Sorting Algorithms:

SOIKOT SHAHRIAR

Sorting is a method to arrange a set of elements in either increasing or decreasing order according to some basis or relationship among the elements. Two types of Sorting:

1. Sorting in **Ascending Order**:

Sorting any set of elements in ascending order refers to arranging the elements, let them be numbers, from the smallest to the largest. As example, the set (1, 9, 2, 8, 7), when sorted in ascending order, becomes (1, 2, 7, 8, 9).

2. Sorting in **Descending Order**:

Sorting any set of elements in descending order refers to arranging the elements, let them be numbers, from the largest to the smallest. As example, the same set (1, 9, 2, 8, 7), when sorted in descending order, becomes (9, 8, 7, 2, 1).

Why do we need sorting?

To make you understand the reason why we need sorting in the simplest of ways, we can see some real-life applications of sorting that we might encounter almost daily.

The most useful application is the dictionary. In a dictionary, the words are sorted lexicographically for you to find any word easily.

Criteria for Analysis of Sorting Algorithms:

Time Complexity: Lesser the time complexity, the better is the algorithm.

- We observe the time complexity of an algorithm to see which algorithm works efficiently for larger data sets and which algorithm works faster with smaller data sets. What if one sorting algorithm sorts only 4 elements efficiently and fails to sort 1000 elements. What if it takes too much time to sort a large data set? These are the cases where we say the time complexity of an algorithm is very poor.
- In general, O(N log N) is considered a better algorithm time complexity than O(N 2), and most of our algorithms' time complexity revolves around these two.

Space Complexity:

- The space complexity criterion helps us compare the space the algorithm uses to sort any data set. If an algorithm consumes a lot of space for larger inputs, it is considered a poor algorithm for sorting large data sets. In some

cases, we might prefer a higher space complexity algorithm if it proposes exceptionally low time complexity, but not in general.

- And when we talk about space complexity, the term in-place sorting algorithm arises. The algorithm which results in constant space complexity is called an in-place sorting algorithm. In-place sorting algorithms mostly use swapping and rearranging techniques to sort a data set. One example is Bubble Sort.

Stability:

The stability of an algorithm is judged by the fact whether the order of the elements having equal status when sorted on some basis is preserved or not. It probably sounded technical, but let us explain.

Suppose, we have a set of numbers, 6, 1, 2, 7, 6 and we want to sort them in increasing order by using an algorithm. Then the result would be 1, 2, 6, 6, 7. But the key thing to look at is whether the 6s follow the same order as that given in the input or they have changed. That is, whether the first 6 still comes before the second 6 or not. If they do, then the algorithm we followed is called stable, otherwise unstable.

Adaptivity:

Algorithms that adapt to the fact that if the data are already sorted and it must take less time are called adaptive algorithms. And algorithms which do not adapt to this situation are not adaptive.

Recursiveness:

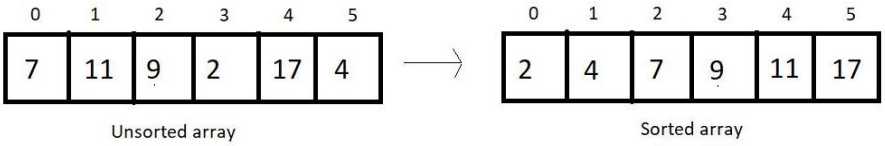
If the algorithm uses recursion to sort a data set, then it is called a recursive algorithm. Otherwise, non-recursive.

Internal & External Sorting Algorithms:

When the algorithm loads the data set into the memory (RAM), we say the algorithm follows internal sorting methods. In contrast, we say it follows the external sorting methods when the data doesn't get loaded into the memory.

Bubble Sort Algorithm

With bubble sort, we intend to ensure that the largest element of the segment reaches the last position at each iteration.

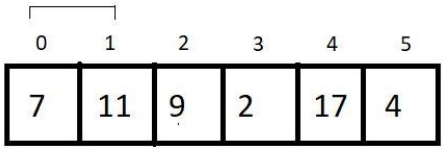


Bubble sort intends to sort an array using (n-1) passes where n is the array's length. And in one pass, the largest element of the current unsorted part reaches its final position, and our unsorted part of the array reduces by 1, and the sorted part increases by 1.

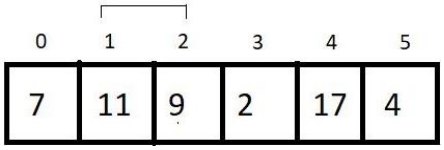
At each pass, we will iterate through the unsorted part of the array and compare every adjacent pair. We move ahead if the adjacent pair is sorted; otherwise, we make it sorted by swapping their positions. And doing this at every pass ensures that the largest element of the unsorted part of the array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

1st Pass:

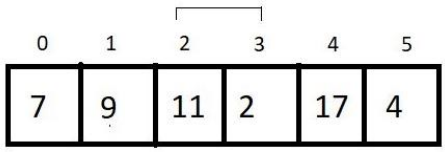
At first pass, our whole array comes under the unsorted part. We will start by comparing each adjacent pair. Since our array is of length 6, we have 5 pairs to compare. Let's start with the first one.



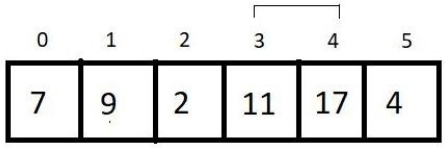
Since these two are already sorted, we move ahead without making any changes.



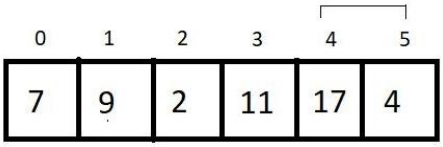
Now since 9 is less than 11, we swap their positions to make them sorted.



Again, we swap the positions of 11 and 2.

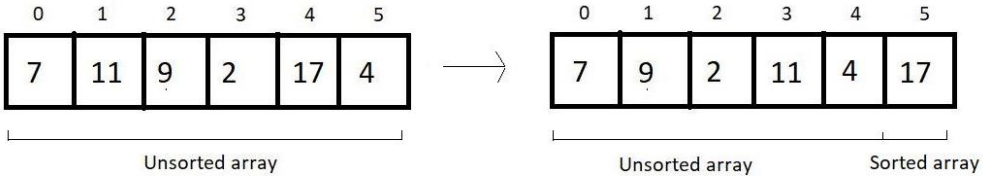


We move ahead without changing anything since they are already sorted.



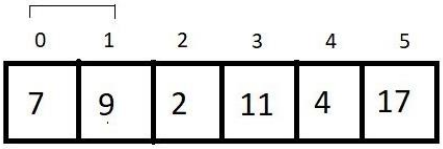
Here, we make a swap since 17 is greater than 4.

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

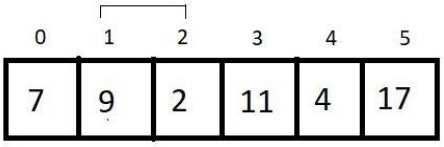


2nd Pass:

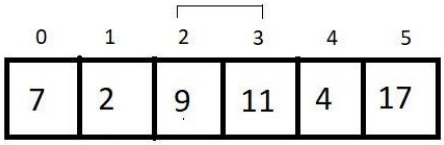
We again start from the beginning, with a reduced unsorted part of length 5. Hence the number of comparisons would be just 4.



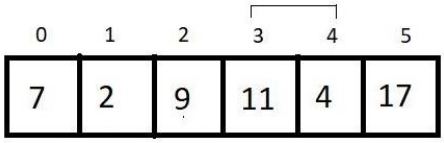
No changes to make.



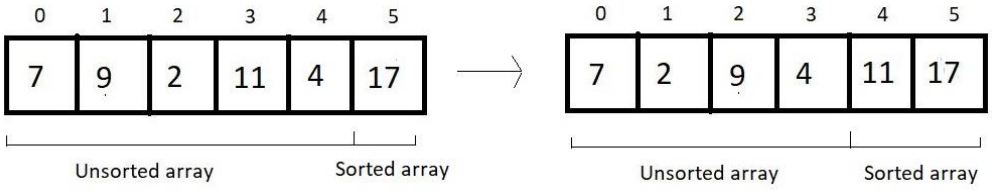
Yes, here we make a swap, since 9>2.



Since 9 < 11, we move further.

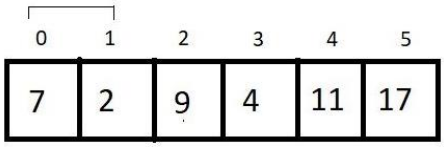


And since 11 is greater than 4, we make a swap again. And that would be it for the second pass. Let's see how close we have reached to the sorted array.

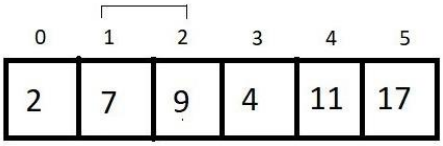


3rd Pass:

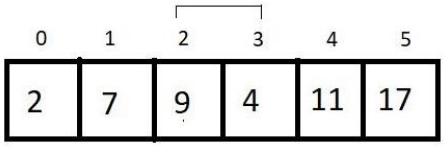
We'll again start from the beginning, and this time our unsorted part has a length of 4; hence no. of comparisons would be 3.



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



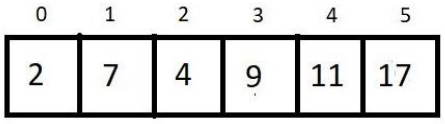
We move ahead without making any change.



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

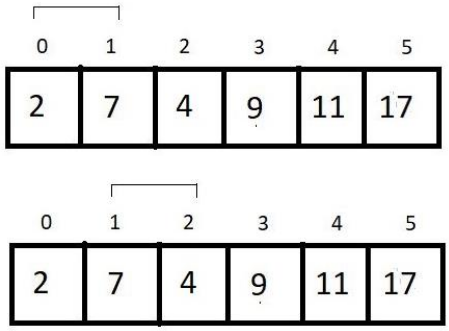
And that was our third pass.

And the result at the end was:

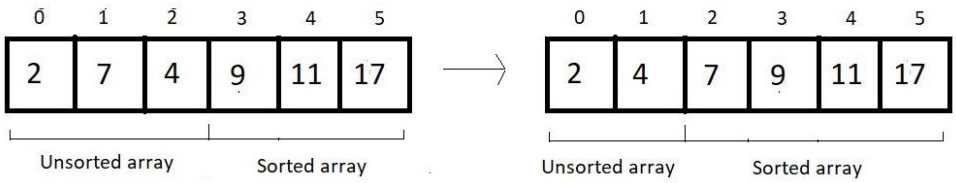


4th Pass:

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

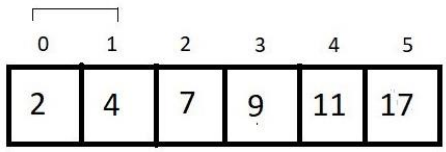


We swap their positions. And that is all in the 4th pass. The resultant array after the 4th pass is:

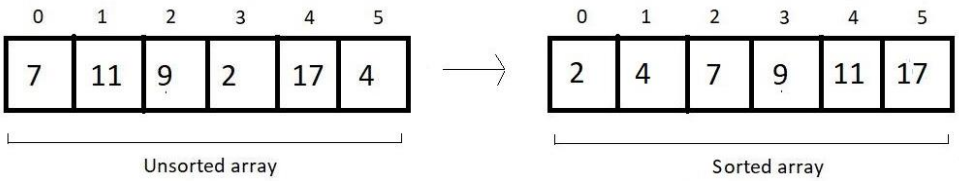


5th (last) Pass:

We have only one comparison to make here.



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



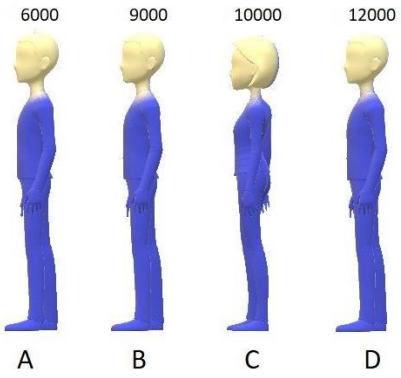
And this is what the Bubble Sort algorithm looks like. We have a few things to conclude and few calculations regarding the complexity of the algorithm to make.

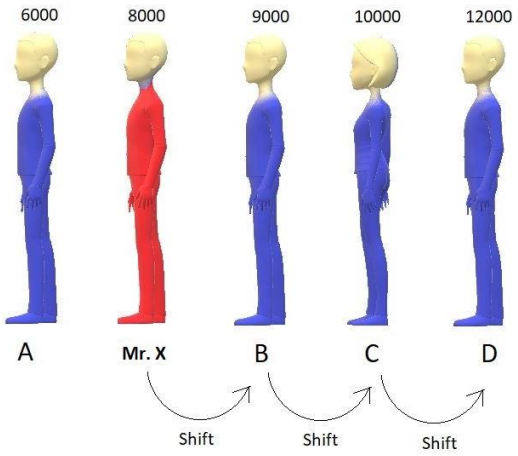
Time Complexity of Bubble Sort:

- If we 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.
- 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).
- And if we could observe, we never made a swap when two elements of a pair become equal. Hence the algorithm is a stable algorithm.
- It is not a recursive algorithm since we didn't use recursion here.
- This algorithm has no adaptive aspect since every pair will be compared, even if the array given has already been sorted. So, no adaptiveness.

Insertion Sort Algorithm

Suppose Mr. X wants 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.

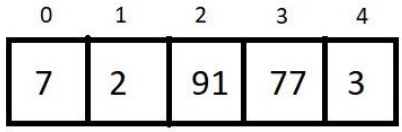




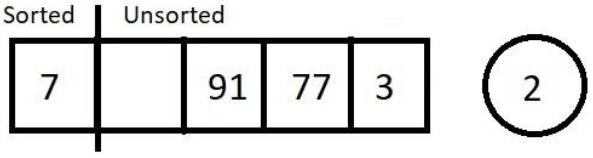
Problem arises when Mr. X suppose he has 8000tk in his pocket, and he wants to be a part of this queue. He doesn't know where to stand. So, now he starts from the last and keep asking the person standing there whether he has more money than him or less money than him. If he finds someone with more money, simply he ask him/her to shift backward. And the moment he finds a person having less money than him, he stands just behind him/her. So, after doing all this, Mr. X finds a position in the 2nd place in the queue.

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.

Insertion Sort Algorithm: Let's just take an array, and use the insertion sort algorithm to sort its elements in increasing order.

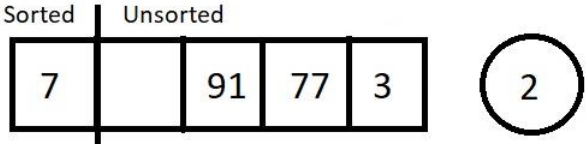


We have learned to put an arbitrary element inside a sorted array, using the insertion method we saw above. And an array of a single element is always sorted. So, what we have now is an array of length 5 with a subarray of length 1 already sorted.

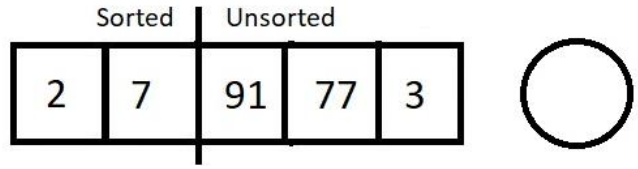


Moving from the left to the right, we will pluck the first element from the unsorted part, and insert it in the sorted subarray. This way at each insertion, our sorted subarray length would increase by 1 and unsorted subarray length decreases by 1. Let's call each of these insertions and the traversal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

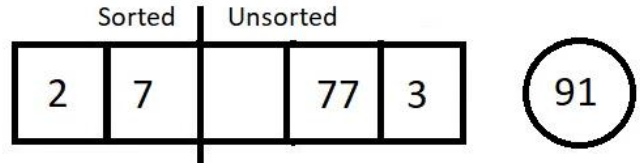
So, let's start with pass 1, which is to insert 2 in the sorted array of length 1.



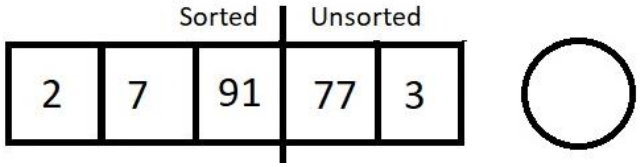
So, we plucked the first element from the unsorted part. Let's insert element 2 at its correct position, which is before 7. And this increases the size of our sorted array.



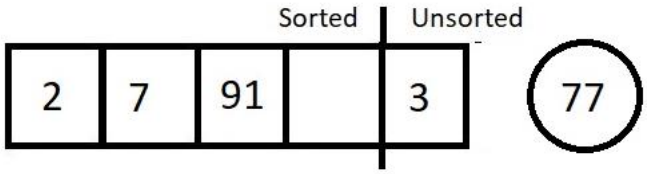
Let's proceed to the next pass.



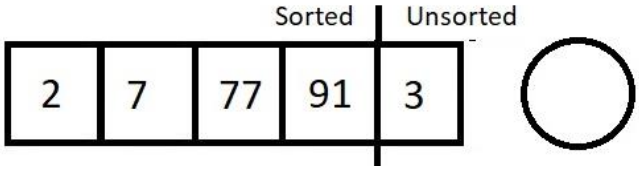
The next element we plucked out was 91. And its position in the sorted array is at the last. So that would cause zero shifting. And our array would look like this.



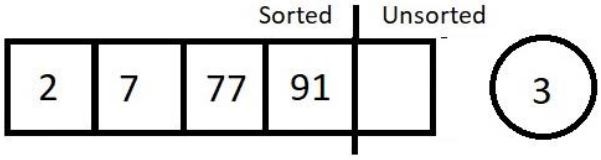
Our sorted subarray now has size 3, and unsorted subarray is now of length 2. Let's proceed to the next pass which would be to traverse in this sorted array of length 3 and insert element 77.



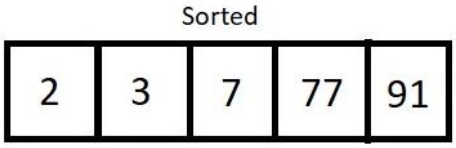
We started checking its best fit, and found the place next to element 7. So, this time it would cause just a single shift of element 91.



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 i th pass, we would have i number of comparisons, and i possible swaps.

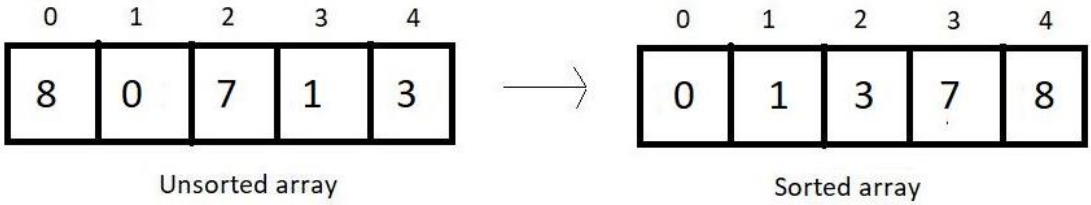
Time Complexity of Insertion Sort Algorithm:

- We made 4 passes for this array of length 5, and for I th 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).
- 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.
- 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.

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.



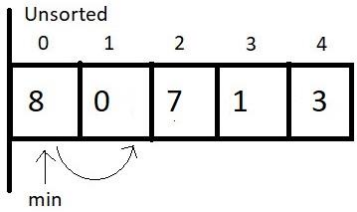
In selection sort, at each pass, we make sure that the smallest element of the current unsorted subarray reaches its final position. And this is pursued by finding the smallest element in the unsorted subarray and replacing it at the end with the element at the first index of the unsorted subarray. This 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 we'll walk through each pass one by one and 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.

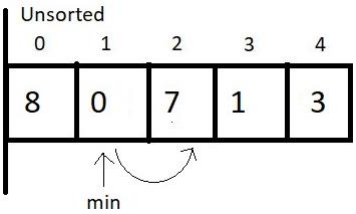
Since our array is of length 5, we will make 4 passes. We must have realized by now the reason why it would take just 4 passes.

1st Pass:

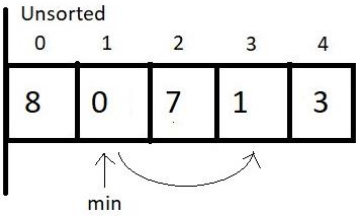
At first pass, our whole array comes under the unsorted part. We will start by assuming 0 as the **min** index. Now, we'll have to check among the remaining 4 elements if there is still a lesser element than the first one.



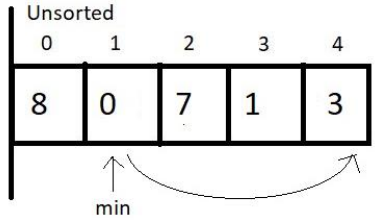
And when we compared the element at min index with the element at index 1, we found that 0 is less than 8 and hence we update our min index to 1.



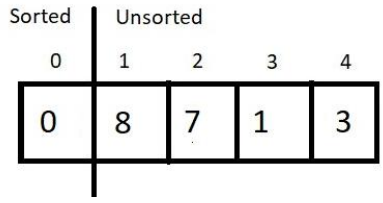
And now we keep checking with the updated min. Since 7 is not less than 0, we move ahead.



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.

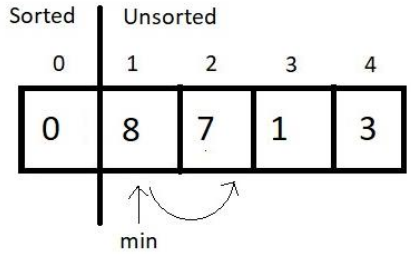


And now we compared the element at the min index with the last element. Since there is nothing to change, we end our 1st pass here. Now we simply replace the element at 0th index with the element at the min index. And this gives us our first sorted subarray of size 1. And this is where our first pass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

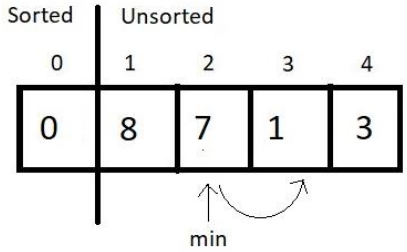


2nd Pass:

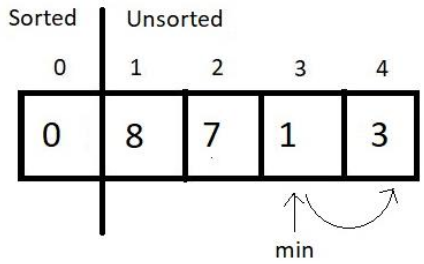
We now start from the beginning of the unsorted array, with a reduced unsorted part of length 4. Hence the number of comparisons would be just 3. We assume the element at index 1 is the one at the min index and start iterating to the right for finding the minimum element.



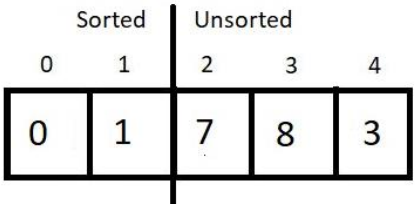
Since 7 is less than 8, we update our min index with 2. And move further.



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.

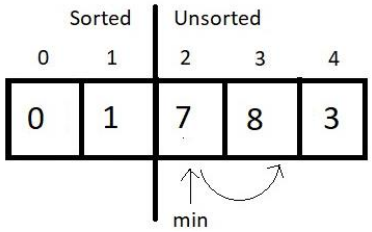


And since 3 is greater than 1, we don't make any changes here. And since we are finished with the array, we stop our pass here itself, and swap the element at index 1 with this element at min index. And that would be it for the second pass. Let's see how close we have reached to the sorted array.

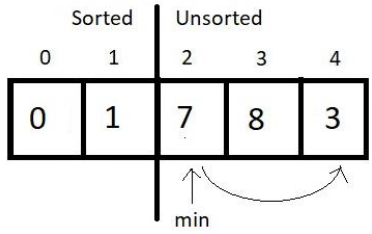


3rd Pass:

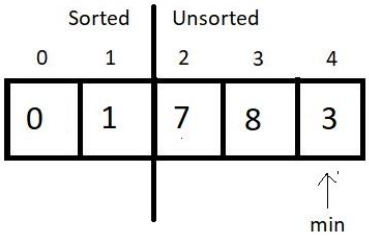
We'll again start from the beginning of the unsorted subarray which is from the index 2, and make the min index equal to 2 for now. And this time our unsorted part has a length 3, hence no. of comparisons would be 2.



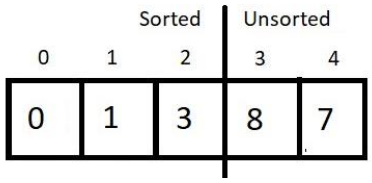
Since 8 is greater than 7, we would make no change, but move ahead.



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.

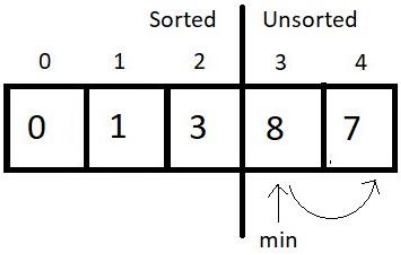


And since that was the last comparison of the third pass, we make a swap of the indices 2 and min. And the result at the end would be:

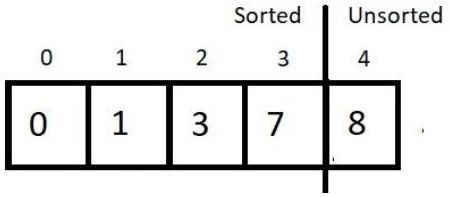


4th Pass:

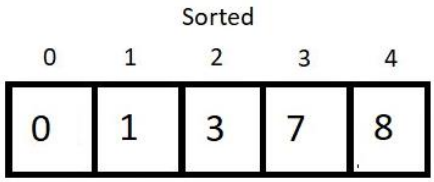
We now have the sorted subarray of length 3, hence the new min would be at the index 3. And for the unsorted part of length 2, we would make just a single comparison. So, let's see that.



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.



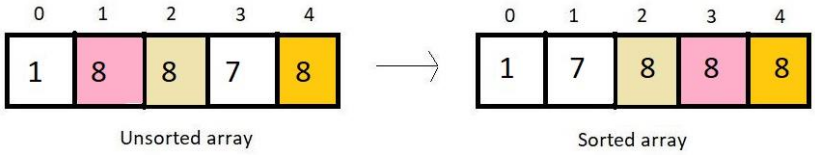
And this is why the Selection Sort algorithm got its name. We select the minimum element at each pass and give it its final position. Few conclusions before we proceed to the programming segment:

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).

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:



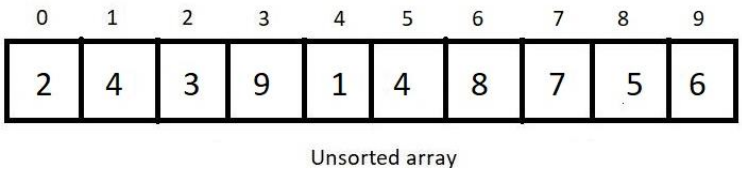
Selection sort would anyways compare every element with the min element, regardless of the fact if the array is sorted or not, hence selection sort is **not an adaptive algorithm** by default.

This algorithm offers the benefit of making the least number of swaps to sort an array. We don't make any redundant swaps here.

Quick Sort Algorithm

The Quick Sort algorithm is quite different from the ones we have studied so far. Here, we use the divide and conquer method to sort our array in pieces reducing our effort and space complexity of the algorithm. There are two new concepts we must know before you jump into the core. First is the **divide and conquer method**. As the name suggests, Divide and Conquer divides a problem into subproblems and solves them at their levels, giving the output as a result of all these subproblems. The second is the **partition method** in sorting. In the partition method, we choose an element as a pivot and try pushing all the elements smaller than the pivot element to its left and all the greater elements to its right. We thus finalize the position of the pivot element. Quick Sort is implemented using both these concepts

Suppose we are given an array of integers, and we are asked to sort them using the quicksort algorithm, then the very first task we would do is to choose a pivot. Pivots are chosen in various ways, but for now, we'll consider the first element of every unsorted subarray as the pivot. Remember this while we proceed.



In the quicksort algorithm, every time we get a fresh unsorted subarray, we do a partition on it. Partition asks you to first choose an element as a pivot. And as already decided, we would choose the first element of the unsorted subarray as the pivot. We would need two more index variables, i and j. Below enlisted is

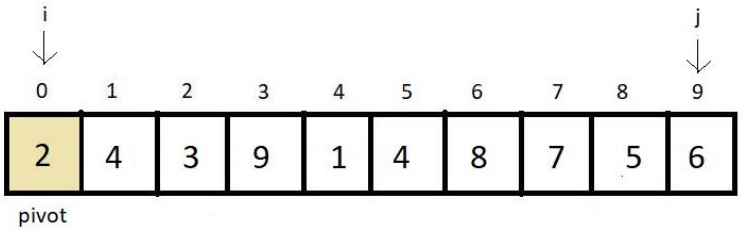
the flow of our partition algorithm we must adhere to. We always start from step 1 with each fresh partition call.

a. Define i as the low index, which is the index of the first element of the subarray, and j as the high index, which is the index of the last element of the subarray.
b. Set the pivot as the element at the low index i since that is the first index of the unsorted subarray.
c. Increase i by 1 until you reach an element greater than the pivot element.
d. Decrease j by 1 until you reach an element smaller than or equal to the pivot element.
e. Having fixed the values of i and j, interchange the elements at indices i and j.
f. Repeat steps 3, 4, and 5 until j becomes less than or equal to i.
g. Finally, swap the pivot element and the element at the index.

This was the **partitioning algorithm**. Every time we call a partition, the pivot element gets its final position. A partition never guarantees a sorted array, but it does guarantee that all the smaller elements are located to the pivot's left, and all the greater elements are located to the pivot's right.

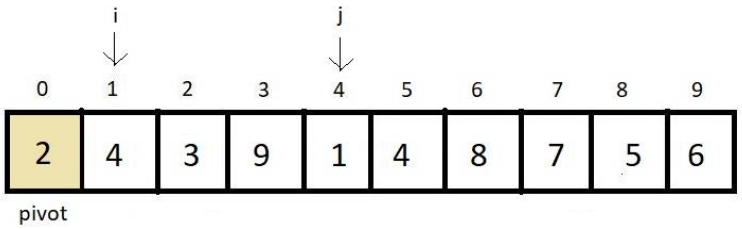
Now let's look at how the array we received at the beginning gets sorted using partitioning and divide and conquer recursively for smaller subarrays.

Firstly, the whole array is unsorted, and hence we apply quicksort on the whole array. Now, we apply a partition in this array. Applying partition asks us to follow all the above steps we discussed.

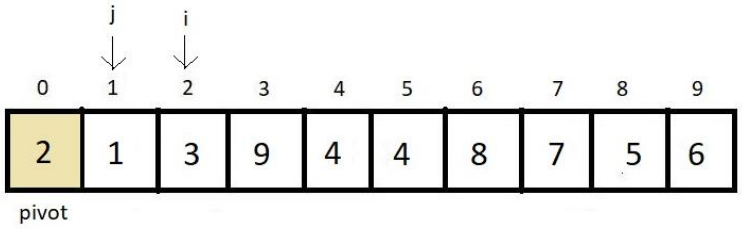


Current Unsorted Subarray

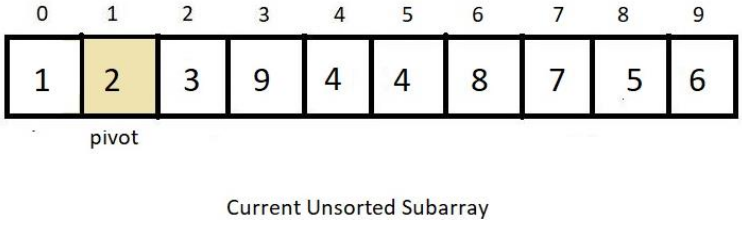
Keep increasing i until we reach an element greater than the pivot, and keep decreasing j until we reach an element smaller or equal to the pivot.



Swap the two elements and continue the search further until j crosses i or becomes equal to i.

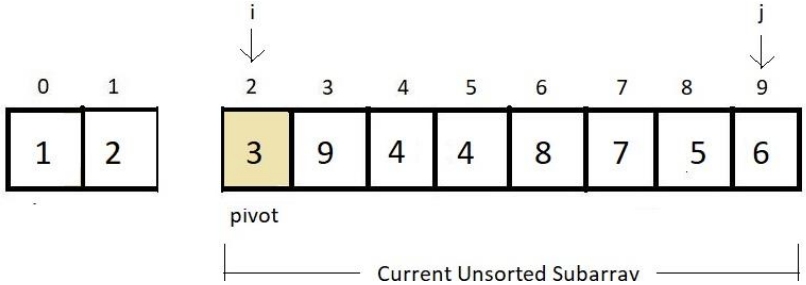


As j crossed i while searching, we followed the final step of swapping the pivot element and the element at j.

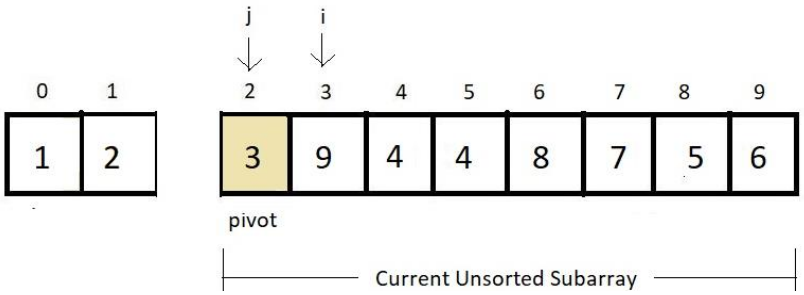


And this would be the final position of the current pivot even in our sorted array. As we can see, all the elements smaller than 2 are on the left, and the rest greater than 2 are on the right. Here comes the role of divide and conquer. We separate our focus from the whole array to just the subarrays, which are not sorted yet. Here, we have subarrays {1} and {3, 9, 4, 4, 8, 7, 5, 6} unsorted. So, we make a call to quicksort on these two subarrays.

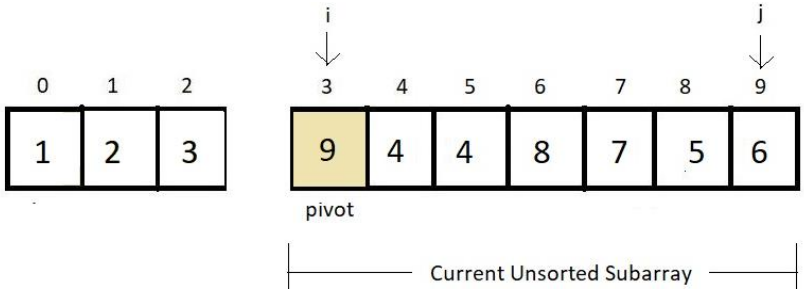
Now since the first subarray has just a single element, we consider it sorted. Let's now sort the second subarray. Follow all the partition steps from the beginning.



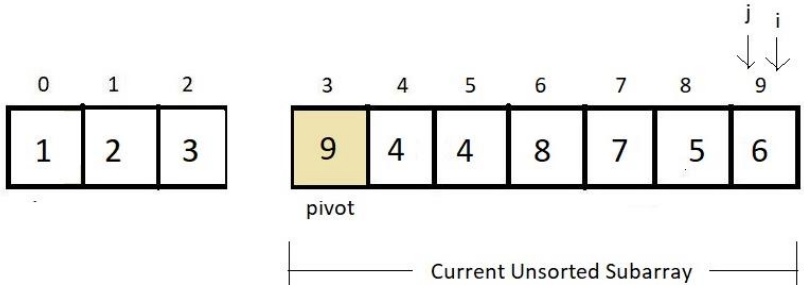
Now, our new pivot is the element at index 2. And i and j are 2 and 9, respectively, marking the start and the end of the subarray. Follow steps 3 and 4.



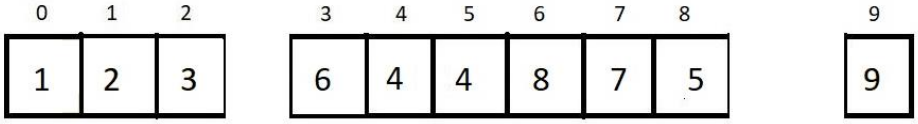
And since there were no elements smaller than 3, j crosses i in the very first iteration. This means 3 was already at its sorted index. And there are no elements to its left; the only unsorted subarray is {9, 4, 4, 8, 7, 5, 6}. And our new situation becomes:



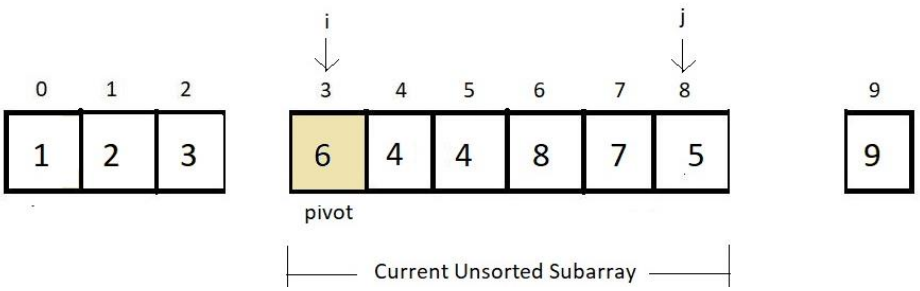
Repeating steps 3 and 4.



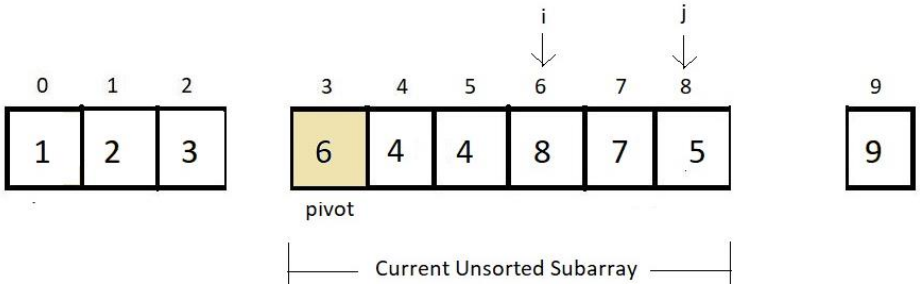
We found that there was no element greater than 9, and hence i reached the last. And 6 was the first element j found to be smaller than 9, and they collided. And this is where we do step 7. Swap the pivot element and the element at index j.



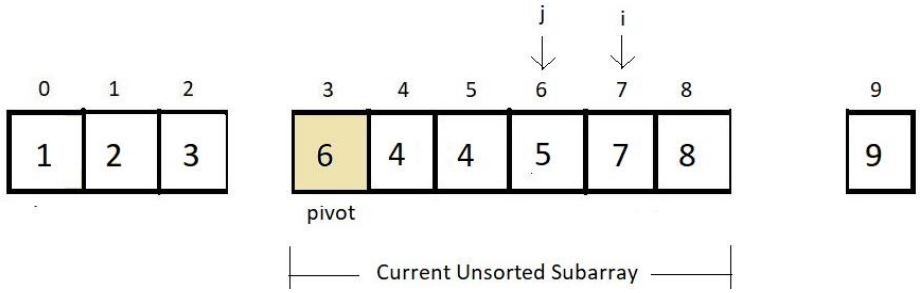
And since there are no elements to the right of element 9, we just have one sorted subarray {6, 4, 4, 8, 7, 5}. Let's call a partition on this as well.



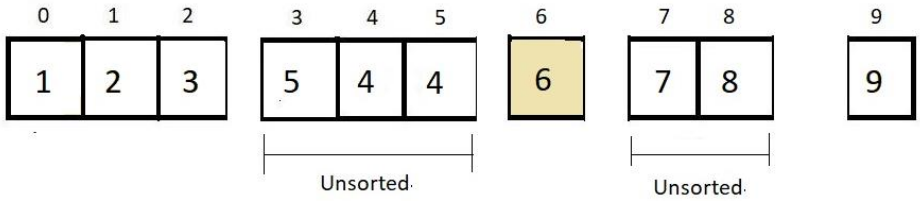
Repeat steps 3 and 4.



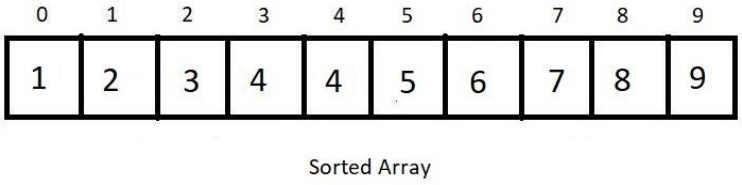
And since the condition j<=i has not been met yet, we just swap the elements at index i and j and continue our search.



And now i and j crossed each other, and now only we swap our pivot element and element at j.



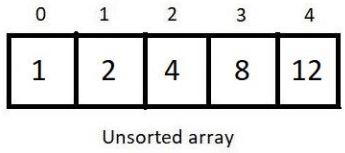
And now, we again divide and consider only the elements that remain unsorted to the left of the pivot and the right of the pivot. And moving things further would just waste our time. We can assume that things move as expected, and it will get sorted at the end and would look something like this.



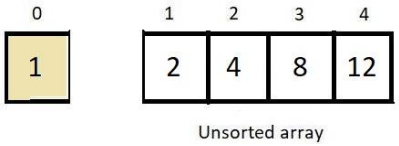
Analysis of Quick Sort Algorithm:

Let's start with the **runtime complexity** of the quick sort algorithm:

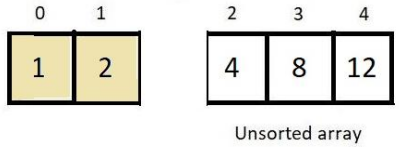
Worst Case: The worst-case in a quicksort algorithm happens when our array is already sorted. I'll take a sorted array of length 5 to demonstrate how it reaches the worst case. Take the one below as an example.



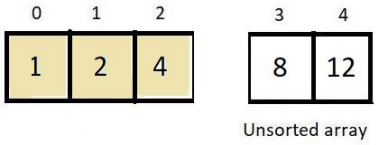
In the first step, we choose 1 as the pivot and apply a partition on the whole array. Since 1 is already at its correct position, we apply quicksort on the rest of the subarrays.



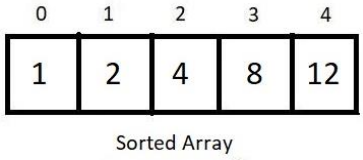
Next, the pivot is element 2, and when applied partition, we found that there is no element less than 2 in the subarray; hence, the pivot remains there itself. We further apply quicksort on the only subarray that is to the right.



Now, the pivot is 4, and since that is already at its correct position, applying partition did make no change. We move ahead.



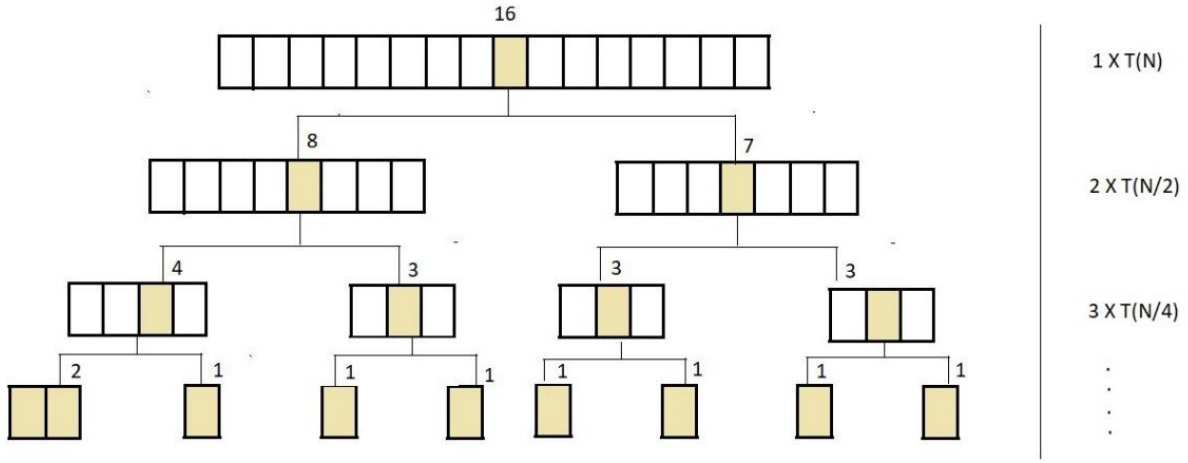
Now, the pivot is 8, and even that is at its correct position; hence things remain unchanged, and there is just one subarray with a single element left. But since any array with a single element is always sorted, we mark the element 12 sorted as well. And hence our final sorted array becomes,



So, if we calculate carefully, for an array of size 5, we had to partition the subarray 4 times. That is, for an array of size n, there would be (n-1) partitions. Now, during each partition, long story short, we made our two index variables, i and j run from either direction towards each other until they actually become equal or cross each other. And we do some swapping in between as well. These operations count to some linear function of n, contributing O(n) to the runtime complexity.

And since there are a total of (n-1) partitions, our total runtime complexity becomes n(n-1) which is simply **O(n2)**. This is our worst-case complexity.

Best Case: The condition when our algorithm performs in its best possible time complexity is when our array gets divided into two almost equal subarrays at each partition. Below mentioned tree defines the state of best-case when we apply quicksort on an array of 16 elements, of which each newly made subarray is almost half of its parent array.



No. partitions were different at each level of the tree. If we count starting from the top, the top-level had one partition on an array of length (n=16), the second level had 2 partitions on arrays of length n/2, then the third level had 4 partitions on arrays of length n/4… and so on.

For the above array of length 16, the calculation goes like the one below.

Here, T(x) is the time taken during the partition of the array with x elements. And as we know, the partition takes a linear function time, and we can assume T(x) to be equal to x; hence the total time complexity becomes,

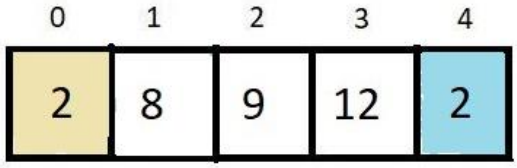
Total time = 1(n) + 2(n/2) +4(n/4) + ...+ until the height of the tree(h) Total time = n*h

H is the height of the tree, and the height of the tree is log2(n), where n is the size of the given array. In the above example, h = 4, since log2(16) equals 4. Hence the time complexity of the algorithm in its best case is **O(nlogn)**.

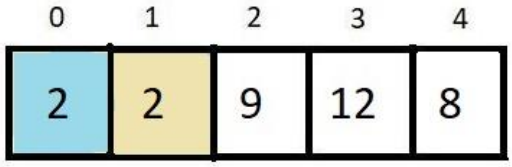
Note: The average time complexity remains **O(nlogn)**. Calculations have been avoided here due to their complexity.

Stability:

The quick sort algorithm is not stable. It does swaps of all kinds and hence loses its stability. An example is illustrated below.



When we apply the partition on the above array with the first element as the pivot, our array becomes



And the two 2s get their order reversed. Hence quick sort is not stable.

Note: Quicksort algorithm is an in-place algorithm. It doesn't consume any extra space in the memory. It does all kinds of operations in the same array itself.

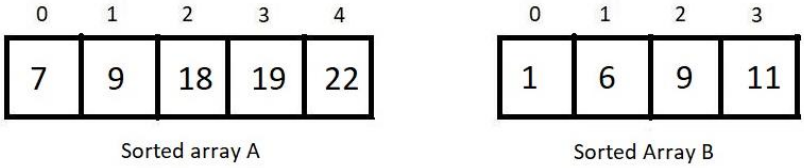
There is no hard and fast rule to choose only the first element as the pivot; rather, we can have any random element as its pivot using the rand() function and that we wouldn’t believe actually reduces the algorithm's complexity.

Merge Sort Algorithm

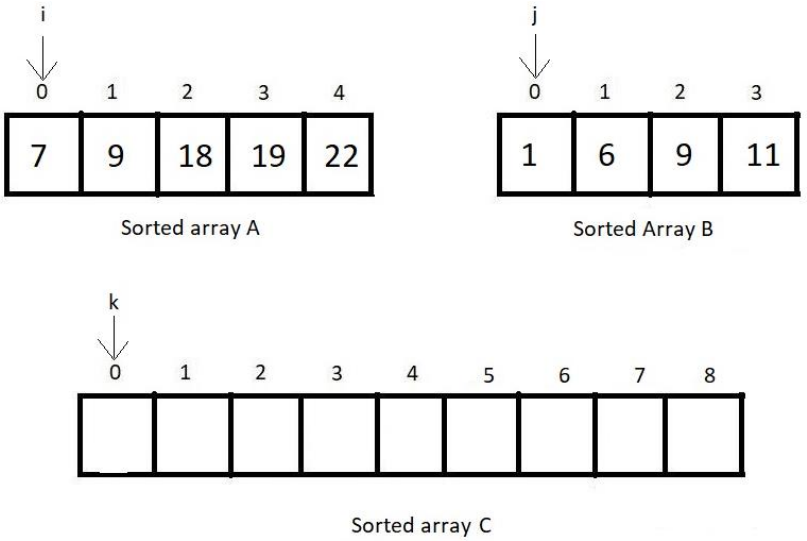
In this algorithm, we divide the arrays into subarrays and subarrays into more subarrays until the size of each subarray becomes 1. Since arrays with a single element are always considered sorted, this is where we merge. Merging two sorted subarrays creates another sorted subarray. We'll show first how merging two sorted subarrays works.

Merging Procedure:

Suppose we have two sorted arrays, A and B, of sizes 5 and 4, respectively.

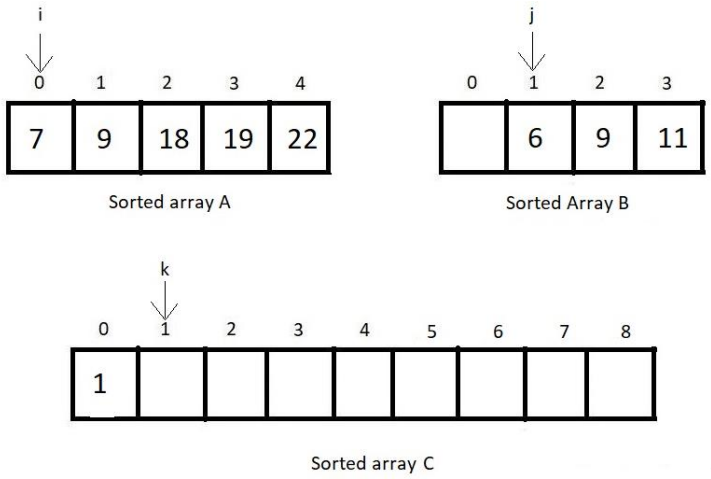


- And we apply merging on them. Then the first job would be to create another array C with size being the sum of both the raw arrays' sizes. Here the sizes of A and B are 5 and 4, respectively. So, the size of array C would be 9.
- Now, we maintain three index variables i, j, and k. i looks after the first element in array A, j looks after the first element in array B, and k looks after the position in array C to place the next element in.

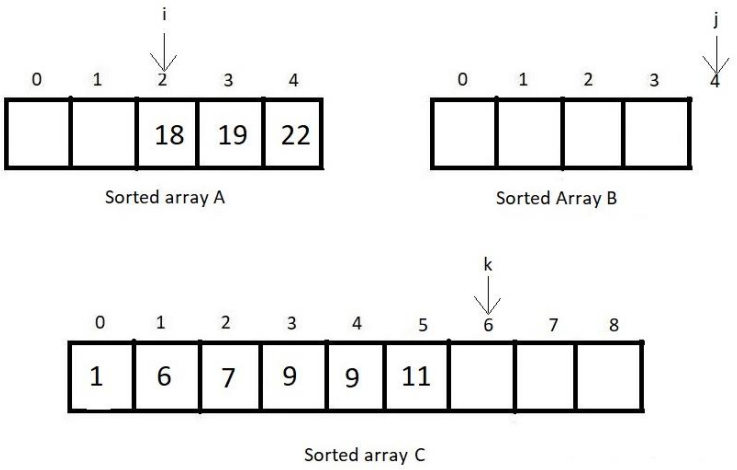


- Initially, all i, j, and k are equal to 0.
- Now, we compare the elements at index i of array A and index j of array B and see which one is smaller. Fill in the smaller element at index k of array C and increment k by 1. Also, increment the index variable of the array we fetched the smaller element from.

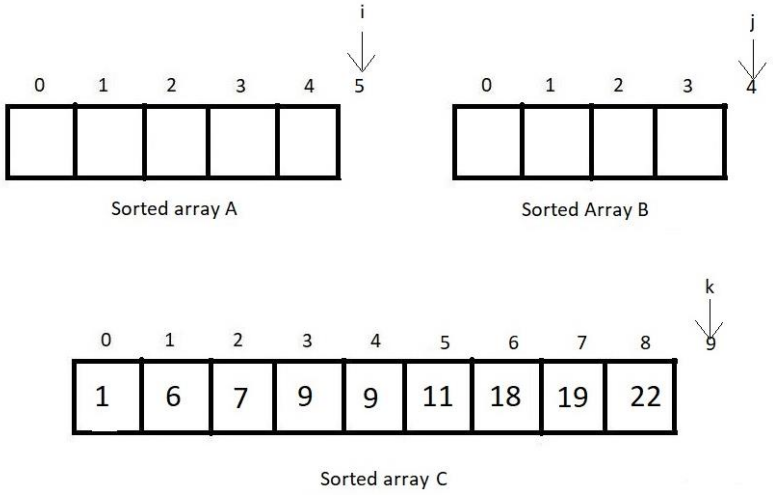
- Here, A[i] is greater than B[j]. Hence, we fill C[k] with B[j] and increase k and j by 1.



- We continue doing step 5 until one of the arrays, A or B, gets empty.



Here, array B inserted all its elements in the merged array C. Since we are only left with the elements of element A, we simply put them in the merged array as they are. This will result in our final merged array C.



The programming part of the merge procedure is not that tough. We just need to follow these steps:

a. Take both the arrays and their sizes to be merged as the parameters of the merge function. By summing the sizes of the two arrays, we can create one larger array.

b. Create three index variables i, j & k. And initialize all of them with 0.

c. And then run a while loop with the condition that both the index variables i and j don't exceed their respective array limits.

d. Now, at each run, see if A[i] is smaller than B[j], if yes, make C[k] = A[i] and increase both i and k by 1, else C[k] = B[j] and both j and k are incremented by 1.

e. And when the loop finishes, either array A or B or both get finished. And now we run two while loops for each array A and B, and insert all the remaining elements as they are in the array C. And we are done merging.

❖ The pseudocode for the above procedure has been attached below.

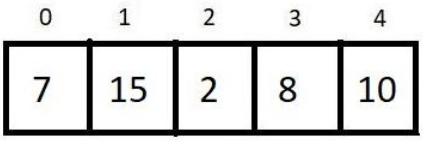
```
void Merge(int A[], int B[], int C[], int n, int m)
{
  int i=0, j=0, k=0;
  while (i <=n && j <= m){
    if (A[i] < B[j]){
      C[k] = A[i];
      i++;
      k++;
    }
    else{
      C[k] = B[j];
      j++;
      k++;
    }
  }
  while (i <=n){
    C[k] = A[i];
    k++;
    i++;
  }
  while (j <= m){
    C[k] = B[j];
    k++;
    j++;
  }
}
```

Copying all remaining elements from A to C

Copying all remaing elements from B to C

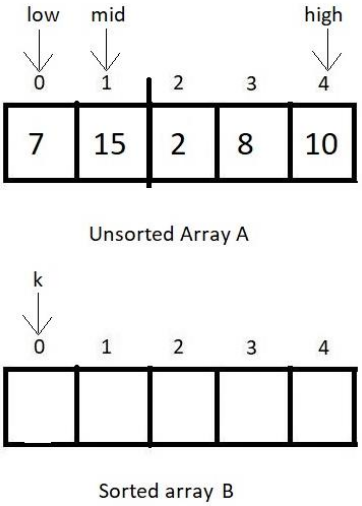
Now, this would quite not be our situation when sorting an array using the merge sort. We wouldn't have two different arrays A and B, rather a single array having two sorted subarrays. So, how to merge two sorted subarrays of a single array in the array itself.

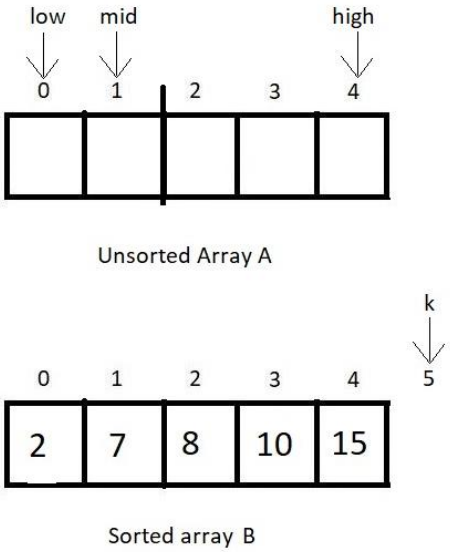
Suppose there is an array A of 5 elements and contains two sorted subarrays of length 2 and 3 in itself.



Unsorted Array A

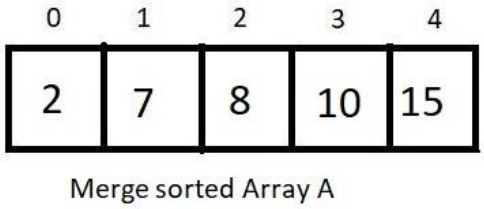
To merge both the sorted subarrays and produce a sorted array of length 5, we will create an auxiliary array B of size 5. Now the process would be more or less the same, and the only change we would need to make is to consider the first index of the first subarray as low and the last index of the second subarray as high. And mark the index prior to the first index of the second subarray as mid.





Previously we had index variables i, j, and k initialized with 0 of their respective arrays. But here, i gets initialized with low, j gets initialized with mid+1, and k gets initialized with low only. And similar to what we did earlier, i runs from low to mid, j runs from mid+1 to high, and until and unless they both get all their elements merged, we continue filling elements in array B.

After all the elements get filled in array C, we revert back to our original array A and fill the sorted elements again from low to high, making our array merge-sorted.



So that was our merging segment.

There were few changes we had to make in the pseudocode.

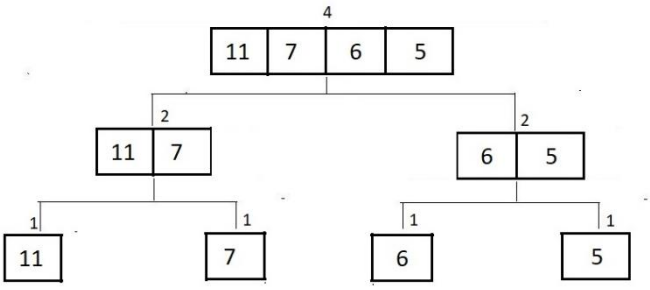
```
void merge(int A[], mid,  low, high)
{
   int i, j, k , B[high+1];
   i = low;
   j = mid + 1;
   k = low;
   while (i <= mid && j <= high){
      if (A[i] < A[j]){
         B[k] = A[i];
         i++;
         k++;
      }
      else{
         B[k] = A[j];
         j++;
         k++;
      }
   }
   while (i <= mid){
      B[k] = A[i];
      k++;
      i++;
   }
   while (j <= high){
      B[k] = A[j];
      k++;
      j++;
   }
   for (int i = low; i <= high; i++)
      A[i] = B[i];
```

Copying all remaining elements from A to C

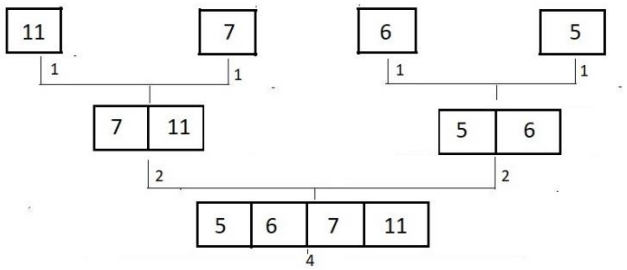
Copying all remaing elements from B to C

Copying elements back to A from B

Whenever we receive an unsorted array, we break the array into fragments till the size of each subarray becomes 1. Let this be clearer via an illustration.



So, we divided the array until there are all subarrays of just length 1. Since any array/subarray of length 1 is always sorted, we just need to merge all these singly-sized subarrays into a single entity. Visit the merging procedure below.

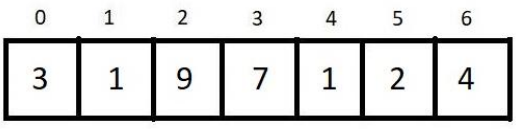


And this is how our array got merge sorted. To achieve this divided merging and sorting, we create a recursive function merge sort and pass our array and the low and high index into it. This function divides the array into two parts: one from low to mid and another from mid+1 to high. Then, it recursively calls itself passing these divided subarrays. And the resultant subarrays are sorted. In the next step, it just merges them. And that's it. Our array automatically gets sorted. Pseudocode for the merge sort function is illustrated below.

```
void MS(A[], low, high){
   int mid;
   if(low<high){
      mid = (low + high) /2;
      MS(A, low, mid);
      MS(A, mid+1, high);
      Merge(A, mid, low, high);
   }
}
```

Count Sort Algorithm

The most intuitive and **easiest** sorting algorithm, known as the count sort algorithm. Suppose we are given an array of integers and are asked to sort them using any sorting algorithm of our choice, then it is not difficult to generate the resultant array, which is just the sorted form of the given array. Still, the method you choose to reach the result matters the most. Count Sort is one of the **fastest** methods of all. The below figure shows the array given.



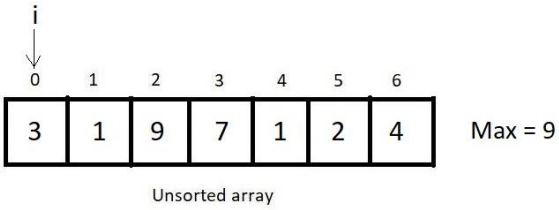
Unsorted array

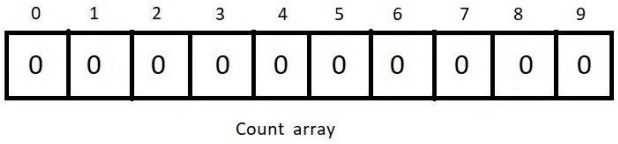
- The algorithm first asks us to find the largest element from all the elements in the array and store it in some integer variable max. Then create a count array of size max+1. This array would count the number of occurrences of some number in the given array. We will have to initialize all count array elements with 0 for that to work.
- After initializing the count array, traverse through the given array, and increment the value of that element in the count array by 1. By defining the size of the count array as the maximum element in the array, we ensure that

each element in the array has its own corresponding index in the count array. After we traverse through the whole array, we'll have the count of each element in the array.

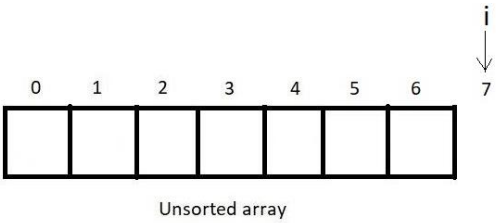
- Now traverse through the count array, and look for the nonzero elements. The moment we find an index with some value other than zero, fill in the sorted array the index of the non-zero element until it becomes zero by decrementing it by 1 every time we fill it in the resultant array. And then move further. This way, we create a sorted array. Let's take the one we have as an example above and use the count sort algorithm to sort it.

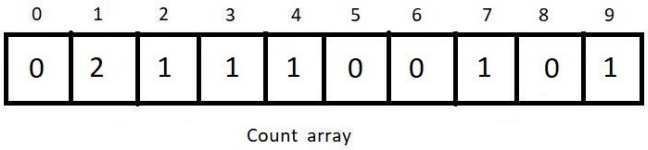
First of all, find the maximum element in the array. Here, it is 9. So, we'll create a count array of size 10 and initialize every element by 0.



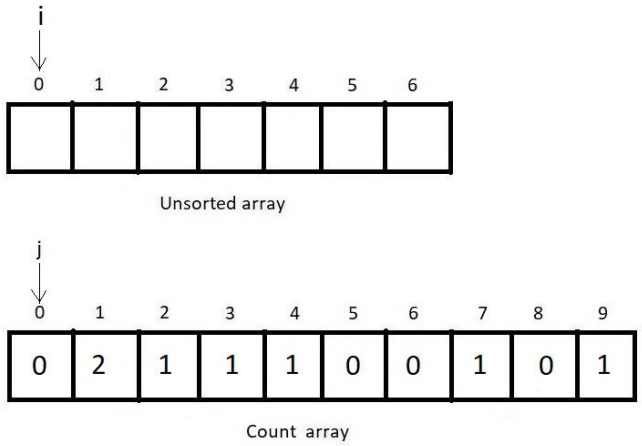


Now, let's iterate through the given array and count the no. of occurrences of each number less than equal to the maximum element.

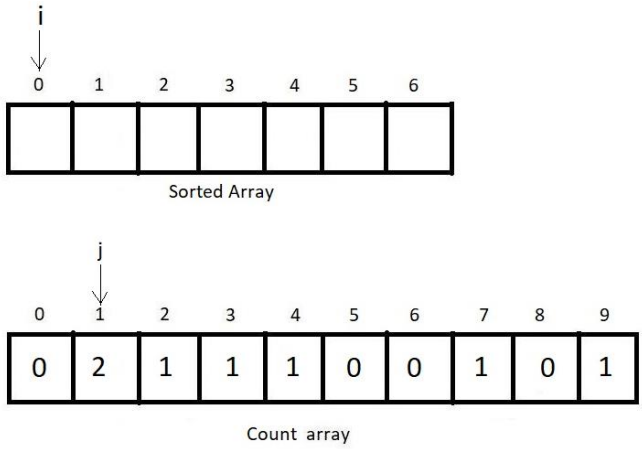




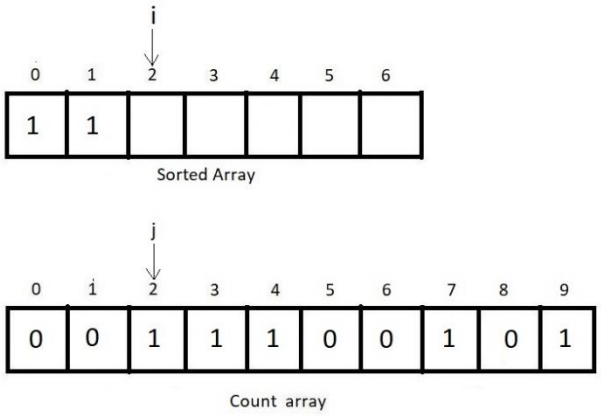
We would iterate through the count array and fill the unsorted array with the index we encounter having a non-zero number of occurrences.



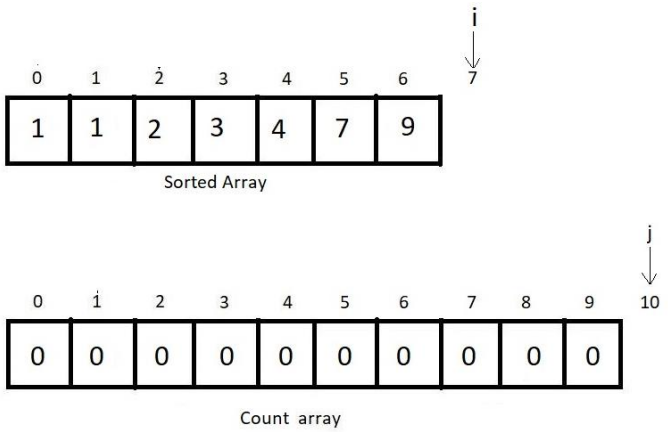
Now since there are zero numbers of zeros in the given array, we move further to index 1.



And since there were two ones in the array we were given, we push two ones in the sorted array and move our iterator to the next empty index.



And following a similar procedure for all the elements in the count array, we reach our sorted array in no time.



Time Complexity of Count Sort:

If we carefully observe the whole process, we only ran two different loops, one through the given array and one through the count array, which had the size equal to the maximum element in the given array. If we suppose the maximum element to be m, then the algorithm's time complexity becomes O(n+m), and for an array of some huge size, this reduces to just O(n), which is the most efficient by far algorithm.

However, there is a negative point as well. The algorithm uses extra space for the count array. And this linear complexity is reachable only at the cost of the space the count array takes.