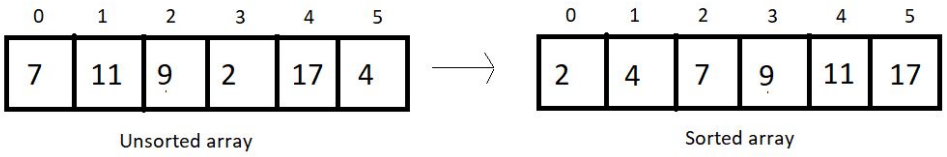
Bubble Sort Algorithm

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.

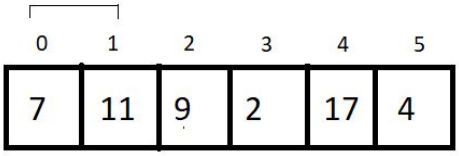
At each pass, we will iterate through the unsorted part of the array and compare every adjacent pair. We move ahead if the adjacent pair is sorted; otherwise, we make it sorted by swapping their positions. And doing this at every pass ensures that the largest element of the unsorted part of the array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

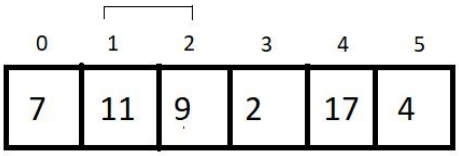
1st Pass:

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.

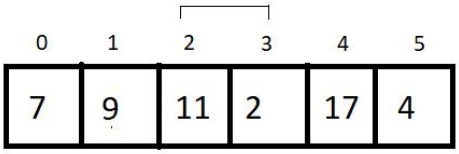
Let’s start with the first one.



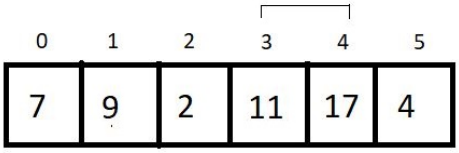
Since these two are already sorted, we move ahead without making any changes.



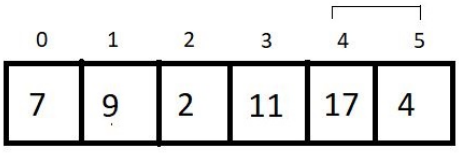
Now since 9 is less than 11, we swap their positions to make them sorted.



Again, we swap the positions of 11 and 2.

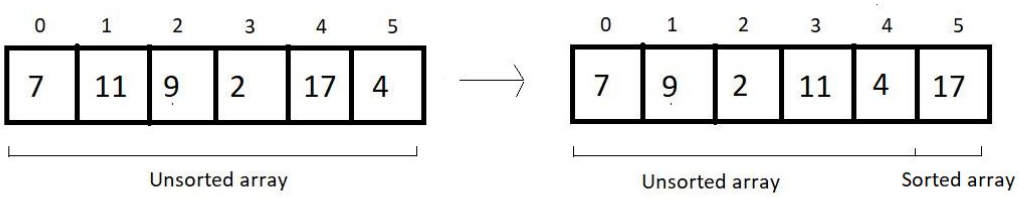


We move ahead without changing anything since they are already sorted.



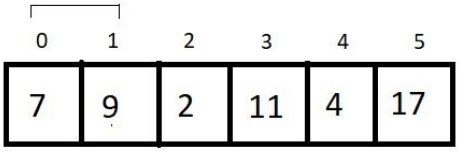
Here, we make a swap since 17 is greater than 4.

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

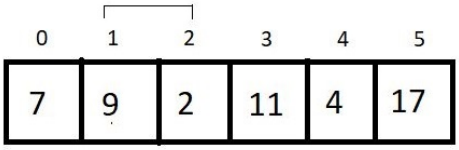


2nd Pass:

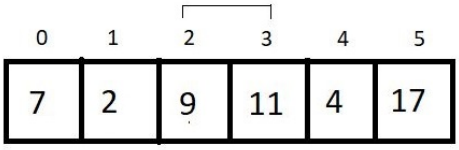
We again start from the beginning, with a reduced unsorted part of length 5. Hence the number of comparisons would be just 4.



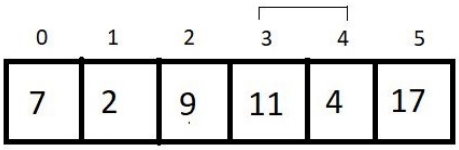
No changes to make.



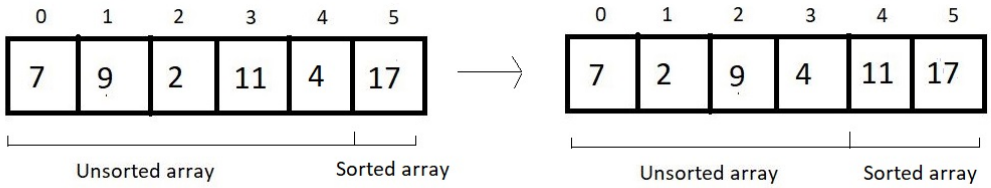
Yes, here we make a swap, since 9>2.



Since 9 < 11, we move further.

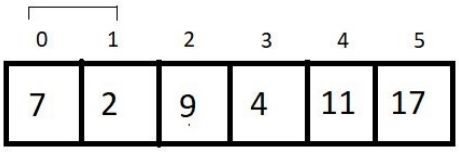


And since 11 is greater than 4, we make a swap again. And that would be it for the second pass. Let's see how close we have reached to the sorted array.

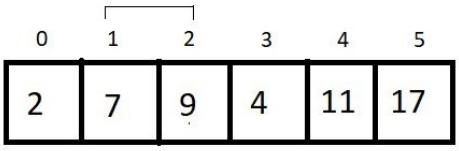


3rd Pass:

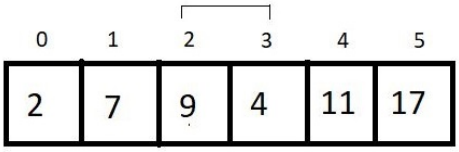
We'll again start from the beginning, and this time our unsorted part has a length of 4; hence no. of comparisons would be 3.



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

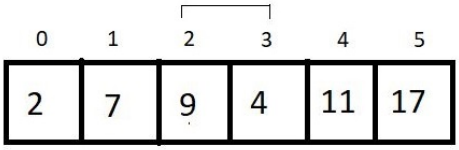


We move ahead without making any change.



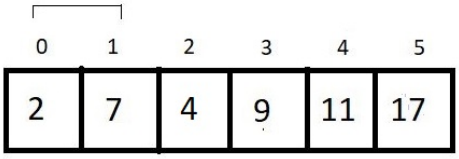
In this final comparison, we make a swap, since 9 > 4.

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

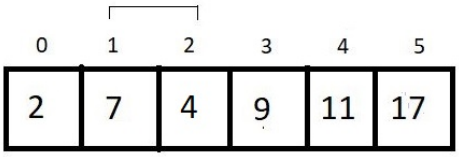


4th Pass:

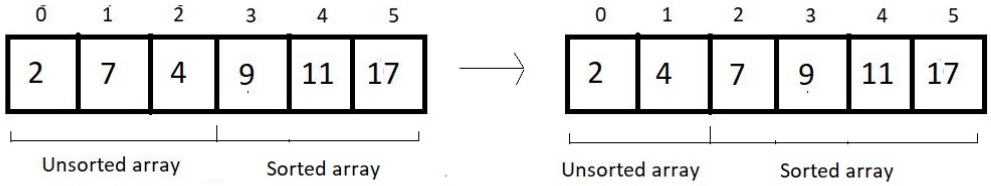
We just have the unsorted part of length 3, and that would cause just 2 comparisons. So, let's see them.



No changes here.

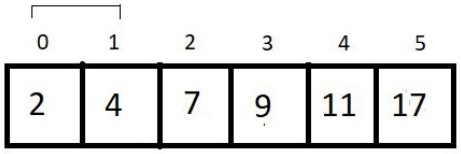


We swap their positions. And that is all in the 4th pass. The resultant array after the 4th pass is:

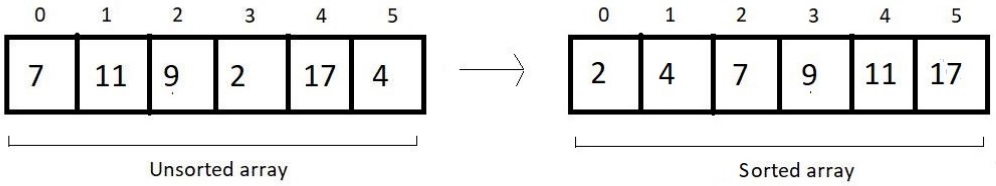


5th (last) pass:

We have only one comparison to make here.



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

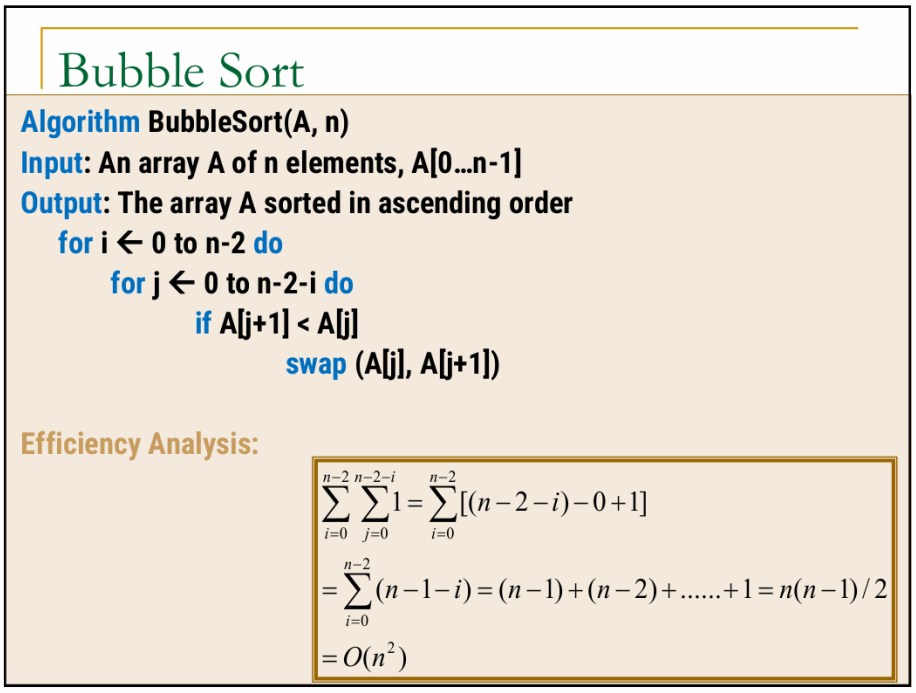


And this is what the Bubble Sort algorithm looks like. We have a few things to conclude and few calculations regarding the complexity of the algorithm to make.

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.

```
void bubbleSortAdaptive(int *A, int n){
    int temp;
    int isSorted = 0;
    for (int i = 0; i < n-1; i++) // For number of pass
    {
        printf("Working on pass number %d\n", i+1);
        isSorted = 1;
        for (int j = 0; j <n-1-i ; j++) // For comparison in each pass
        {
            if(A[j]>A[j+1]){
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
                isSorted = 0;
            }
        }
        if(isSorted){
            return;
        }
    }
}
```



Bubble Sort Complexity Analysis:

- **Best Case**: O(n)

Condition: The input array is already sorted. In this case, Bubble Sort only needs one pass through the array to confirm that no swaps are necessary.

- **Worst Case**: O(n^2)

  Condition: The input array is sorted in reverse order. Each element must be compared with every other element, leading to O(n^2) comparisons and swaps.

- **Average Case**: O(n^2)

  Condition: For random inputs, Bubble Sort will perform on average O(n^2) comparisons and swaps, as each element interacts with every other element in the array.