Sorting:

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

Importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner.

Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

Telephone Directory − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

Dictionary − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

A. In-place Sorting and Not-in-place Sorting

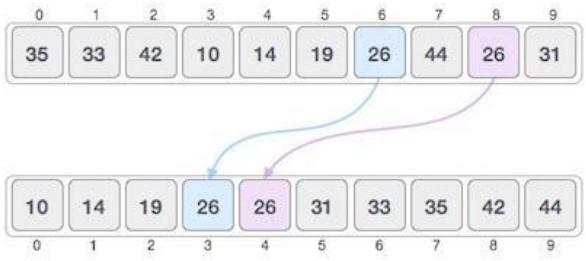
Sorting algorithms may require some extra space for comparison and temporary storage of few data elements.

These algorithms do not require any extra space and produces an output in the same memory that contains the data by transforming the input 'in-place': **in-place sorting**, eg, Bubble sort

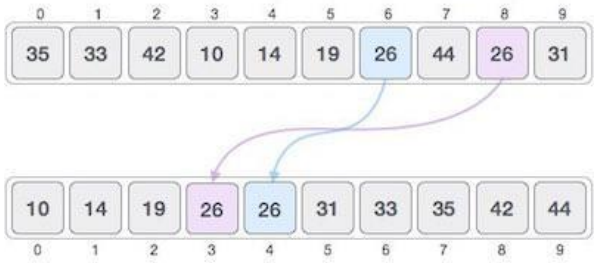
Sorting which uses equal or more extra space : **not-in-place sorting**.e,g, Merge-sort

B. Stable and Not Stable Sorting

If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.



If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

C. Adaptive and Non-Adaptive Sorting Algorithm

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting **if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them**.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They **try to force every single element to be re-ordered to confirm their sortedness**.

Order of Sorting:

Sequence of the values is called order.

Ascending (Increasing)
Descending (Decreasing)

Sorting technique depends on the situation. It depends on two parameters.

1. **Execution time** of program that means time taken for execution of program.
2. **Space** that means space taken by the program.

Sorting techniques are differentiated by their efficiency and space requirements.

Bubble Sort
Insertion Sort
Selection Sort
Quick Sort

Tree structure based Sort

Tree sort
Heap Sort

1. Bubble Sort

Simple illustration:

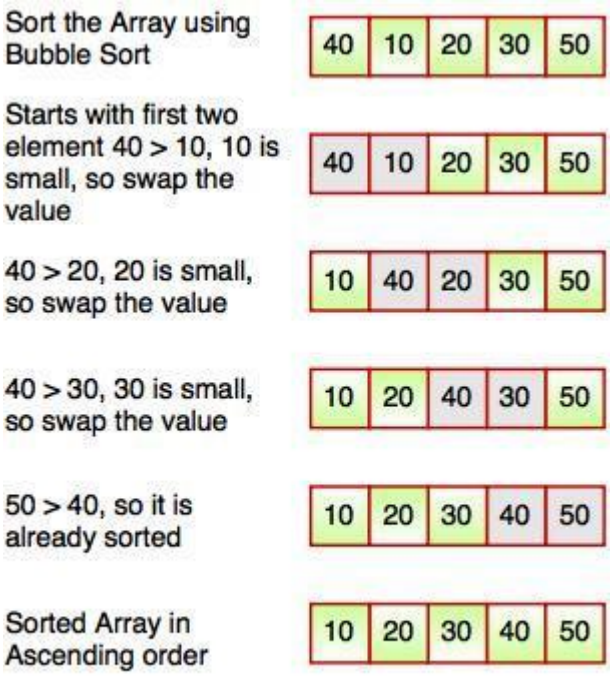


Fig. Working of Bubble Sort

Interesting Perception of Bubble Sorting:

With Bubble Sort (sometimes "Bubblesort"), **two successive elements are compared** with each other, and – if the left element is larger than the right one – they are swapped.

These comparison and swap operations are performed from left to right across all elements. Therefore, after the first pass, the largest element is positioned on the far right. Or better: *at the latest* after the first pass – it may have arrived there before.

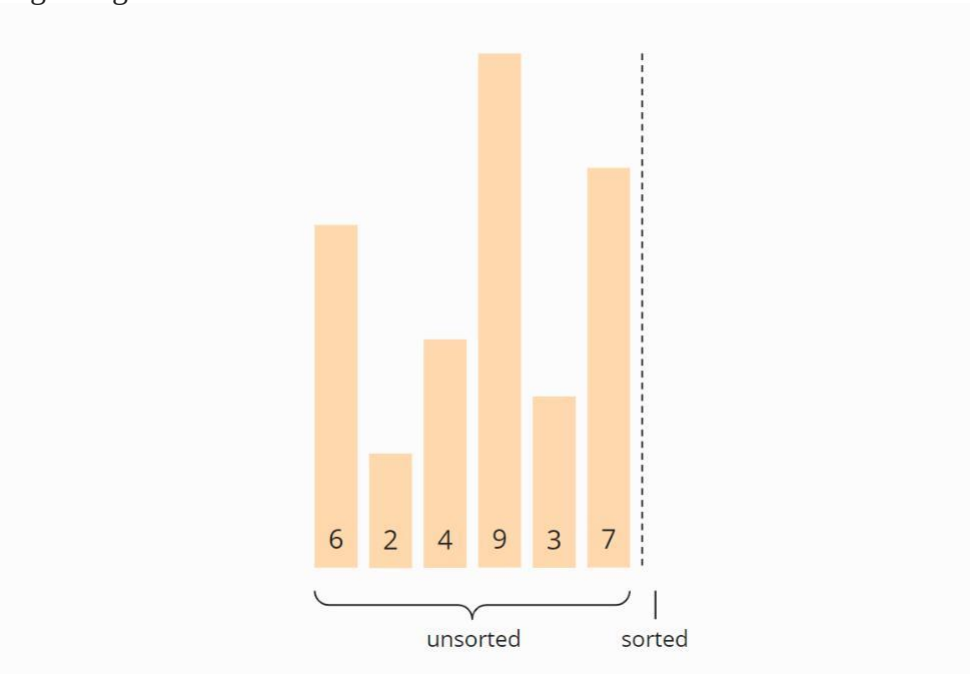
You **repeat this process n-1** times to make sure that all the elements are sorted

Bubble Sort Example

In the following visualizations, I show how to sort the array [6, 2, 4, 9, 3, 7] with Bubble Sort:

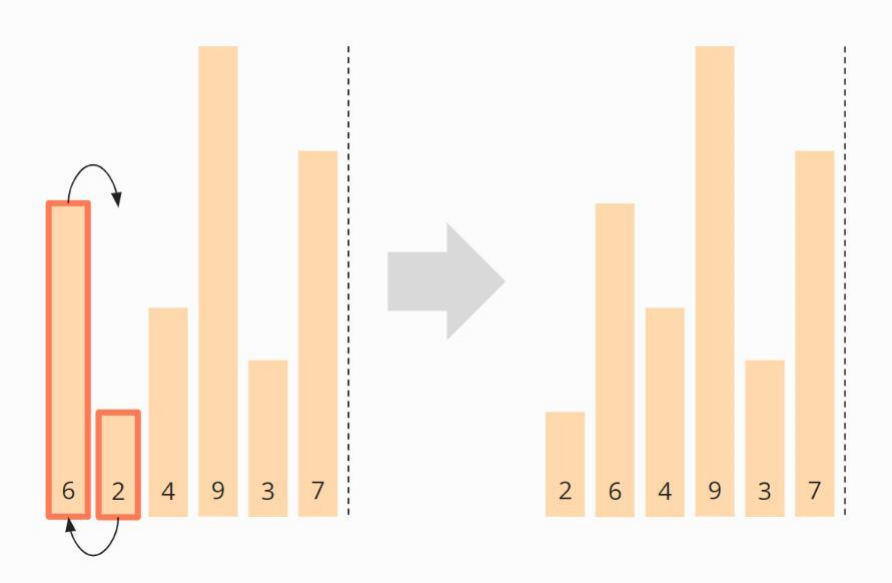
Preparation

We divide the array into a left, unsorted – and a right, sorted part. The right part is empty at the beginning:

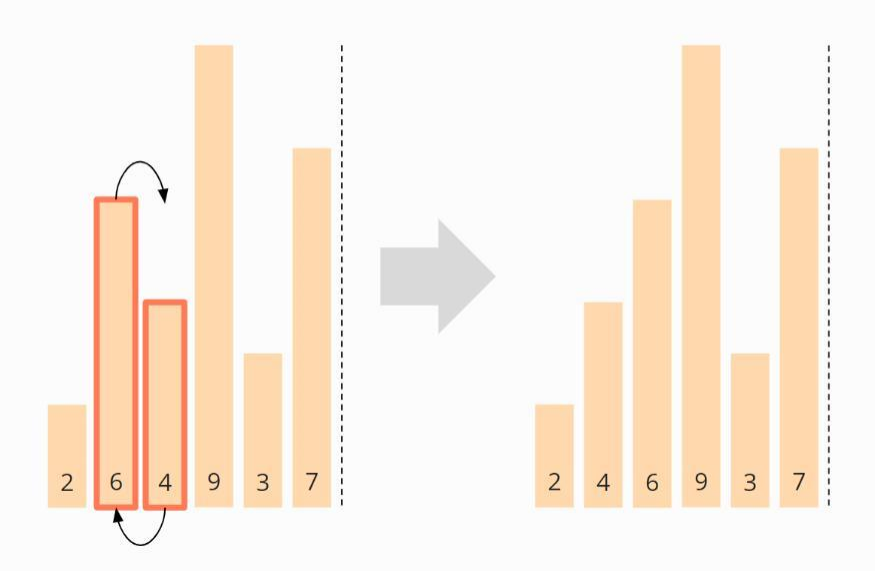


Iteration 1

We compare the first two elements, the 6 and the 2, and since the 6 is smaller, we swap the elements:

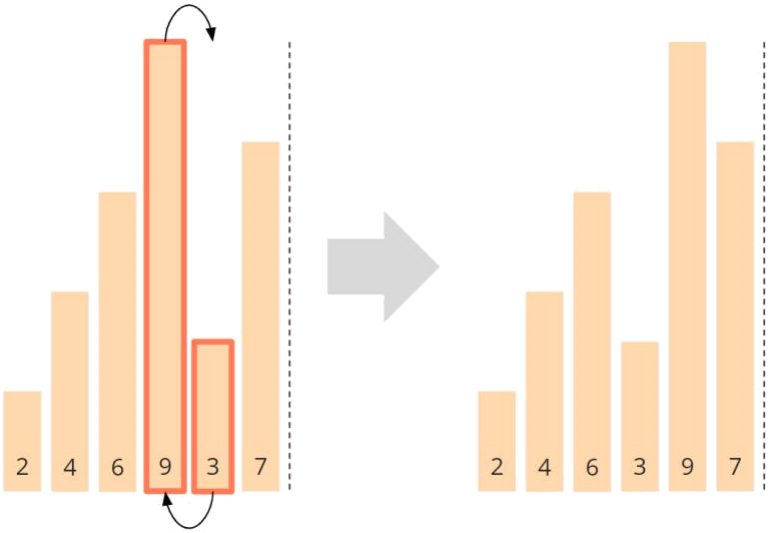


Now we compare the second with the third element, i.e., the 6 with the 4. These are also in the wrong order and are, therefore, swapped:

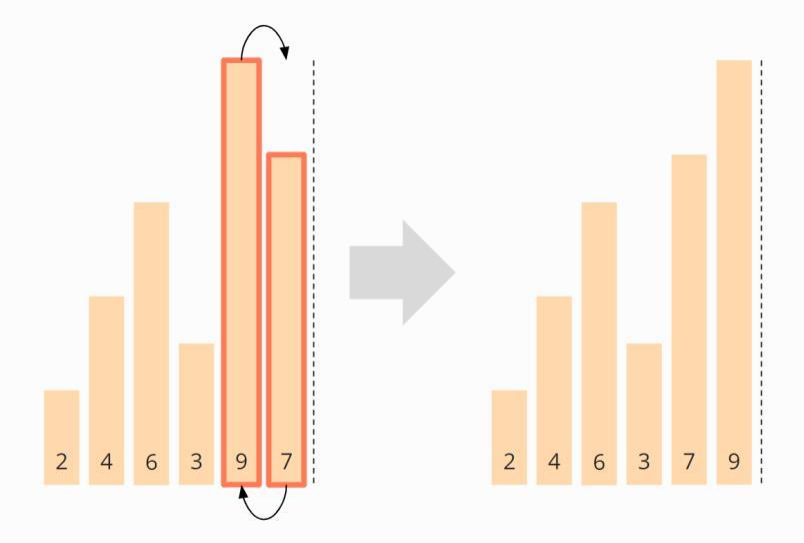


We compare the third with the fourth element, i.e., the 6 with the 9. The 6 is smaller than the 9, so we do not need to swap these two elements.

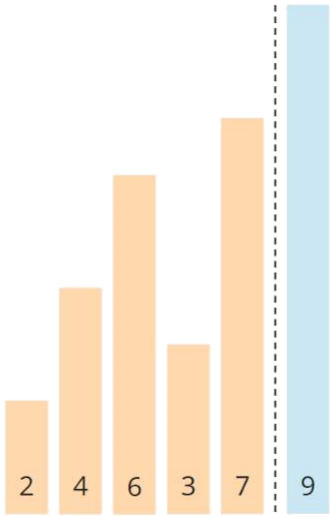
The fourth and fifth element, the 9 and the 3, need to be swapped again:



And finally, the fifth and sixth elements, the 9 and the 7, must be swapped. After that, the first iteration is finished.



The 9 has reached its final position, and we move the border between the areas one field to the left:



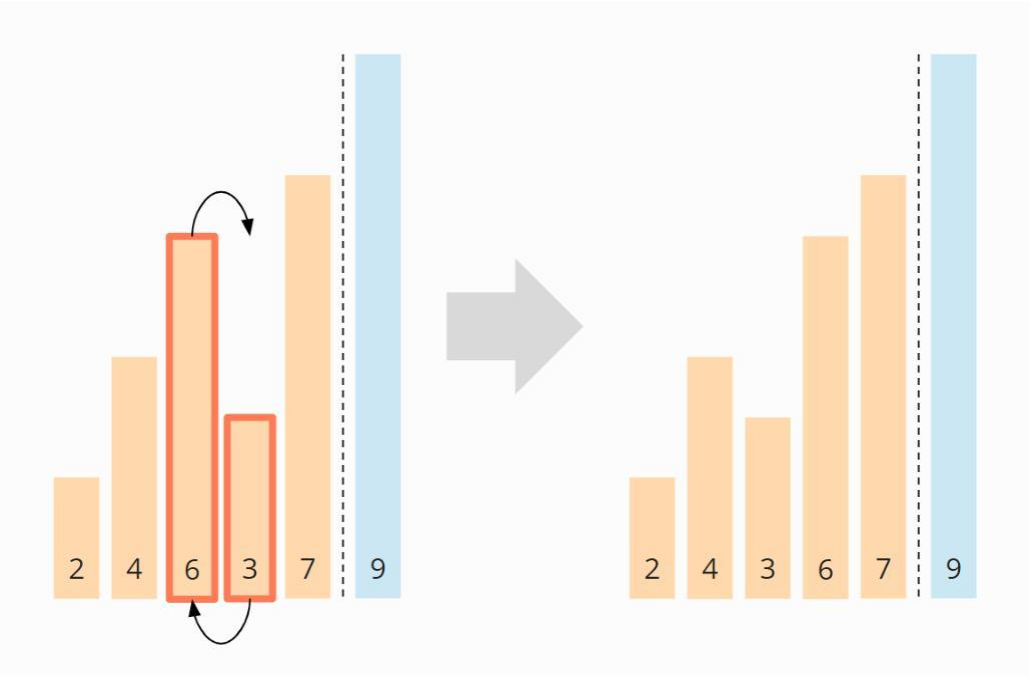
In the next iteration, this boundary shows us up to which position the elements have to be compared. By the way, the area boundary only exists in the optimized version of Bubble Sort. In the original variant, it is missing. Consequently, in every iteration, the comparison is performed unnecessarily until the end of the array.

Iteration 2

We start again at the beginning of the array and compare the 2 with the 4. These are in the correct order and need not be swapped.

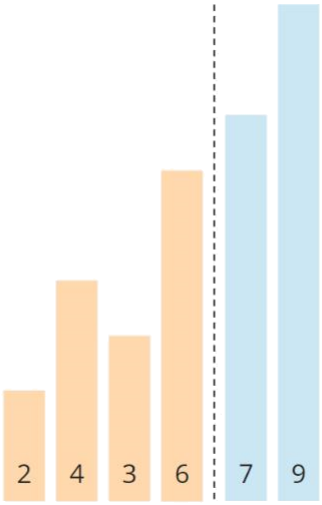
The same applies to the 4 and the 6.

The 6 and the 3, however, must be swapped to be in the correct order:



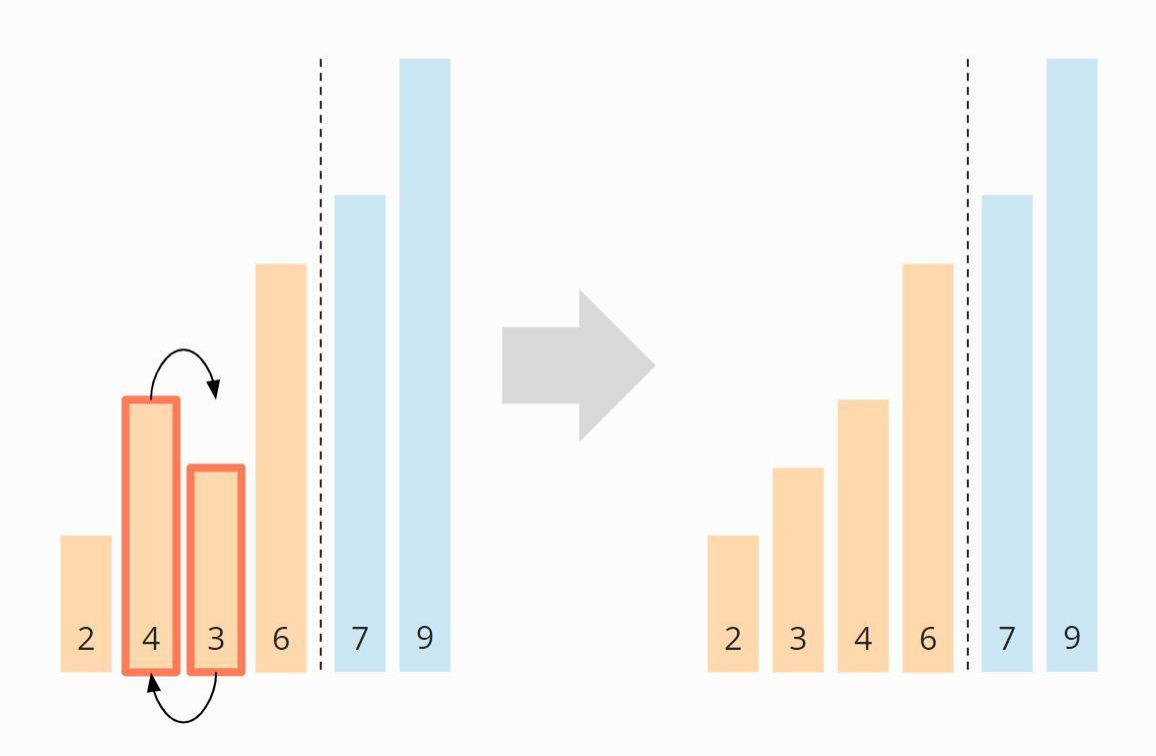
The 6 and the 7 are in the right order and do not need to be swapped. We do not need to compare further since the 9 is already in the sorted area.

Finally, we move the area boundary one position to the left again so that we don't have to look at the last two elements, the 7 and the 9, any further.

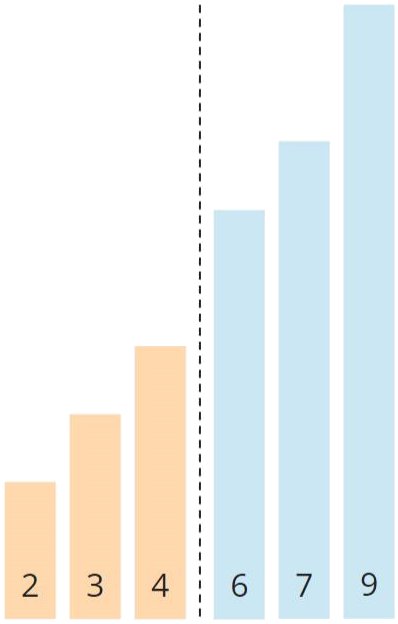


Iteration 3

Again we start at the beginning of the array. The 2 and the 4 are positioned correctly to each other. The 4 and the 3 must be swapped:

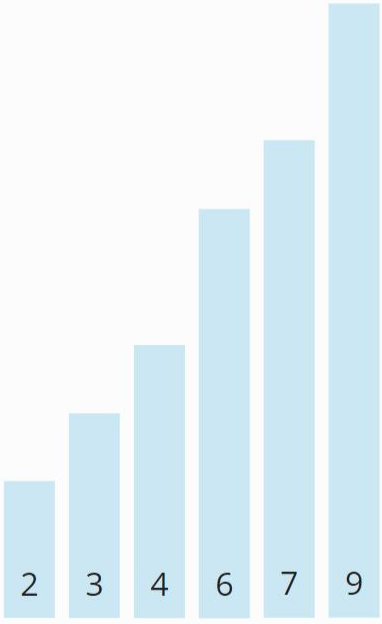


The 4 and the 6 do not have to be swapped. The 7 and the 9 are already sorted. So this iteration is already finished, and we move the area border to the left:



Iteration 4

We start again at the beginning of the array. In the unsorted area, neither the 2 and 3 nor the 3 and 4 have to be swapped. Now all elements are sorted, and we can finish the algorithm.



Origin of the Name

When we animate the previous example's swapping operations, the elements gradually rise to their target positions – similar to bubbles, hence the name "Bubble Sort":

Algorithm:

```
Algorithm Bubblesort(input: n, V)
def n : number of values to sort;
def V[n] : array of size n;
def temp, i, j: integer variables;
1. i := n - 1;
2. while ( i >= 1 ) do
3.     j := 0;
4.     while ( j < i ) do
5.           if ( V[j] < V[j+1] )
6.                 temp   := V[j];
7.                 V[j]   := V[j+1];
8.                 V[j+1]:= temp;
9.           end if
10.          j := j + 1;
11.    end while
12.    i := i - 1;
13. end while
14. return V;
```

Best Case Time Complexity

Let's start with the most straightforward case: If the numbers are already sorted in ascending order, the algorithm will determine in the first iteration that no number pairs need to be swapped and will then terminate immediately.

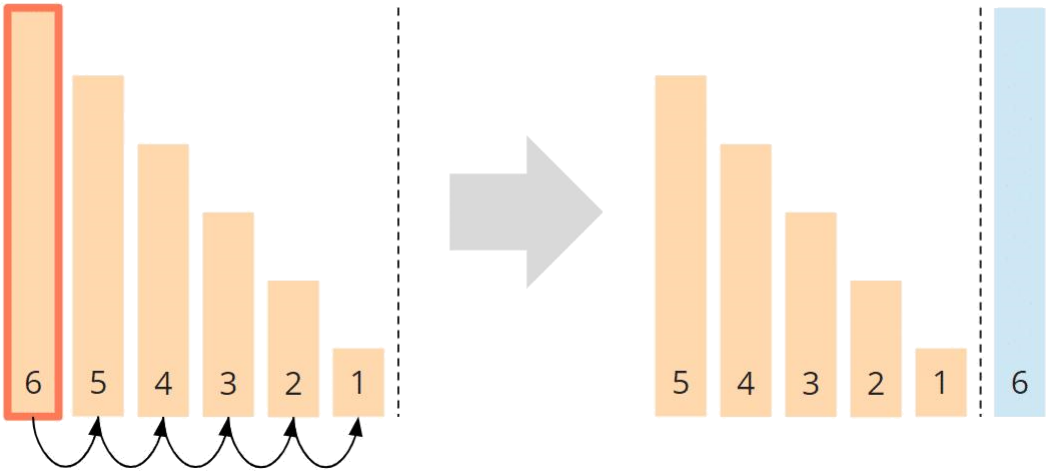
The algorithm must perform n-1 comparisons; therefore:

The best-case time complexity of Bubble Sort is: *O(n)*

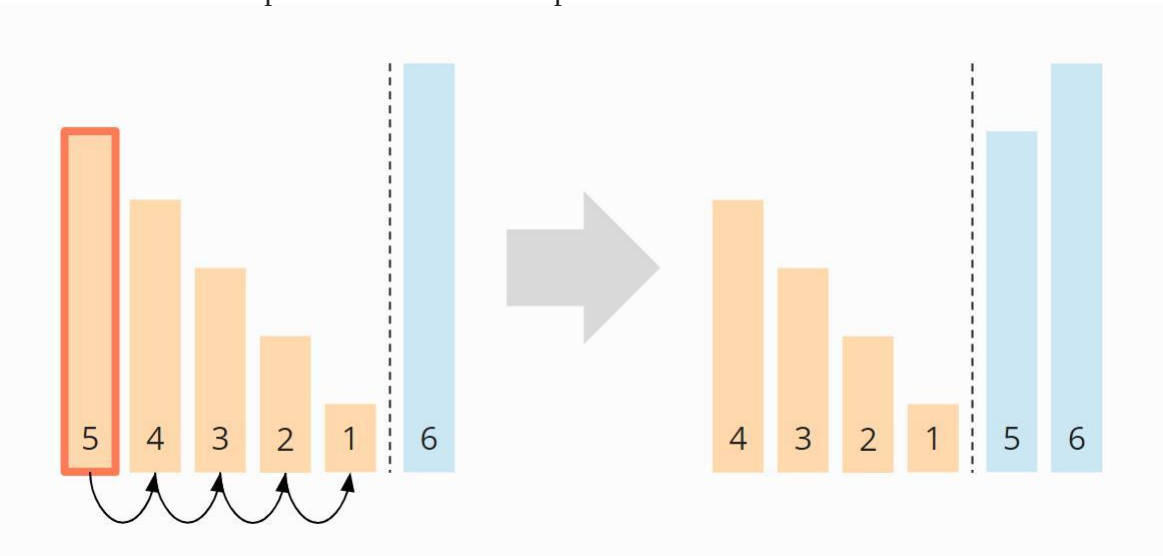
Worst Case Time Complexity

I will demonstrate the worst case with an example. Let's assume we want to sort the descending array [6, 5, 4, 3, 2, 1] with Bubble Sort.

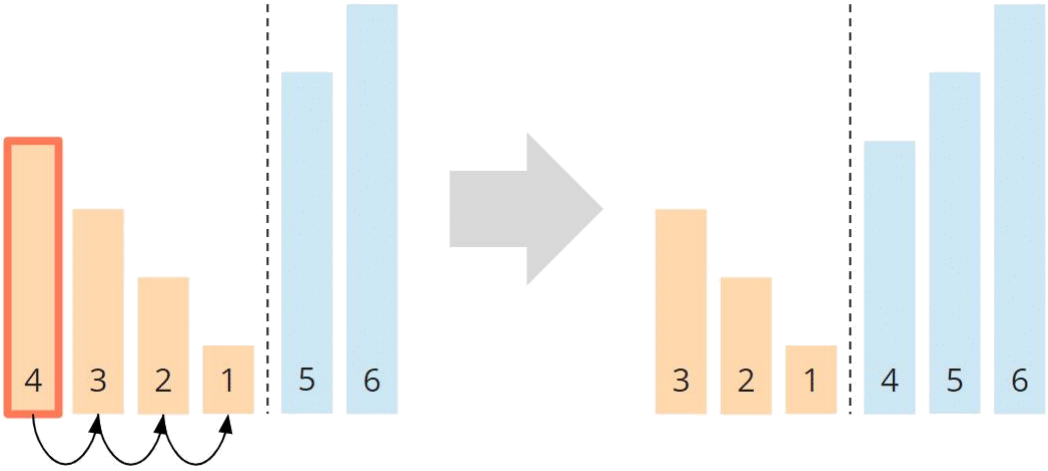
In the first iteration, the largest element, the 6, moves from far left to far right. I omitted the five single steps (swapping the pairs 6/5, 6/4, 6/3, 6/2, 6/1) in the figure:



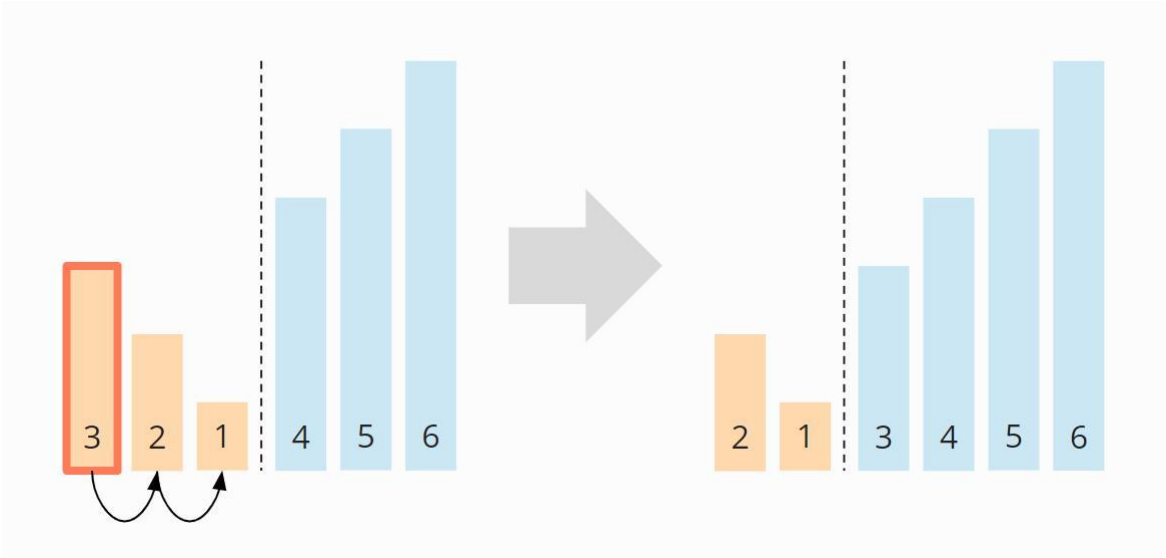
In the second iteration, the second largest element, the 5, is moved from the far left – via four intermediate steps – to the second last position:



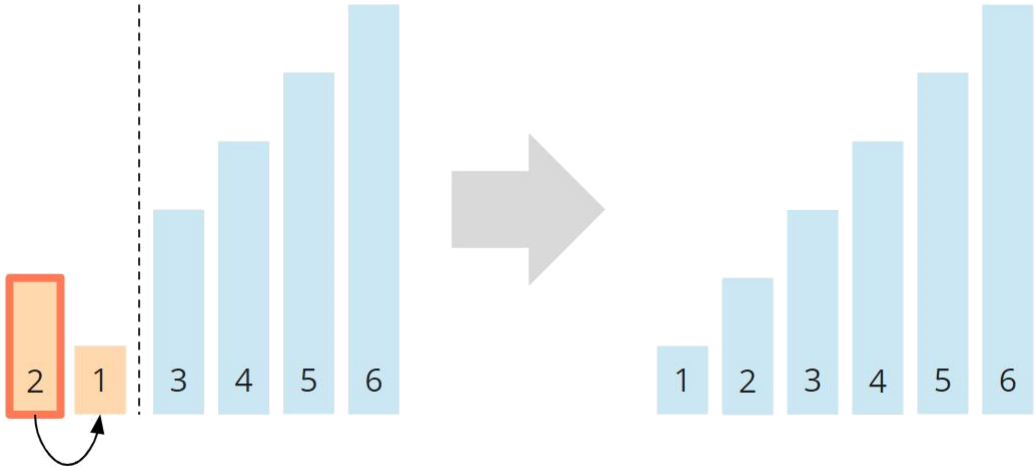
In the third iteration, the 4 is pushed to the third last place – via three intermediate steps.



In the fourth iteration, the 3 is moved – via two single steps – to its final position:



And finally, the 2 and the 1 are swapped:



So in total we have 5 + 4 + 3 + 2 + 1 = 15 comparison and exchange operations.

We can also calculate this as follows:

Six elements times five comparison and exchange operations; divided by two, since on average across all iterations, half of the elements are compared and swapped:

$$6 \times 5 \times ½ \quad = \quad 30 \times ½ \quad = \quad 15$$

$$(n-1) + (n-2) + (n-3) + ... + 3 + 2 + 1$$

$$\text{Sum} = n(n-1)/2$$

The highest power of n in this term is n^2; therefore:

The worst-case time complexity of Bubble Sort is: $O(n^2)$

Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.
Following are the Time and Space complexity for the Bubble Sort algorithm.

Worst Case Time Complexity [Big-O]: **$O(n^2)$**
// Best Case Time Complexity [Big-omega]: **$\Omega(n)$**
Average Time Complexity [Big-theta]: **$\Theta(n^2)$**
Space Complexity: **O(1)**

Pros and cons of Bubble sort:

Pros: Bubble sort algorithm is considered as very simple sorting technique since all you need to do is compare all the adjacent elements and swap them if they are in wrong order.

The main advantage of Bubble Sort is the simplicity of the algorithm.

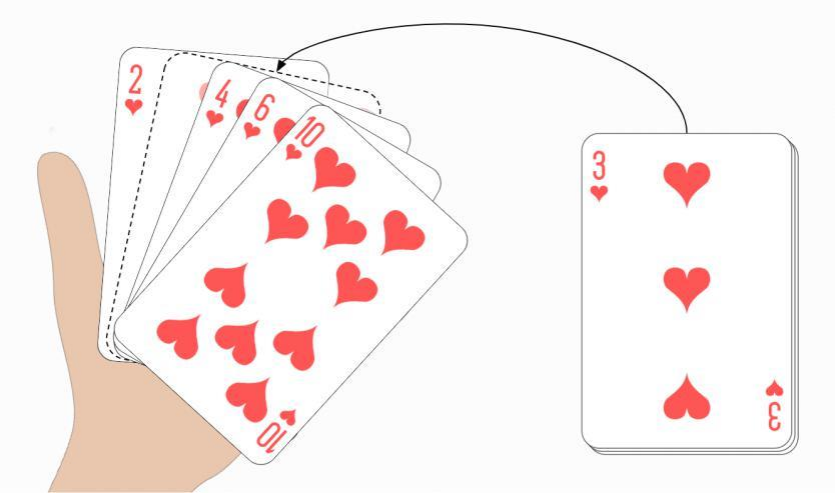
The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for temp variable. (In place)

Cons: Main drawback of bubble sort is its **time complexity which is O(N^2)** since all the pairs are compared, even when the original array is sorted.

Insertion Sort Algorithm

Example: Sorting Playing Cards

Let us start with a playing card example.



Consider you have 10 cards out of a deck of cards in your hand. And they are sorted, or arranged in the ascending order of their numbers.

If I give you another card, and ask you to **insert the card in just the right position**, so that the cards in your hand are still sorted. What will you do?

Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card. Once you find the right position, you will **insert** the card there.

Similarly, if more new cards are provided to you, you can easily repeat the same process and insert the new cards and keep the cards sorted too.

Have you ever sorted cards this way before?

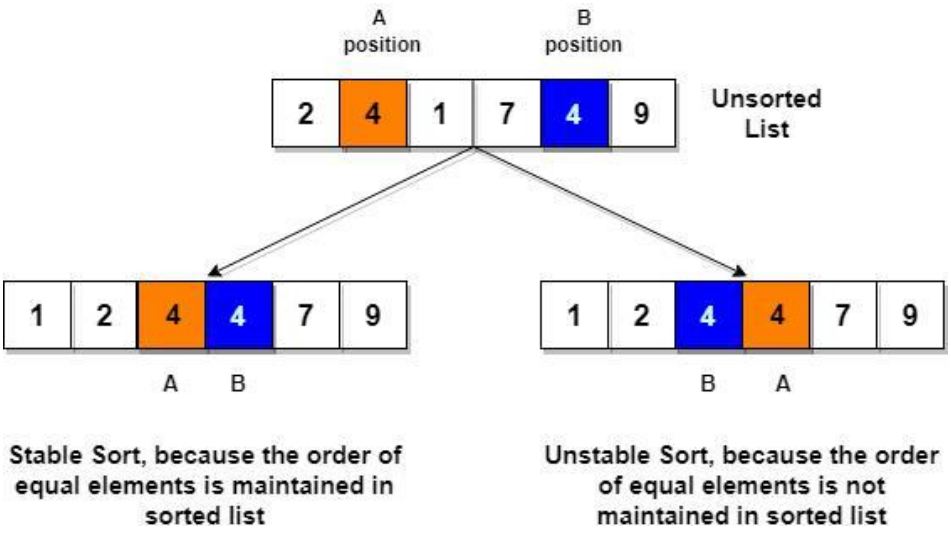
If so, then you have intuitively used "Insertion Sort".

This is exactly how **insertion sort** works. It starts from the index 1(not 0), and each index starting from index 1 is like a new card, that you have to place at the right position in the sorted subarray on the left.

Some of the important characteristics of Insertion Sort:

1. It is efficient for smaller data sets, but very inefficient for larger lists.
2. **Insertion Sort is adaptive**, that means it reduces its total number of steps if a partially sorted array is provided as input, making it efficient.
3. It is better than Selection Sort and Bubble Sort algorithms.
4. Its space complexity is less. Like bubble Sort, insertion sort also requires a single additional memory space.

5. **It is a stable sorting** technique, as it does not change the relative order of elements which are equal.

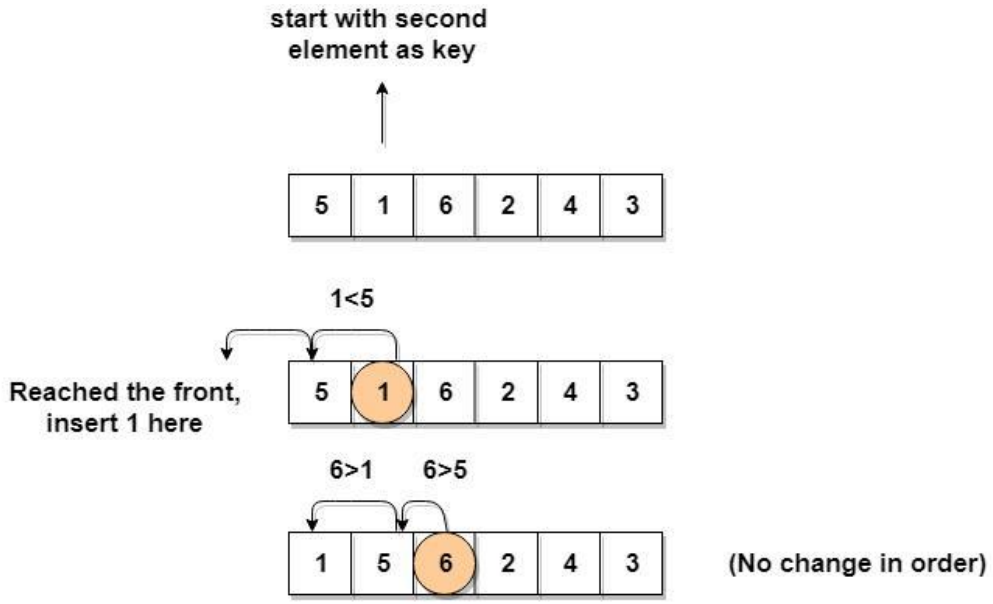


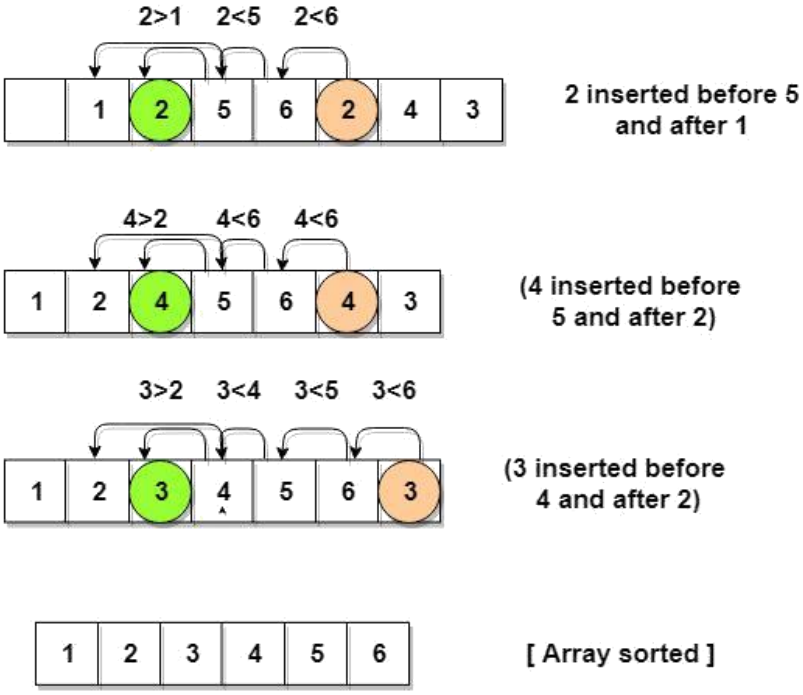
Working of Insertion Sort

1. We start by making the **second element** of the given array, i.e. element at index 1, the key. The key element here is the new card that we need to add to our existing sorted set of cards(remember the example with cards above).
2. We **compare the key element with the element(s) before it**, in this case, element at index 0:
    - If the key element is less than the first element, we insert the key element before the first element.
    - If the key element is greater than the first element, then we insert it after the first element.
3. Then, we make the **third** element of the array as **key** and will compare it with elements to it's left and insert it at the right position.
4. And we go on repeating this, until the array is sorted

Example : Input : {5, 1, 6, 2, 4, 3}

Below, we have a pictorial representation of how bubble sort will sort the given array





As you can see in the diagram above, after picking a key, we start iterating over the elements to the left of the key.

We continue to move towards left if the elements are greater than the key element and stop when we find the element which is less than the key element.

And, insert the key element after the element which is less than the key element.

Insertion sort is a very simple method to sort numbers in an ascending or descending order. This method **follows the incremental method**. It can be compared with the technique how cards are sorted at the time of playing a game.

The numbers, which are needed to be sorted, are known as **keys**. Here is the algorithm of the insertion sort method.

Algorithm: Insertion-Sort(A)

```
INSERTION-SORT(A)
   for i = 1 to n
        key ← A [i]
        j ← i – 1
          while j > = 0 and A[j] > key
                 A[j+1] ← A[j]
                 j ← j – 1
        End while
        A[j+1] ← key
  End for
```

Example

Unsorted list:	2	13	5	18	14

1st iteration:

Key = a[2] = 13

a[1] = 2 < 13

Swap, no swap	2	13	5	18	14

2nd iteration: Key = a[3] = 5 : a[2] = 13 > 5

Swap 5 and 13	2	5	13	18	14

Next, a[1] = 2 < 13

Swap, no swap	2	5	13	18	14

3rd iteration:

Key = a[4] = 18

a[3] = 13 < 18,

a[2] = 5 < 18,

a[1] = 2 < 18

Swap, no swap

2	5	13	18	14

4th iteration:

Key = a[5] = 14

a[4] = 18 > 14

Swap 18 and 14

2	5	13	14	18

Next, a[3] = 13 < 14,

a[2] = 5 < 14,

a[1] = 2 < 14

So, no swap

2	5	13	14	18

Finally,

the sorted list is

2	5	13	14	18

Note :

As we mentioned above that insertion sort is an efficient sorting algorithm, as it does not run on preset conditions using for loops, but instead it uses one while loop, which avoids extra steps once the array gets sorted.
Even though insertion sort is efficient, still, if we provide an already sorted array to the insertion sort algorithm, it will still execute the outer for loop, thereby requiring n steps to sort an already sorted array of n elements, which makes its **best case time complexity** a linear function of n.

If the given numbers are sorted, this algorithm runs in *O(n)* time.

~~**If the given numbers are in reverse order, the algorithm runs in $O(n^2)$ time.**~~

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **O(n)**

Average Time Complexity [Big-theta]: **$O(n^2)$**

Space Complexity: **O(1)**

Selection Sort Algorithm

Selection sort is conceptually the most simplest sorting algorithm.

This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted.

It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place.

How Selection Sort Works?

Following are the steps involved in selection sort(for sorting a given array in ascending order):

1. Starting from the first element, we search the smallest element in the array, and replace it with the element in the first position.
2. We then move on to the second position, and look for smallest element present in the subarray, starting from index 1, till the last index.
3. We replace the element at the **second** position in the original array, or we can say at the first position in the subarray, with the second smallest element.
4. This is repeated, until the array is completely sorted.

Example

Unsorted list:

5	2	1	4	3

1st iteration:

Smallest = 5

2 < 5, smallest = 2

1 < 2, smallest = 1

4 > 1, smallest = 1

3 > 1, smallest = 1

Swap 5 and 1

1	2	5	4	3

2nd iteration:

Smallest = 2

2 < 5, smallest = 2

2 < 4, smallest = 2

2 < 3, smallest = 2

No Swap

1	2	5	4	3

3rd iteration:

Smallest = 5

4 < 5, smallest = 4

3 < 4, smallest = 3

Swap 5 and 3

1	2	3	4	5

4th iteration:

Smallest = 4

4 < 5, smallest = 4

No Swap

1	2	3	4	5

Finally,

the sorted list is

1	2	3	4	5

Algorithm1:

Algorithm: Selection-Sort (A)

```
fori ← 1 to n-1 do
  min_index ← i;      // index of the key
  min_value ← A[i] // key
  for j ←i + 1 to n do
    if A[j] < min_value then
      min_index ← j
      min_value ← A[j]
  A[min_index] ← A [i]
  A[i] ← min_value
```

Moduler Algorithm for selection Sort:

```
FindMinIndex(Arr[], start, end)
        min_index = start

        FOR i from (start + 1) to end:
            IF Arr[i] < Arr[min_index]:
                min_index = i
            END of IF
        END of FOR

  Return min_index
```

Suppose, there are ‘n’ elements in the array. Therefore, at worst case, there can be n iterations in FindMinIndex() for start = 1 and end = n. We did not take any auxiliary space.
Therefore,

Time complexity: O(n)
Space complexity: O(1)

```
SelectionSort(Arr[], arr_size):
        FOR i from 1 to arr_size:
            min_index = FindMinIndex(Arr, i, arr_size)

            IF i != min_index:
                swap(Arr[i], Arr[min_index])
            END of IF
        END of FOR
```

Suppose, there are ‘n’ elements in the array. Therefore, at worst case, there can be n iterations in FindMinIndex() for start = 1 and end = n. No auxiliary space used.
Total iterations = (n – 1) + (n – 2) + . . . + 1 = (n * (n – 1)) / 2 = (n_2 – n) /
2 Therefore,

Time complexity: O(n^2)
Space complexity: O(1)

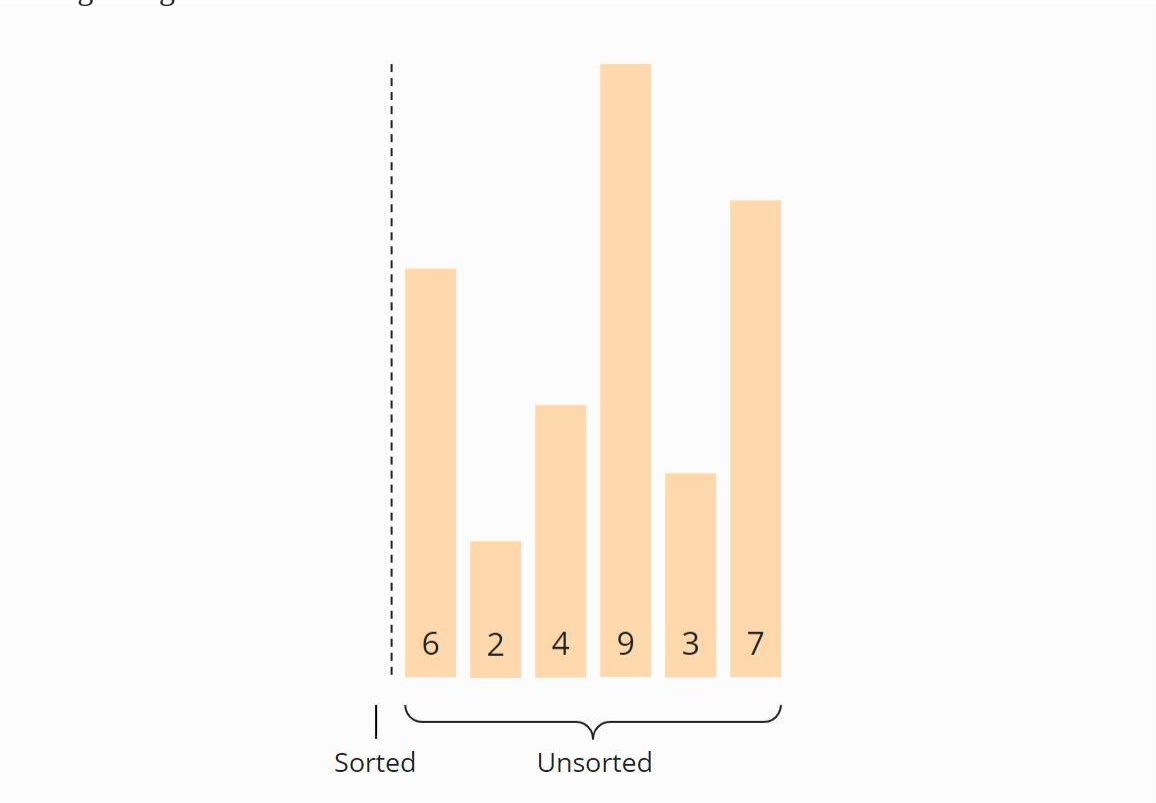
Complexity Analysis of Selection Sort

Visualization of Selection Sort

The algorithm can be explained most simply by an example. In the following steps, I show how to sort the array [6, 2, 4, 9, 3, 7] with Selection Sort:

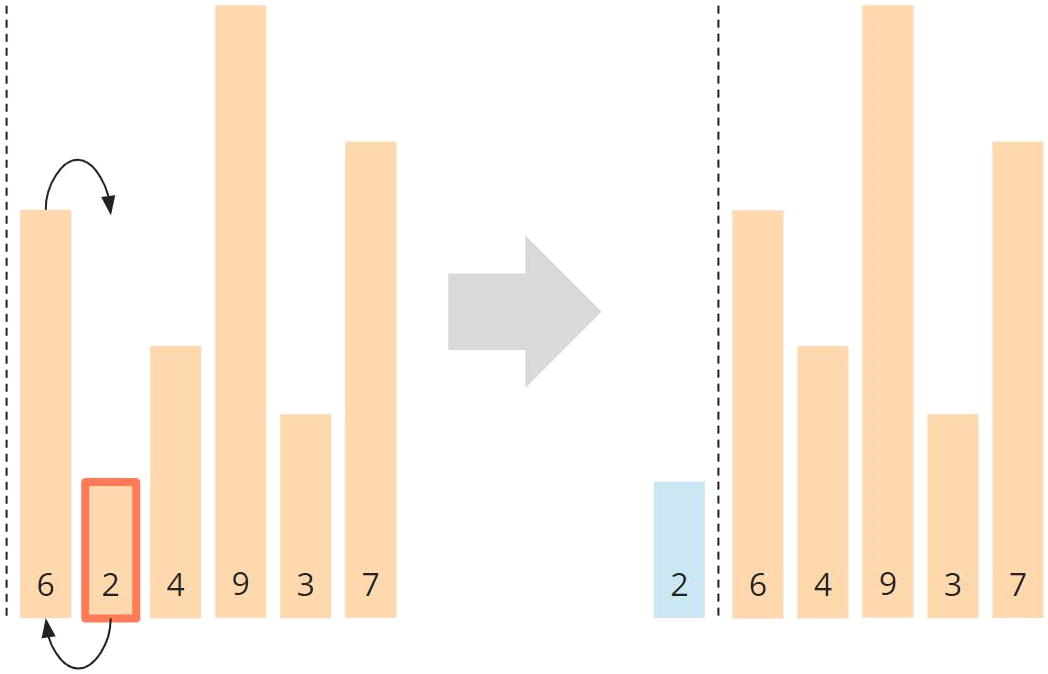
Step 1

We divide the array into a left, sorted part and a right, unsorted part. The sorted part is empty at the beginning:



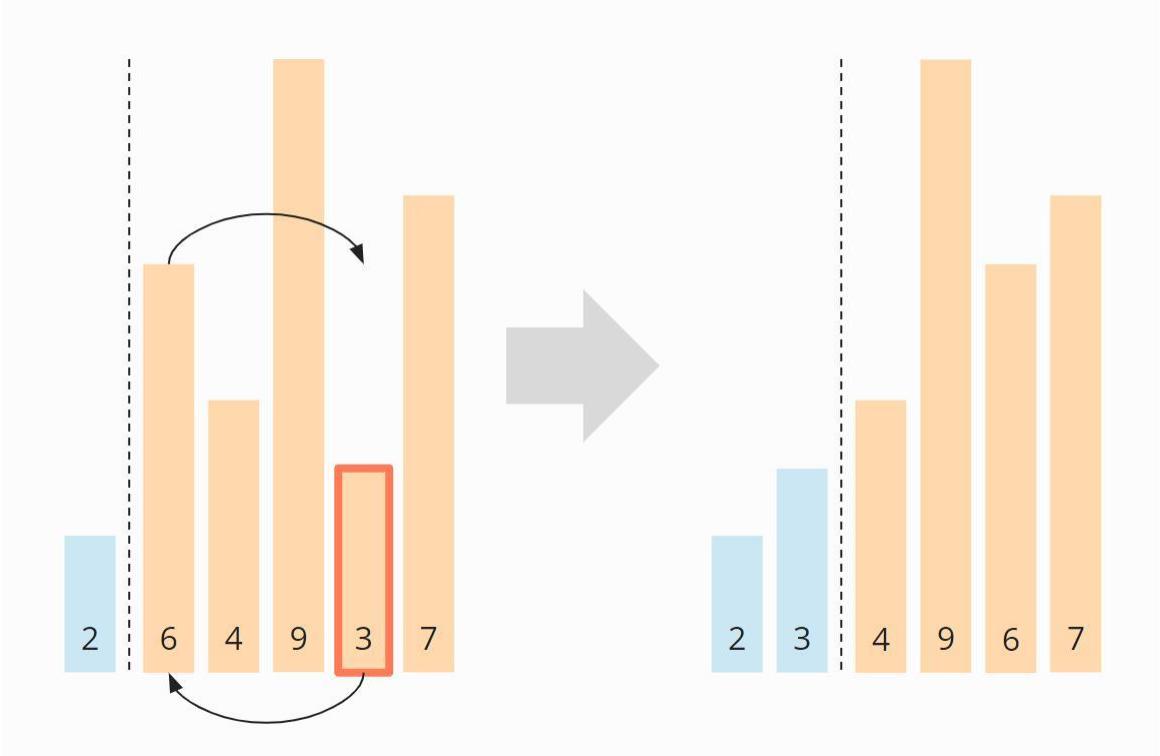
Step 2

We search for the smallest element in the right, unsorted part. To do this, we first remember the first element, which is the 6. We go to the next field, where we find an even smaller element in the 2. We walk over the rest of the array, looking for an even smaller element. Since we can't find one, we stick with the 2. We put it in the correct position by swapping it with the element in the first place. Then we move the border between the array sections one field to the right:



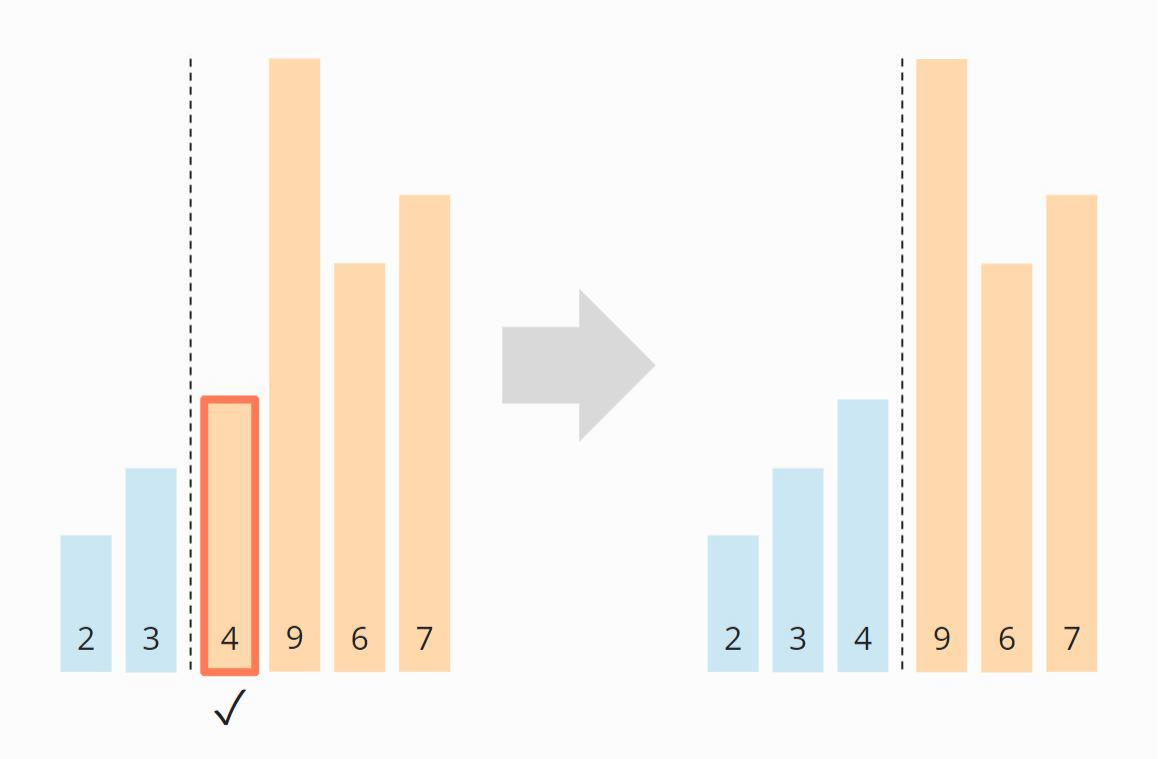
Step 3

We search again in the right, unsorted part for the smallest element. This time it is the 3; we swap it with the element in the second position:



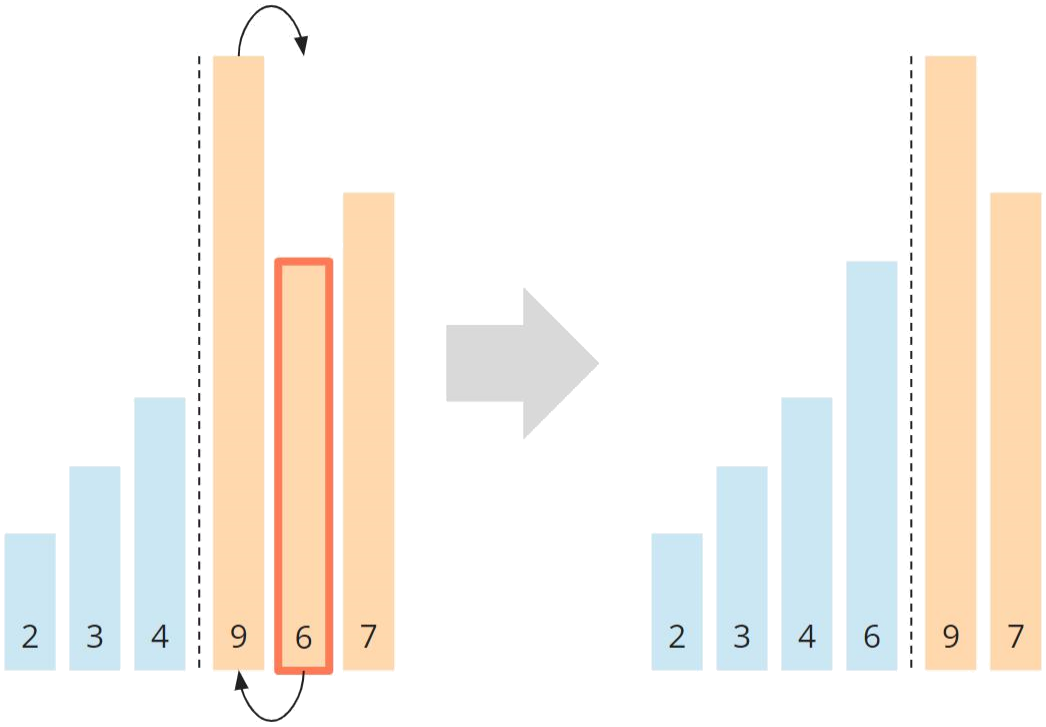
Step 4

Again we search for the smallest element in the right section. It is the 4, which is already in the correct position. So there is no need for swapping operation in this step, and we just move the section border:



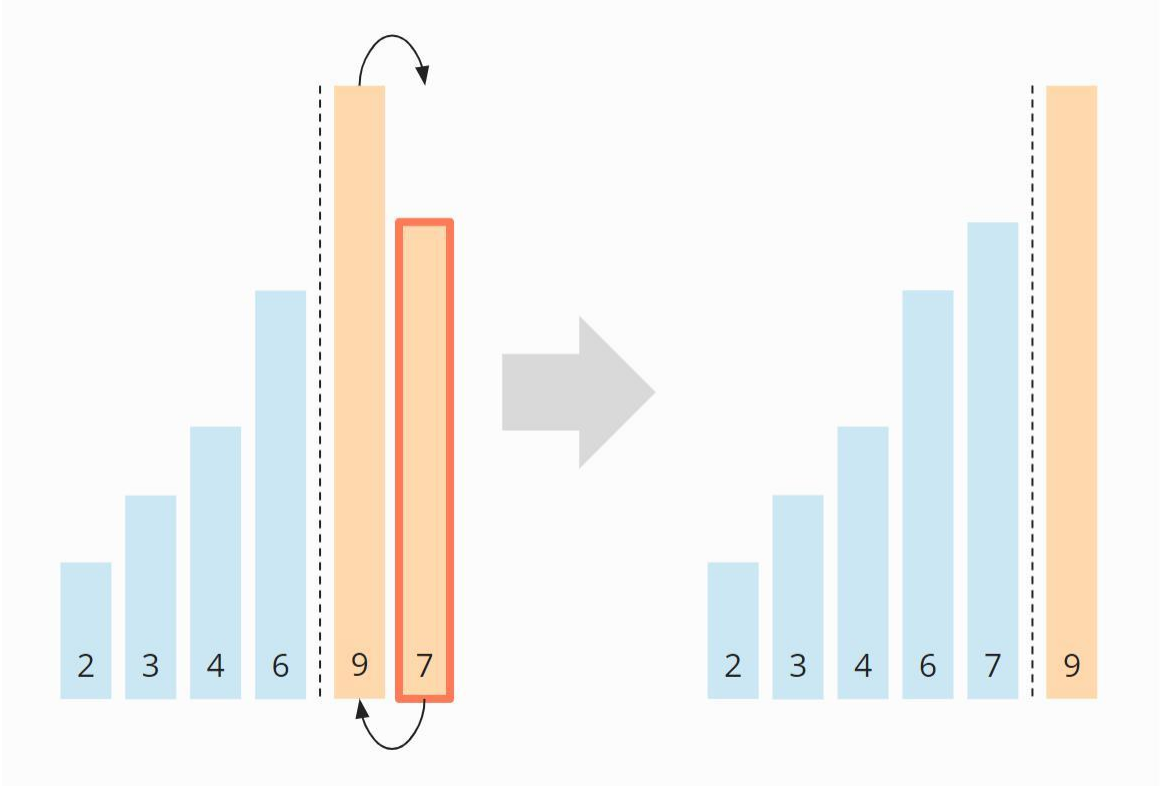
Step 5

As the smallest element, we find the 6. We swap it with the element at the beginning of the right part, the 9:



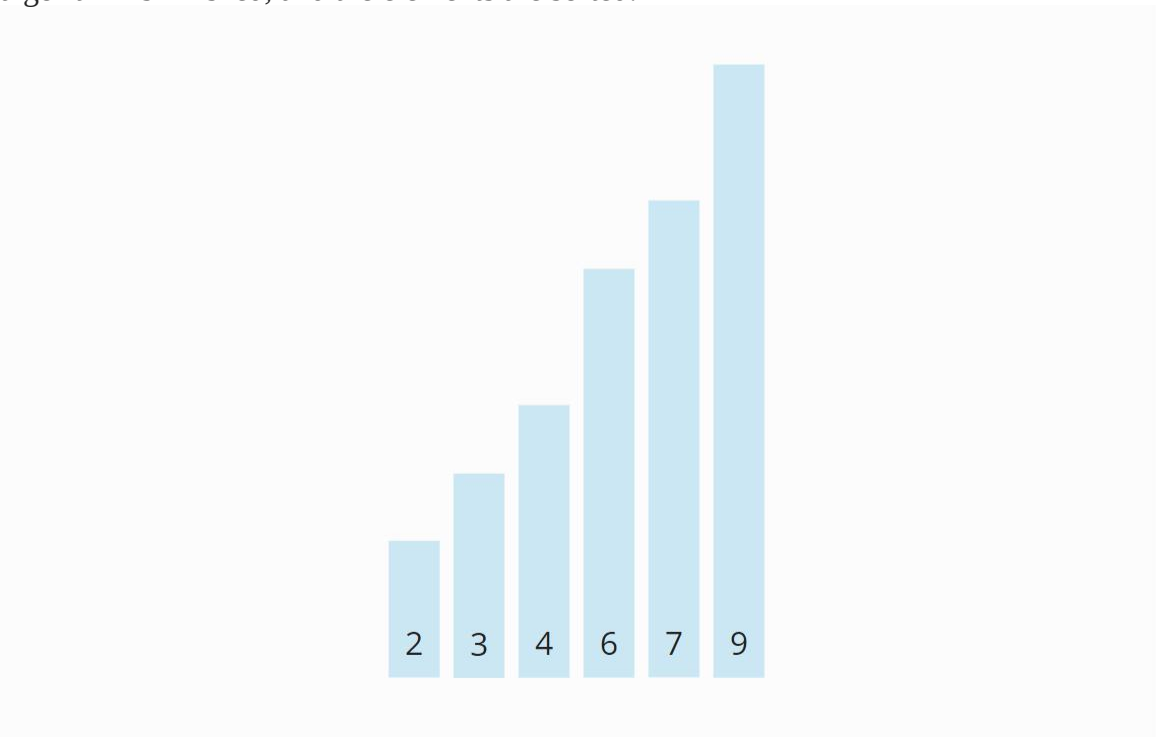
Step 6

Of the remaining two elements, the 7 is the smallest. We swap it with the 9:



Algorithm Finished

The last element is automatically the largest and, therefore, in the correct position. The algorithm is finished, and the elements are sorted:



Selection Sort requires two nested for loops to complete itself, one for loop is in the function selectionSort, and inside the first loop we are making a call to another function indexOfMinimum, which has the second(inner) for loop.

Hence for a given input size of n, following will be the time and space complexity for selection sort algorithm:

Worst Case Time Complexity [Big-O]: **$O(n^2)$**

Best Case Time Complexity [Big-omega]: **$\Omega(n^2)$**

Average Time Complexity [Big-theta]: **$\Theta(n^2)$**

Space Complexity: **O(1)**

Note:

The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Stability : The default implementation is not stable. However it can be made stable. For that we have to use stable selection sort.

In Place : Yes, it does not require extra space.

Comparison among Bubble Sort, Selection Sort and Insertion Sort (Self Study)

1. Bubble Sort

Time Complexity:

Best Case Sorted array as input. Or almost all elements are in proper place.
[**O(N)**]. **O(1)** swaps.

Worst Case: Reversely sorted / Very few elements are in proper place. [**$O(N_2)$**]
. **$O(N_2)$** swaps.

Average Case: [**$O(N_2)$**] . **$O(N_2)$** swaps.

Space Complexity: A temporary variable is used in swapping [auxiliary, **O(1)**]. Hence it is In-Place sort.

1. It is the simplest sorting approach.
2. Best case complexity is of **O(N)** [for optimized approach] while the array is sorted.
3. Stable sort: does not change the relative order of elements with equal keys.
4. In-Place sort.

1. Bubble sort is comparatively slower algorithm. (Number of Swaps are more)

2. Selection Sort

Best Case [**$O(N_2)$**]. Also **O(1)** swaps.
Worst Case: Reversely sorted, and when inner loop makes maximum comparison. [**$O(N_2)$**] .
Also **O(N)** swaps.

Average Case: [**$O(N_2)$**] . Also **O(N)** swaps.

1. It can also be used on list structures that make add and remove efficient, such as a linked list. Just remove the smallest element of unsorted part and end at the end of sorted part.
2. Best case complexity is of **O(N)** while the array is already sorted.
3. Number of swaps reduced. **O(N)** swaps in worst cases.
4. In-Place sort.

1. Time complexity in all cases is **$O(N_2)$**, no best case scenario.

3. Insertion Sort

Best Case Sorted array as input, [**O(N)**]. And **O(1)** swaps.
Worst Case: Reversely sorted, and when inner loop makes maximum comparison, [**$O(N_2)$**] .
And **$O(N_2)$** swaps.

Average Case: [**$O(N_2)$**] . And **$O(N_2)$** swaps.

Space Complexity: [auxiliary, **O(1)**]. In-Place sort.

Advantage:

1. It can be easily computed.
2. Best case complexity is of **O(N)** while the array is already sorted.
3. Number of swaps reduced than bubble sort.
4. For smaller values of N, insertion sort performs efficiently like other quadratic sorting algorithms.
5. Stable sort.
6. Adaptive: total number of steps is reduced for partially sorted array.
7. In-Place sort.

Disadvantage:

1. It is generally used when the value of N is small. For **larger values of N**, it is **inefficient**.

Time and Space Complexity:

SORTING ALGORITHM	TIME COMPLEXITY			SPACE COMPLEXITY
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	**O(N)**	**O(N_2)**	**O(N_2)**	**O(1)**
Selection Sort	**O(N_2)**	**O(N_2)**	**O(N_2)**	**O(1)**
Insertion Sort	**O(N)**	**O(N_2)**	**O(N_2)**	**O(1)**

What is the difference between Bubble Sort and Insertion Sort?

Even though both the bubble sort and insertion sort algorithms have average case time complexities of O(n2), bubble sort is almost all the time outperformed by the insertion sort. This is due to the number of swaps needed by the two algorithms (bubble sorts needs more swaps). But due to the simplicity of bubble sort, its code size is very small. Also there is a variant of insertion sort called the shell sort, which has a time complexity of O(n3/2), which would allow it to be used practically. Furthermore, insertion sort is very efficient for sorting “nearly sorted” lists, when compared with the bubble sort