Insertion Sort Algorithm

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?

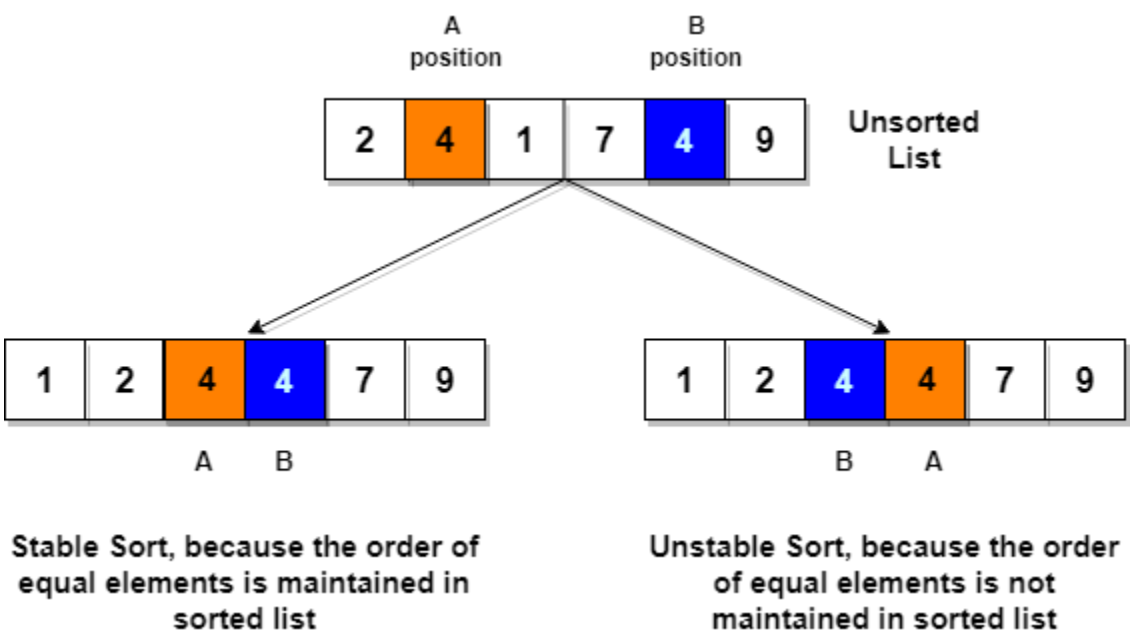
Well, you will have to go through each card from the starting or the back and find the right position for the new card, comparing it's value with each card. Once you find the right position, you will **insert** the card there.

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.

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.

Following are 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.



How Insertion Sort Works?

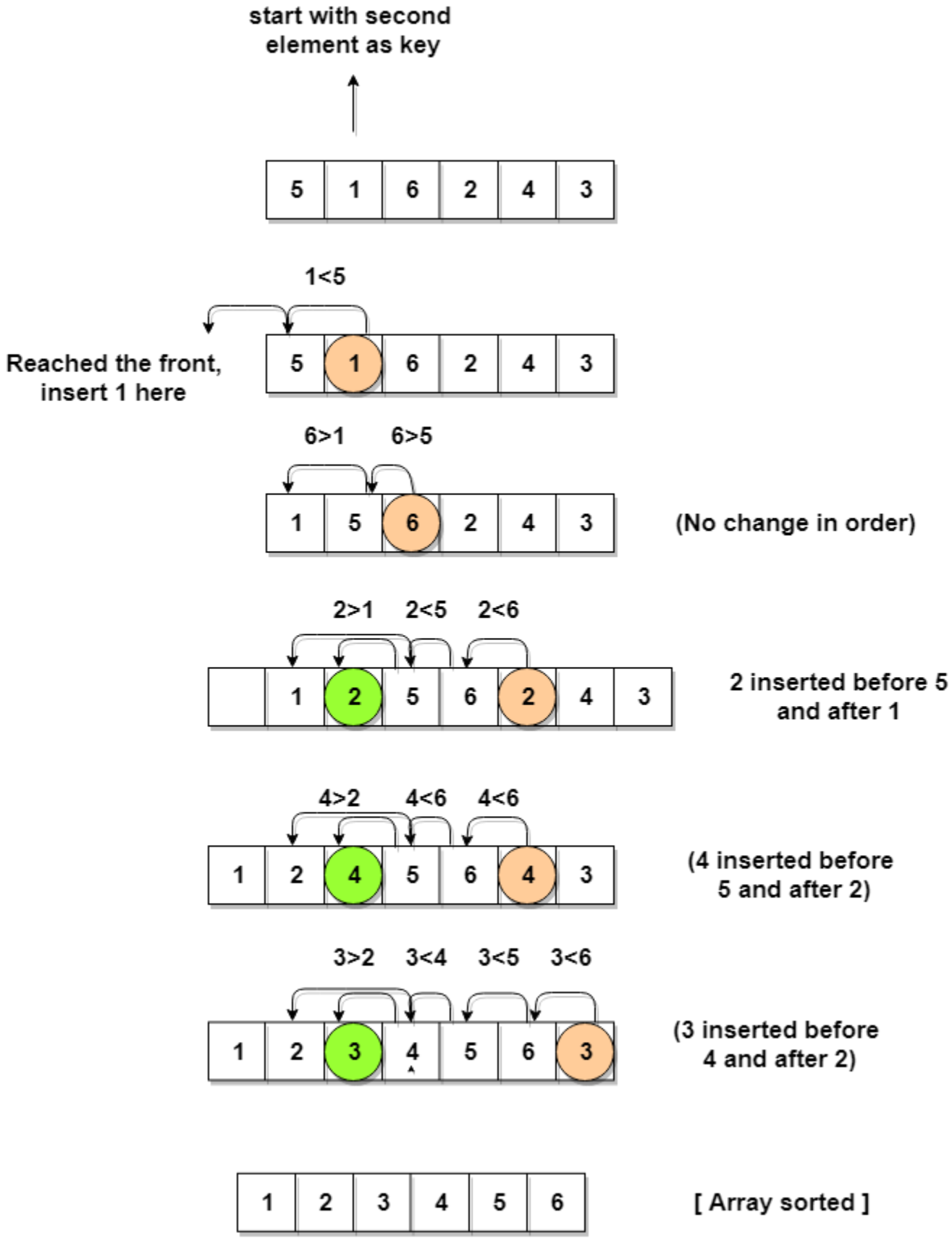
Following are the steps involved in insertion sort:

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.

Let's consider an array with values `{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.

Implementing Insertion Sort Algorithm

Below we have a simple implementation of Insertion sort in C++ language.

```
#include <stdlib.h>
#include <iostream>

using namespace std;

//member functions declaration
void insertionSort(int arr[], int length);
void printArray(int array[], int size);

// main function
int main()
{
     int array[5] = {5, 1, 6, 2, 4, 3};
     // calling insertion sort function to sort the array
     insertionSort(array, 6);
     return 0;
}

void insertionSort(int arr[], int length)
{
     int i, j, key;
     for (i = 1; i < length; i++)
     {
          j = i;
          while (j > 0 && arr[j - 1] > arr[j])
          {
               key = arr[j];
```

```
                arr[j] = arr[j - 1];

                arr[j - 1] = key;

                j--;

            }

        }

        cout << "Sorted Array: ";

        // print the sorted array

        printArray(arr, length);

}


// function to print the given array

void printArray(int array[], int size)

{

        int j;

        for (j = 0; j < size; j++)

        {

                cout <<" "<< array[j];

        }

        cout << endl;

}
```

Copy

```
Sorted Array:  1 2 3 4 5 6
```

Now let's try to understand the above simple insertion sort algorithm.

We took an array with **6** integers. We took a variable `key`, in which we put each element of the array, during each pass, starting from the **second** element, that is `a[1]`.

Then using the `while` loop, we iterate, until `j` becomes equal to **zero** or we find an element which is greater than `key`, and then we **insert** the `key` at that position.

We keep on doing this, until `j` becomes equal to **zero**, or we encounter an element which is smaller than the `key`, and then we stop. The current `key` is now at the right position.

We then make the next element as `key` and then repeat the same process.

In the above array, first we pick **1** as `key`, we compare it with **5**(element before 1), **1** is smaller than **5**, we insert **1** before **5**. Then we pick **6** as `key`, and compare it with **5** and **1**, no shifting in

position this time. Then **2** becomes the `key` and is compared with **6** and **5**, and then **2** is inserted after **1**. And this goes on until the complete array gets sorted.

Complexity Analysis of Insertion Sort

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`.

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)**
