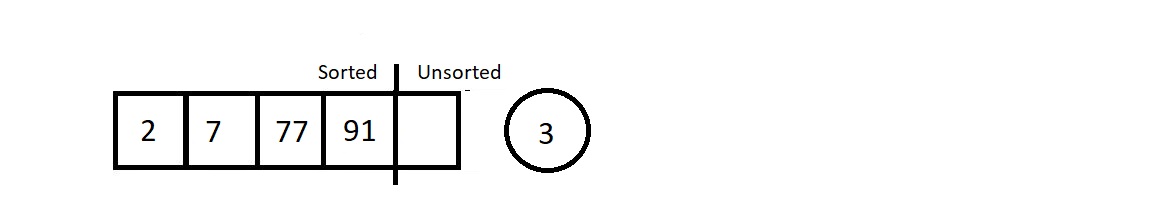
Our sorted subarray now has size 3, and unsorted subarray is now of length 2. Let’s proceed to the next pass which would be to traverse in this sorted array of length 3 and insert element 77.



We started checking its best fit, and found the place next to element 7. So this time it would cause just a single shift of element 91.



As a result, we are left with a single element in the unsorted subarray. Let’s pull that out too in our last pass.



Since our new element to insert is the element 3, we started checking for its position from the back. The position is, no doubt, just next to element 2. So, we shifted elements 7, 77, and 91. Those were the only three shifts.  And the final sorted we received is illustrated below.



So, this was the main procedure behind the insertion sort algorithm.

Analysis:

Conclusively, we had to have 4 passes to sort an array of length 5. And in the first pass, we had to compare the to-be inserted element with just one single element 7. So, only one comparison, and one possible swap. Similarly, for ith pass, we would have i number of comparisons, and i possible swaps.

1. Time Complexity of Insertion Sort Algorithm:

Let’s now calculate the time complexity of the algorithm. We made 4 passes for this array of length 5, and for ith pass, we made i number of comparisons. So, the total number of comparisons is 1+2+3+4. Similarly, for an array of length n, the total number of comparison/possible swaps would be 1+2+3+4+ . . . + (n-1) which is n(n-1)/2, which ultimately is O(n2).

2. Insertion sort algorithm is a stable algorithm, since we start comparing from the back of the sorted subarray, and never cross an element equal to the to be inserted element.

3. Insertion sort algorithm is an adaptive algorithm. When our array is already sorted, we just make (n-1) passes, and don’t make any actual comparison between the elements. Hence, we accomplish the job in O(n).

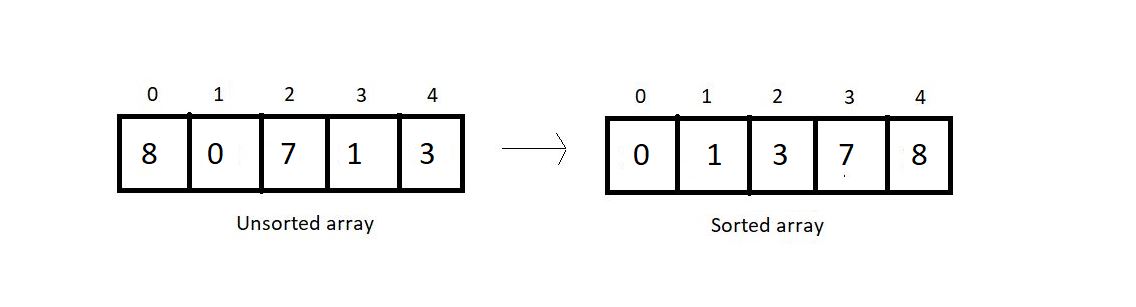
Note: At each pass, we get a sorted subarray at the left, but this intermediate state of the array has no real significance, unlike the bubble sort algorithm where at each pass, we get the largest element having its position fixed at the end.

And that was all about the insertion sort algorithm. I expect you all to take your own unsorted array, and use the insert sort algorithm this time to sort it. Rather, use both bubble sort and insertion sort, and see if it matches, and tell me which one you found convenient. If something seems unclear, go through the lectures again.

# Selection Sort Algorithm

We have already finished learning about two sorting algorithms so far, the bubble sort algorithm and the insertion sort algorithm. In the last tutorial, we implemented the selection sort algorithm in the C language. Today we are interested in learning a new sorting algorithm called the 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.



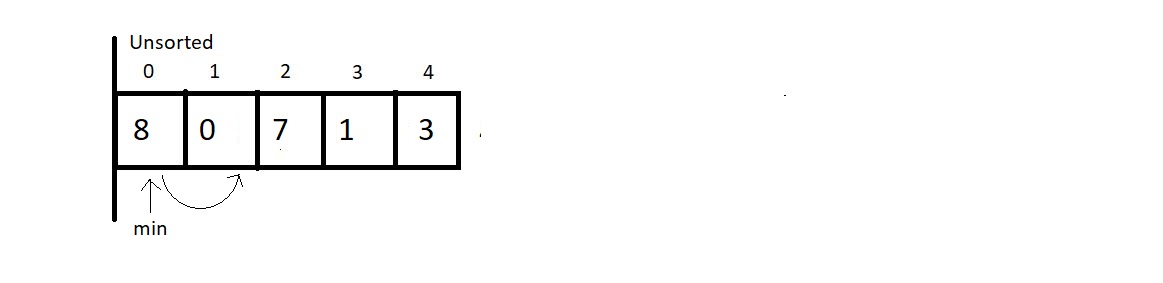
In selection sort, at each pass, we make sure that the smallest element of the current unsorted subarray reaches its final position. And this is pursued by finding the smallest element in the unsorted subarray and replacing it at the end with the element at the first index of the unsorted subarray. This 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 these work. Take a look at the unsorted array above, and I'll walk you through each pass one by one and you will see how we reach the result.

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.

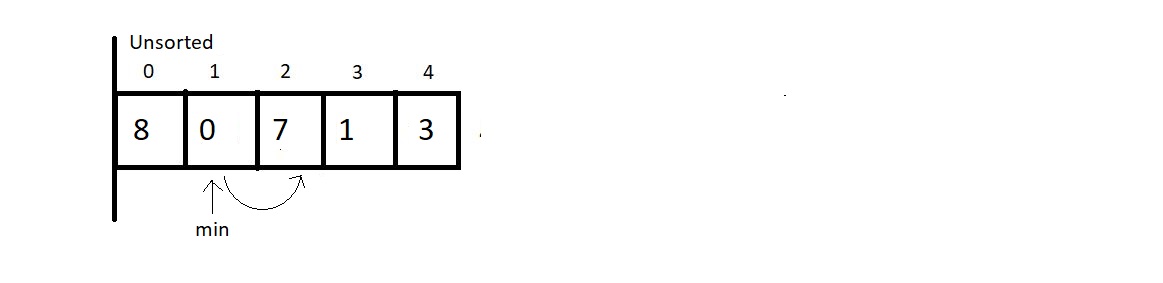
Since our array is of length 5, we will make 4 passes. You must have realized by now the reason why it would take just 4 passes.

1st Pass:

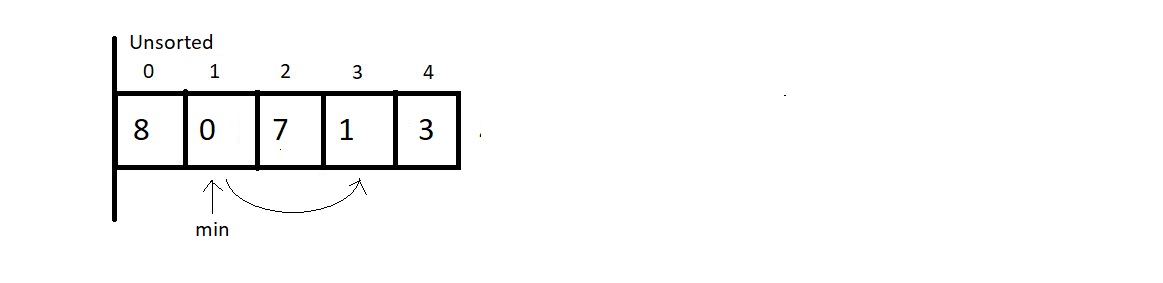
At first pass, our whole array comes under the unsorted part. We will start by assuming 0 as the min index. Now, we’ll have to check among the remaining 4 elements if there is still a lesser element than the first one.



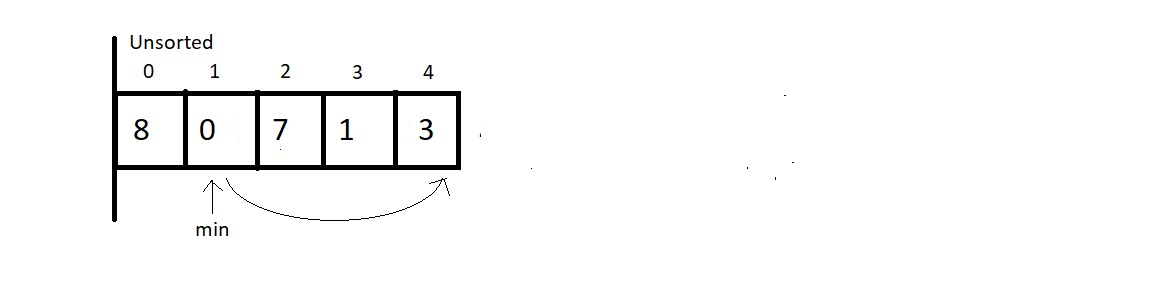
And when we compared the element at min index with the element at index 1, we found that 0 is less than 8 and hence we update our min index to 1.



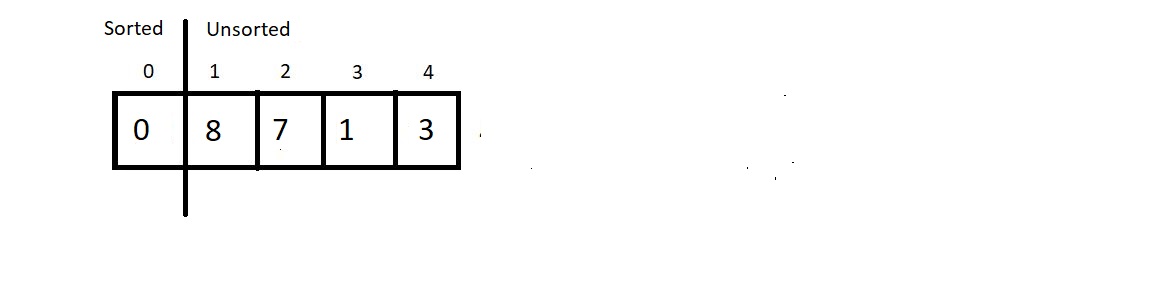
And now we keep checking with the updated min. Since 7 is not less than 0, we move ahead.



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.

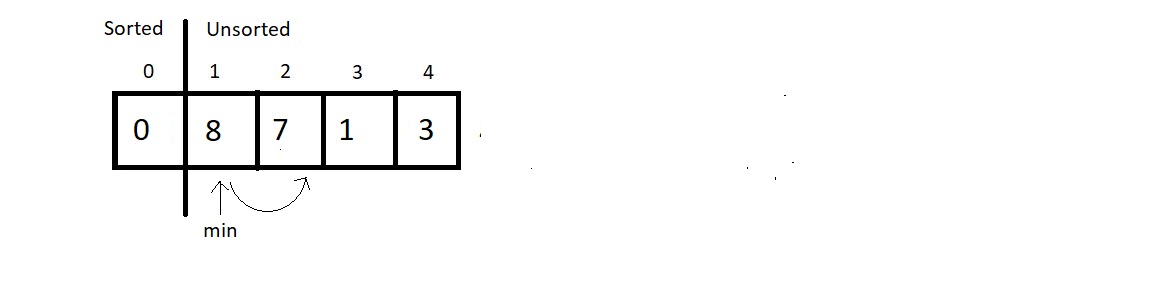


And now we compared the element at the min index with the last element. Since there is nothing to change, we end our 1st pass here. Now we simply replace the element at 0th index with the element at the min index. And this gives us our first sorted subarray of size 1. And this is where our first pass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

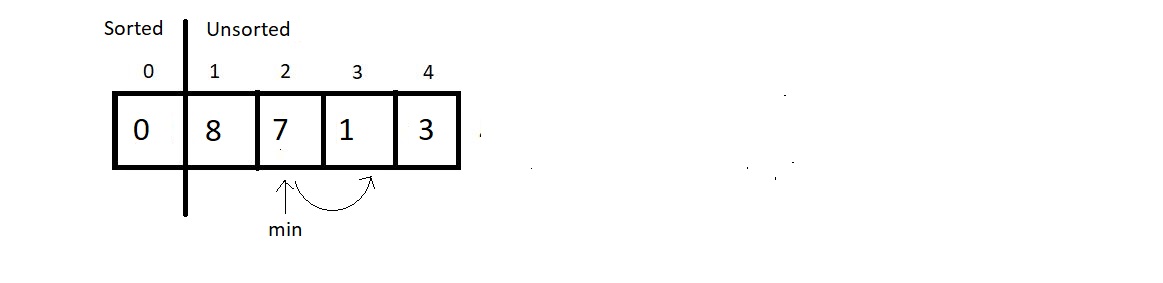


2nd Pass:

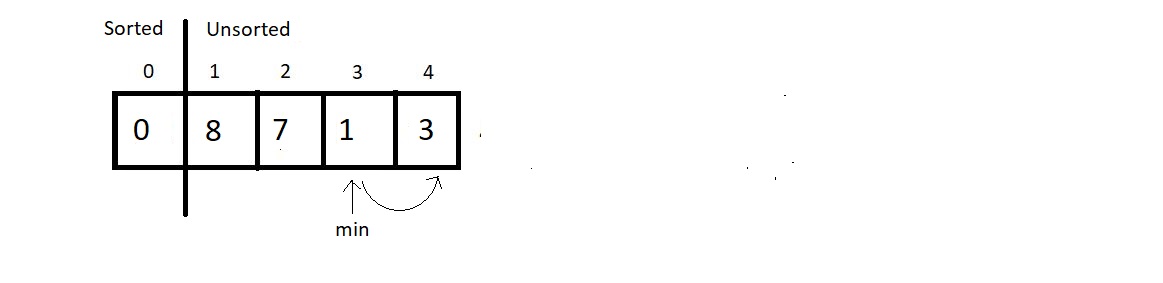
We now start from the beginning of the unsorted array, with a reduced unsorted part of length 4. Hence the number of comparisons would be just 3. We assume the element at index 1 is the one at the min index and start iterating to the right for finding the minimum element.



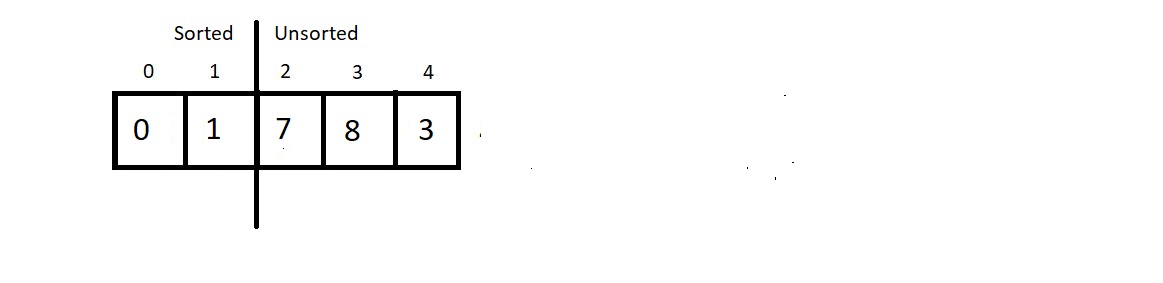
Since 7 is less than 8, we update our min index with 2. And move further.



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.

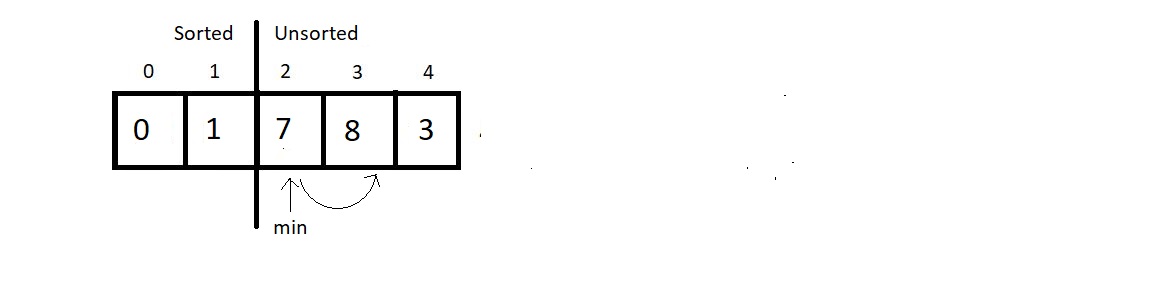


And since 3 is greater than 1, we don’t make any changes here. And since we are finished with the array, we stop our pass here itself, and swap the element at index 1 with this element at min index. And that would be it for the second pass. Let’s see how close we have reached to the sorted array.

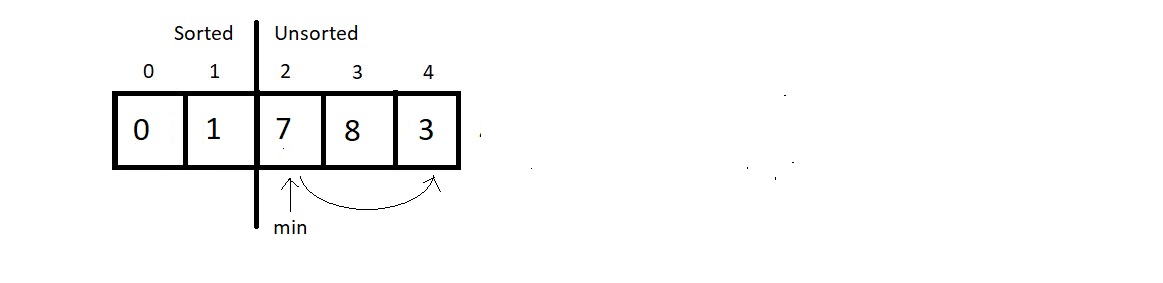


3rd Pass:

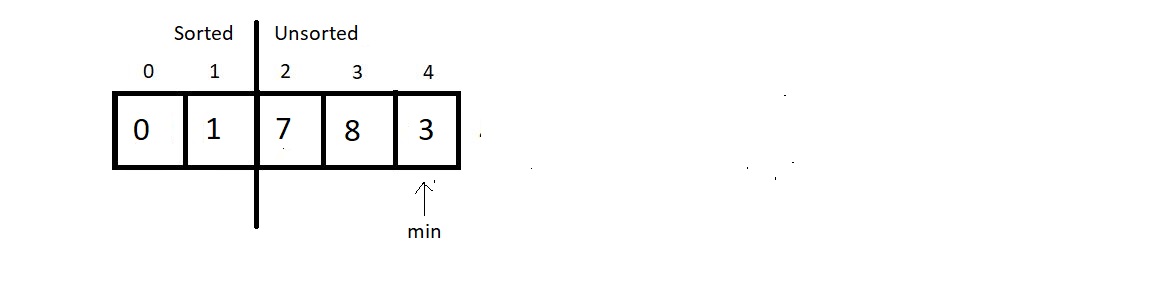
We’ll again start from the beginning of the unsorted subarray which is from the index 2, and make the min index equal to 2 for now. And this time our unsorted part has a length 3, hence no. of comparisons would be 2.



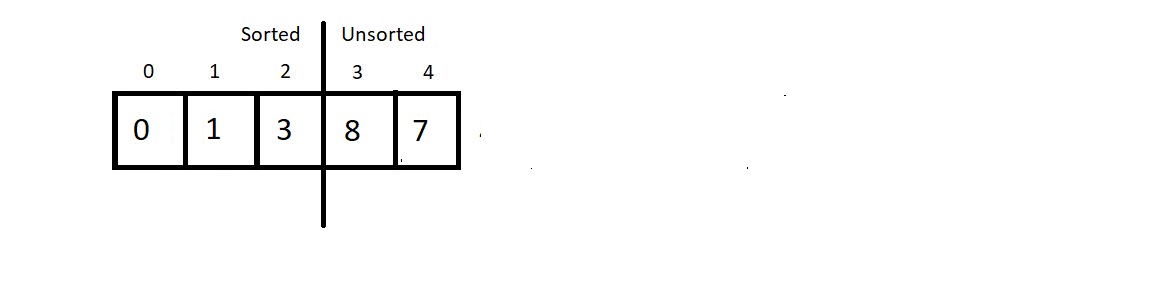
Since 8 is greater than 7, we would make no change, but move ahead.



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.

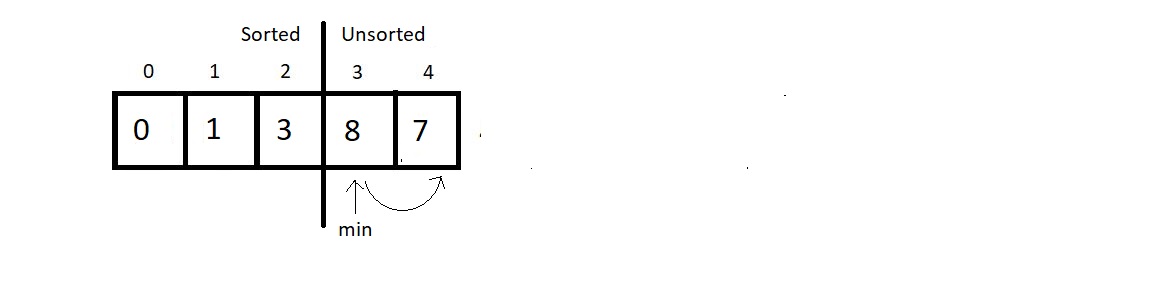


And since that was the last comparison of the third pass, we make a swap of the indices 2 and min. And the result at the end would be:

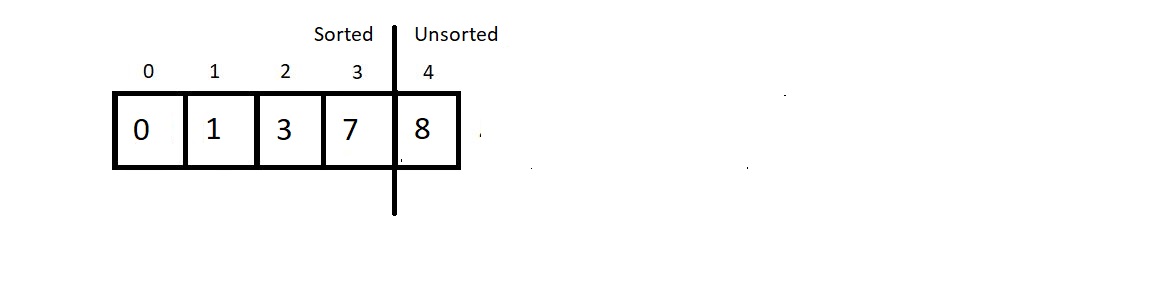


4th Pass:

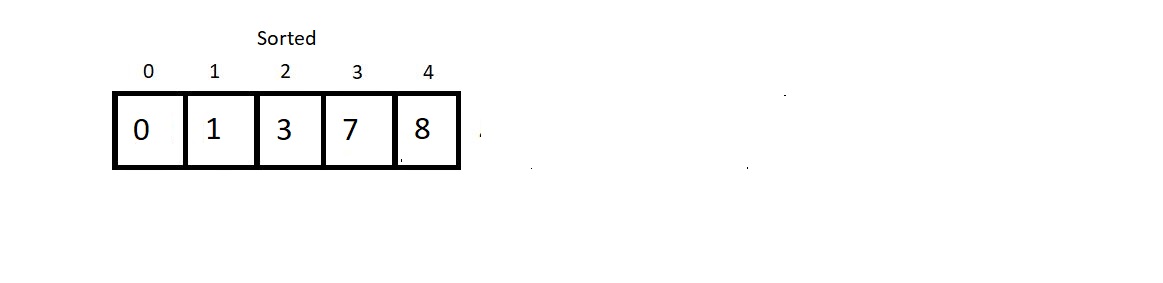
We now have the sorted subarray of length 3, hence the new min would be at the index 3. And for the unsorted part of length 2, we would make just a single comparison. So, let’s see that.



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. Few conclusions before we proceed to the programming segment:

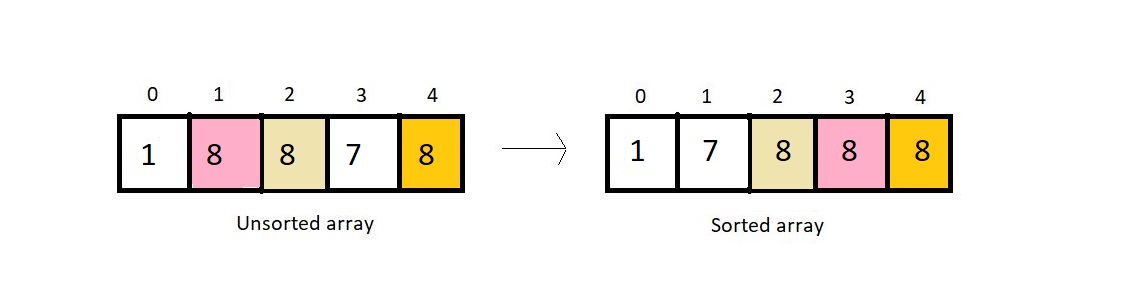
1. Time Complexity of Selection Sort:

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) + . . . . . + 1 comparisons.

Sum from 1 to n-1, we get , and hence the time complexity of the algorithm would be O(n2).

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.

Practice selection sort on your own. Compare each of these algorithms we have learned so far and on various criteria we mentioned. Next would be the programming segment of selection sort.