## SORTING

- Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order.
- A collection of records called a list where every record has one or more fields.
- The fields which contain a unique value for each record is termed as the ***key*** field.
- For example, a phone number directory can be thought of as a list where each record has three fields - 'name' of the person, 'address' of that person, and their 'phone numbers’.
- Being unique phone number can work as a key to locate any record in the list.
- Sorting is the operation performed to arrange the records of a table or list in some order according to some specific ordering criterion.
- Sorting is performed according to some key value of each record.
- The records are either sorted either numerically or alphanumerically.
- The records are then arranged in ascending or descending order depending on the numerical value of the key. Here is an example, where the sorting of a lists of marks obtained by a student in any particular subject of a class.

**Categories of Sorting Techniques:**

The techniques of sorting can be divided into two categories. These are:

- Internal Sorting
- External Sorting

**Internal Sorting:**

If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.

internal Sort is any data sorting process that takes place entirely within the main memory of computer

example :**bubble sort , I**nsertion sort

**External Sorting:**

When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc, then external sorting methods are performed.

external sorting is a term for a class of sorting algorithms that can handle massive amounts of data .

example : **external merge sort** algorithm

## Complexity of Sorting Algorithm

The complexity of sorting algorithm calculates the running time of a function in which 'n' number of items are to be sorted.

The choice for which sorting method is suitable for a problem depends on several dependency configurations for different problems.

The most noteworthy of these considerations are:

- The length of time spent by the programmer in programming a specific sorting program
- Amount of machine time necessary for running the program
- The amount of memory necessary for running the program

## The Efficiency of Sorting Techniques

To get the amount of time required to sort an array of 'n' elements by a particular method, the normal approach is to analyze the method to find the number of comparisons (or exchanges) required by it. Most of the sorting techniques are data sensitive, and so the metrics for them depends on the order in which they appear in an input array.

Various sorting techniques are analyzed in various cases and named these cases as follows:

- Best case
- Worst case
- Average case

Hence, the result of these cases is often a formula giving the average time required for a particular sort of size 'n.' Most of the sort methods have time requirements that range from O(nlog n) to O($n^2$).

## Types of Sorting:

Comparision based sorting:

- Bubble Sort
- Selection Sort
- Merge Sort
- Insertion Sort
- Quick Sort
- Shell Sort

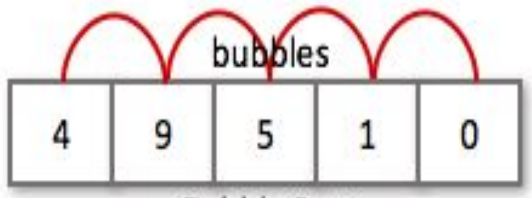
Non-comparision based Sorting:

- Counting Sort
- Radix Sort
- Bucket Sort

## Bubble Sort:

Bubble sort compares the value of first element with the immediate next element and swaps according to the requirement and goes till the last element.

This iteration repeats for (N - 1) times/steps where N is the number of elements in the list.



Bubble Sort

Compare and swapping two elements like small soap bubbles and hence the name given as **bubble sort.**

The **bubble sort** makes multiple passes through a list.

It compares adjacent items and exchanges those that are out of order.

Each pass through the list places the next largest value in its proper place.

In essence, each item "bubbles" up to the location where it belongs.

## What is Bubble Sort and how is it associated with Algorithms?

Bubble Sort is a sorting algorithm, which is commonly used in computer science.
Bubble Sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order.

## Bubble Sort Algorithm: Steps on how it works:

1. In an unsorted array of 5 elements, start with the first two elements and sort them in ascending order.

   (Compare the element to check which one is greater).

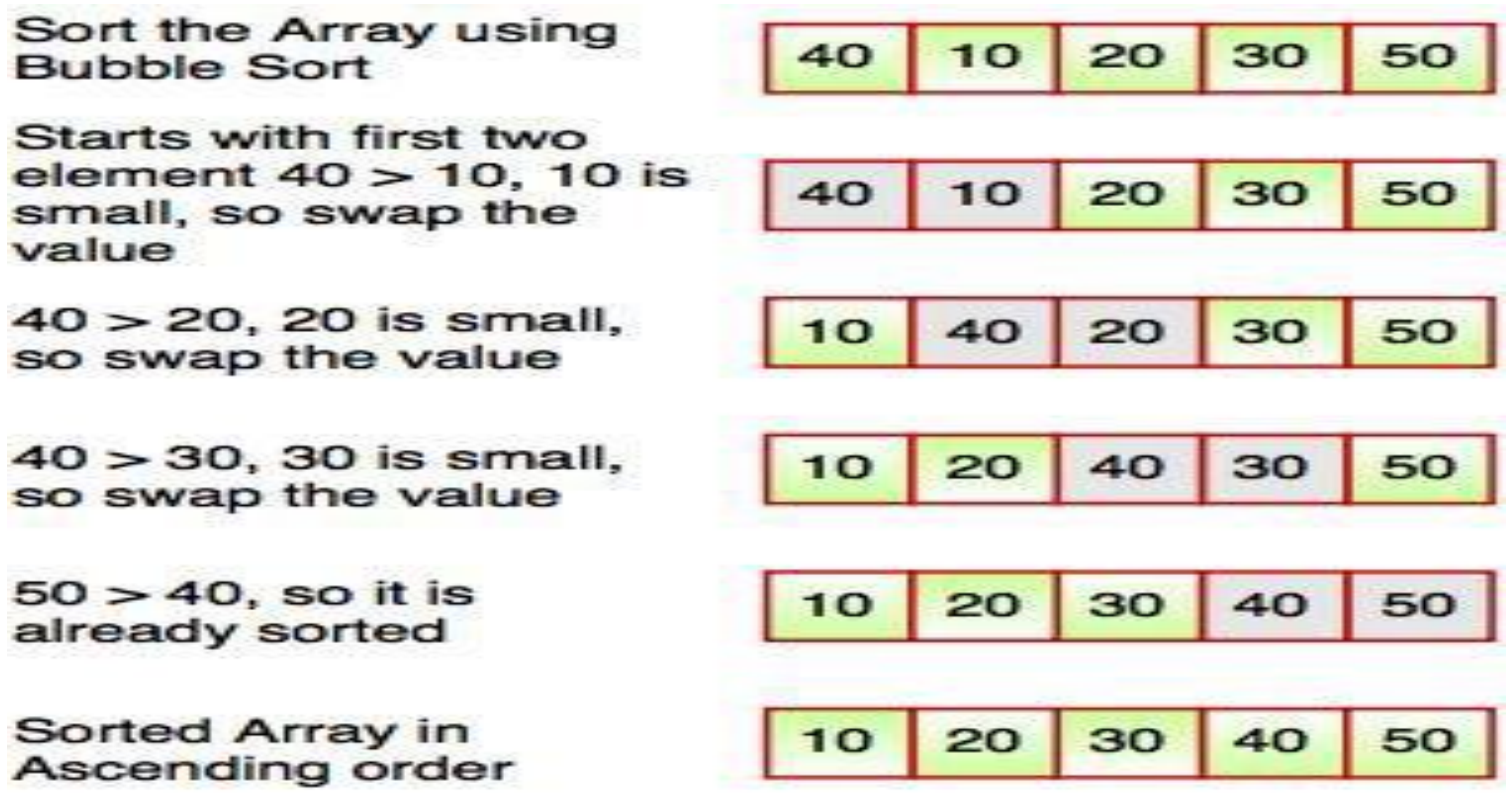
2. Compare the second and third element to check which one is greater, and sort them in ascending order.

3. Compare the third and fourth element to check which one is greater, and sort them in ascending order.

4. Compare the fourth and fifth element to check which one is greater, and sort them in ascending order.

5. Repeat steps 1–5 until no more swaps are required.

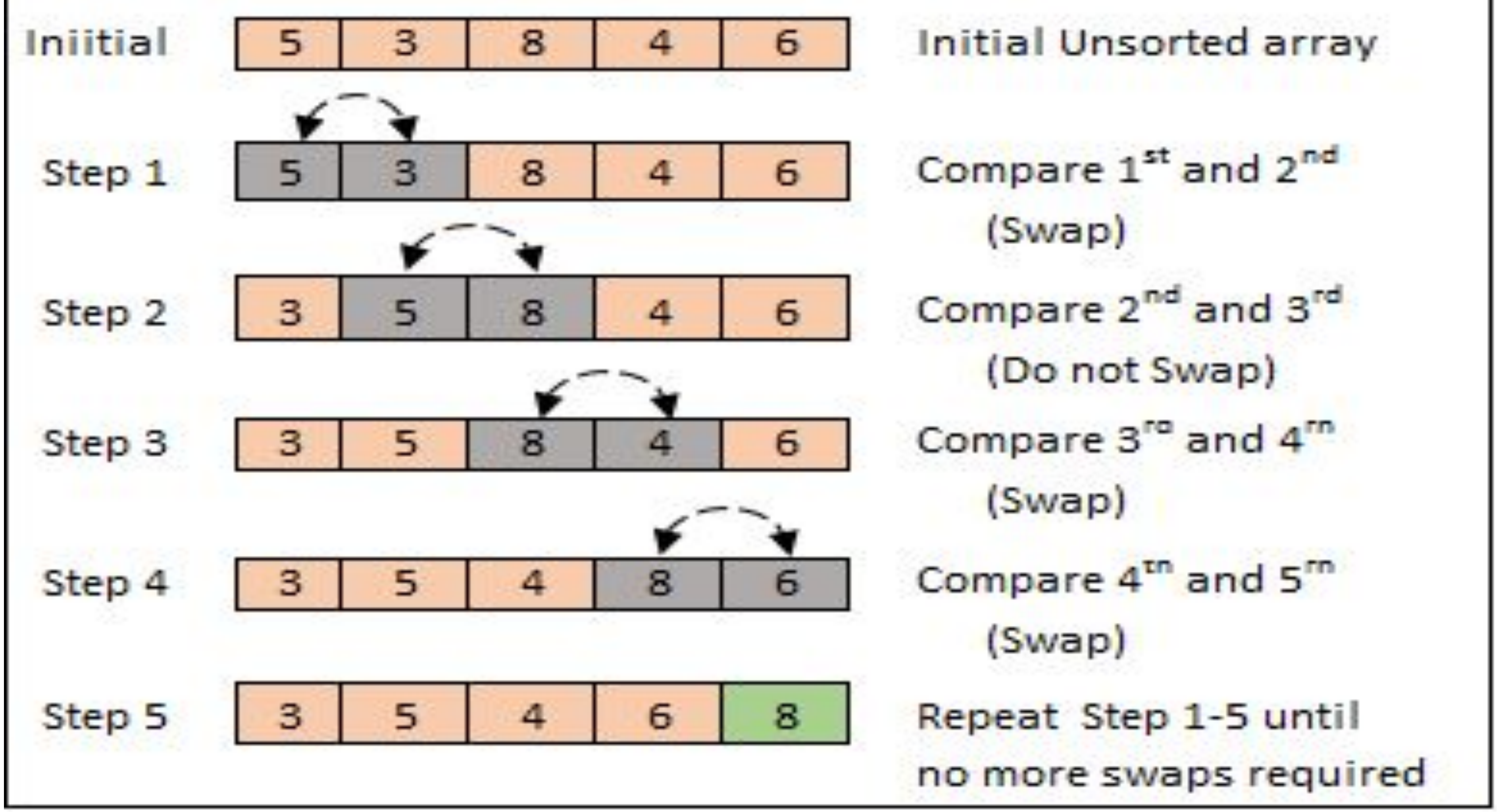
Below is an image of an array, which needs to be sorted. We will use the Bubble Sort Algorithm, to sort this array:

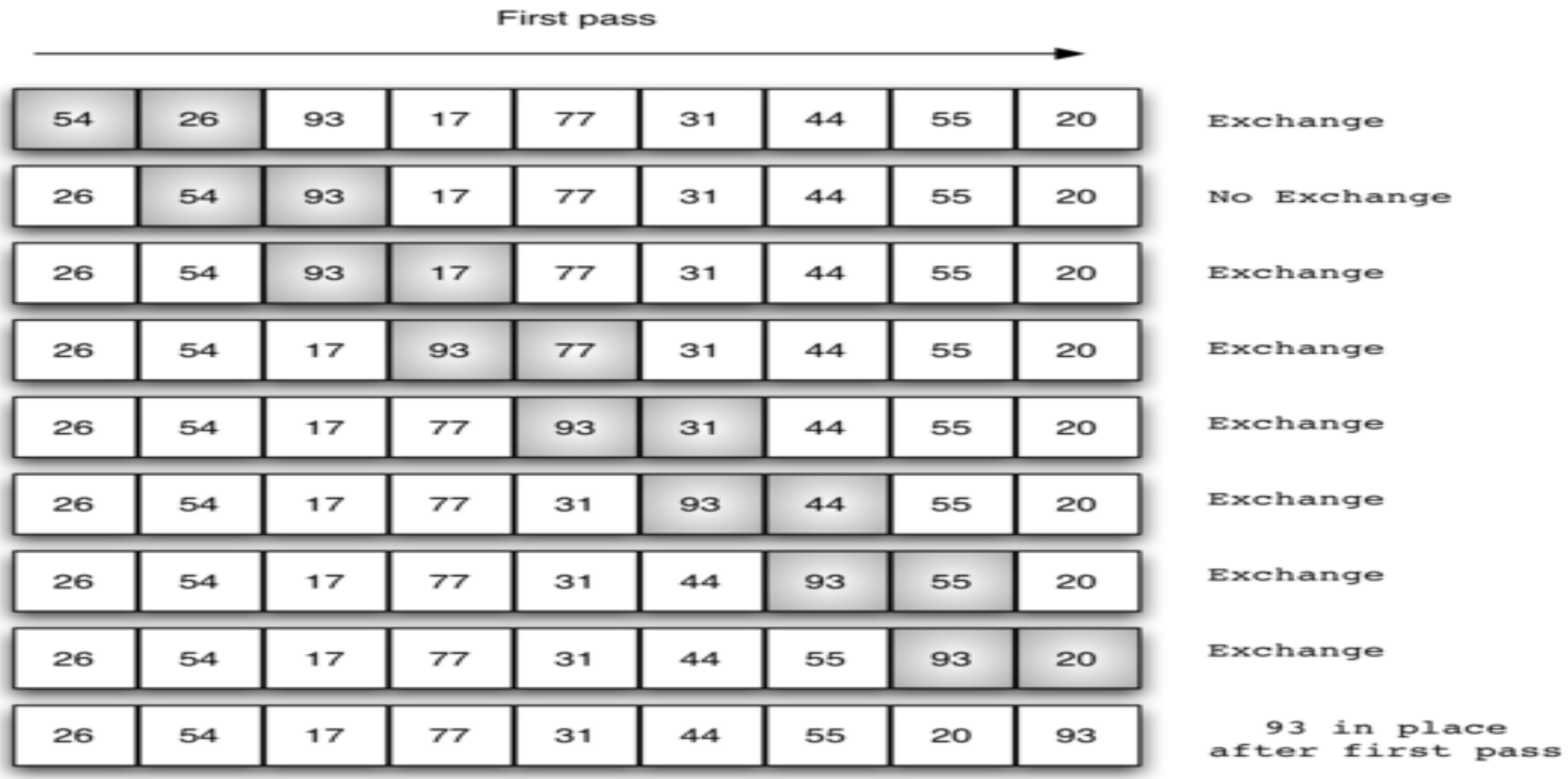


| | | | | | |
|---|---|---|---|---|---|
| Sort the Array using Bubble Sort | 40 | 10 | 20 | 30 | 50 |
| Starts with first two element 40 > 10, 10 is small, so swap the value | 40 | 10 | 20 | 30 | 50 |
| 40 > 20, 20 is small, so swap the value | 10 | 40 | 20 | 30 | 50 |
| 40 > 30, 30 is small, so swap the value | 10 | 20 | 40 | 30 | 50 |
| 50 > 40, so it is already sorted | 10 | 20 | 30 | 40 | 50 |
| Sorted Array in Ascending order | 10 | 20 | 30 | 40 | 50 |

Fig. Working of Bubble Sort

## Bubble sort example





shows the first pass of a bubble sort. The shaded items are being compared to see if they are out of order. If there are n items in the list, then there are n−1 pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.

At the start of the second pass, the largest value is now in place. There are n−1 items left to sort, meaning that there will be n−2 pairs.

Since each pass places the next largest value in place, the total number of passes necessary will be n−1. After completing the n−1 passes, the smallest item must be in the correct position with no further processing required.

ActiveCode1 shows the complete bubble Sort function. It takes the list as a parameter, and modifies it by exchanging items as necessary.

The exchange operation, sometimes called a "swap," is slightly different in Python than in most other programming languages.

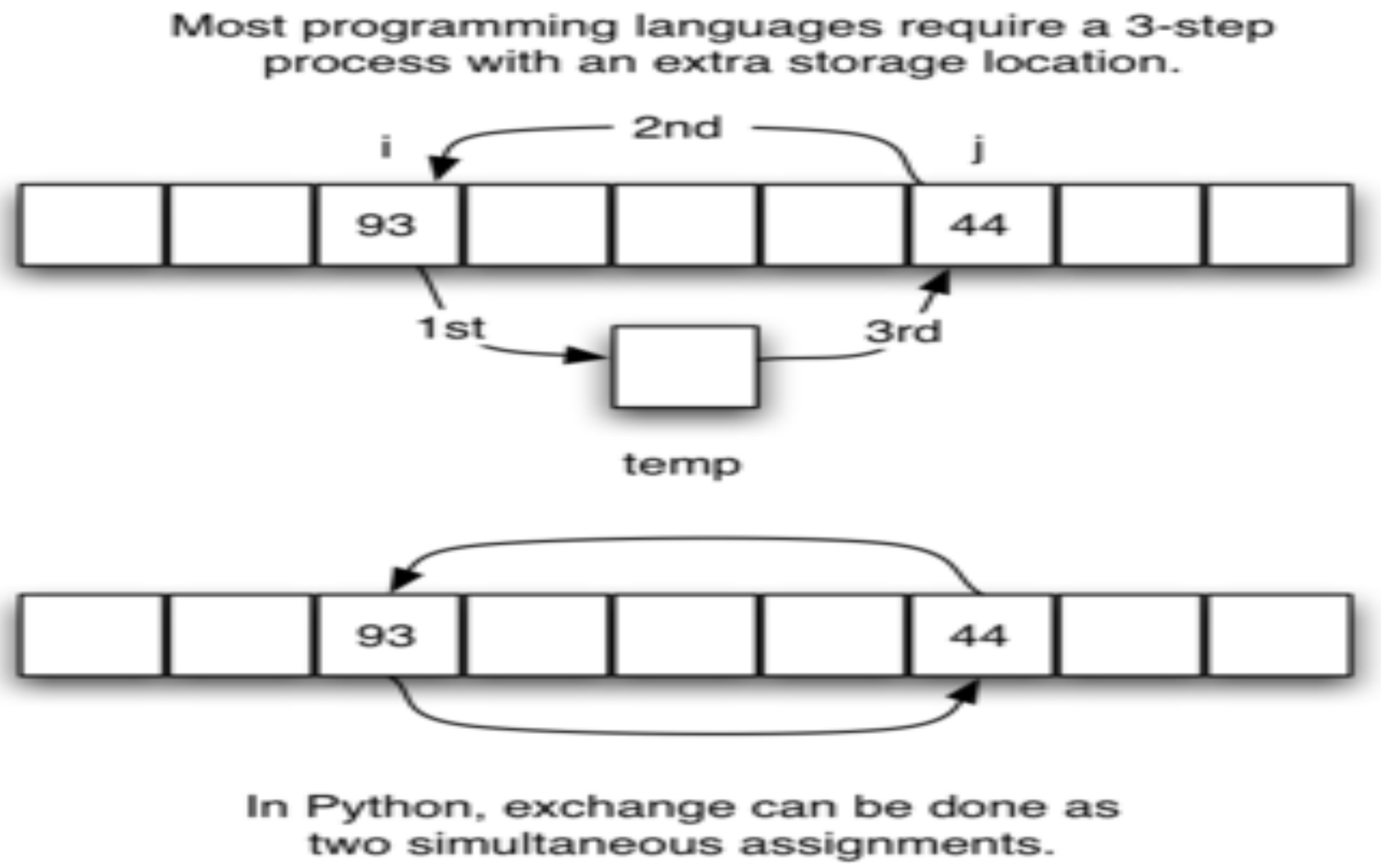
Typically, swapping two elements in a list requires a temporary storage location (an additional memory location). A code fragment such as

```
temp = alist[i]
alist[i] = alist[j]
alist[j] = temp
```

will exchange the *ith* and *jth* items in the list. Without the temporary storage, one of the values would be overwritten.

n Python, it is possible to perform simultaneous assignment. The statement a,b=b,a will result in two assignment statements being done at the same time (see Figure 2). Using simultaneous assignment, the exchange operation can be done in one statement.

Lines 5-7 in ActiveCode 1 perform the exchange of the i and (i+1)th items using the three–step procedure described earlier. Note that we could also have used the simultaneous assignment to swap the items.


Most programming languages require a 3-step process with an extra storage location.
2nd
i
j
93
44
1st
3rd
temp
93
44
In Python, exchange can be done as two simultaneous assignments.

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp


alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

To analyze the bubble sort, we should note that regardless of how the items are arranged in the initial list, $n-1$ n−1 passes will be made to sort a list of size *n*. Table 1 shows the number of comparisons for each pass. The total number of comparisons is the sum of the first $n-1$ n−1 integers. Recall that the sum of the first *n* integers is $\frac{1}{2}n^{2}+\frac{1}{2}n$ 12n2+12n.

The sum of the first $n-1$ n−1 integers is $\frac{1}{2}n^{2}+\frac{1}{2}n-n$ 12n2+12n−n, which is $\frac{1}{2}n^{2}-\frac{1}{2}n$ 12n2−12n.

This is still $O(n^{2})$ O(n2) comparisons. In the best case, if the list is already ordered, no exchanges will be made. However, in the worst case, every comparison will cause an exchange. On average, we exchange half of the time.

Table 1: Comparisons for Each Pass of Bubble Sort

| **Pass** | **Comparisons** |
|---|---|
| 1 | n−1n−1 |
| 2 | n−2n−2 |
| 3 | n−3n−3 |
| … | … |
| n−1n−1 | 1 |

A bubble sort is often considered the most inefficient sorting method since it must exchange items before the final location is known.

These “wasted” exchange operations are very costly. However, because the bubble sort makes passes through the entire unsorted portion of the list, it has the capability to do something most sorting algorithms cannot. In particular, if during a pass there are no exchanges, then we know that the list must be sorted.

A bubble sort can be modified to stop early if it finds that the list has become sorted.

This means that for lists that require just a few passes, a bubble sort may have an advantage in that it will recognize the sorted list and stop. ActiveCode 2 shows this modification, which is often referred to as the **short bubble**.

**Characteristics of Bubble Sort:**

•Large values are always sorted first.

•It only takes one iteration to detect that a collection is already sorted.

•The best time complexity for Bubble Sort is O(n). The average and worst time complexity is O($n^2$).

•The space complexity for Bubble Sort is O(1), because only single additional memory space is required.

**Efficiency and Big(O)**

Bubble sort of N elements can take (N - 1) steps and (N -1) iterations in each steps.

Thus resultant is (N - 1)*(N - 1). This sorting algorithm is not however the best in performance when count of the elements are large.

Time complexities of bubble sort is O(N^2) [Square of N].

This sorting is well suited for small number of elements and it is easy the implement in C or any other programming languages.

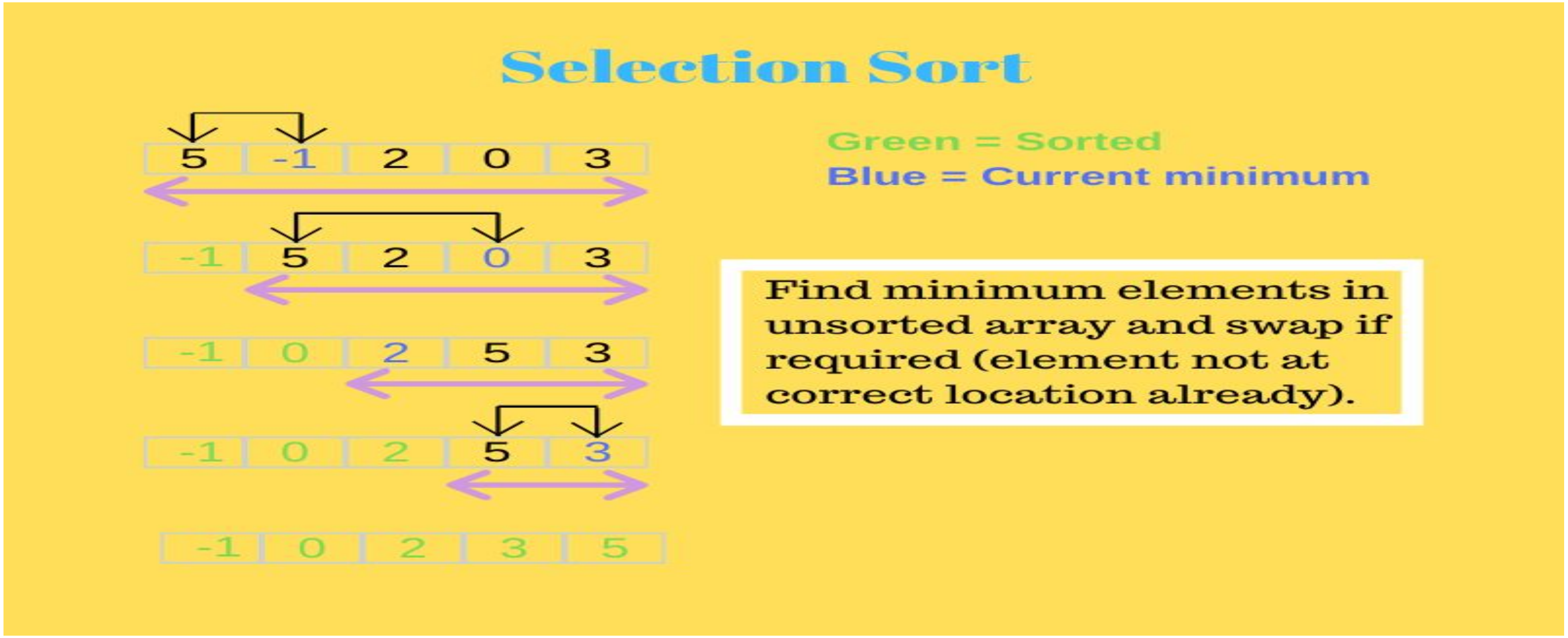
Suppose you have the following list of numbers to sort: <br> [19, 1, 9, 7, 3, 10, 13, 15, 8, 12] which list represents the partially sorted list after three complete passes of bubble sort?

A. [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]
B. [1, 3, 7, 9, 10, 8, 12, 13, 15, 19]
C. [1, 7, 3, 9, 10, 13, 8, 12, 15, 19]
D. [1, 9, 19, 7, 3, 10, 13, 15, 8, 12]

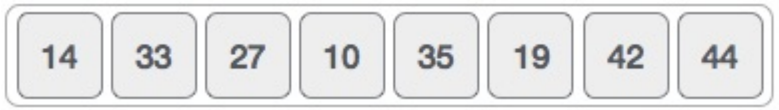
## Selection Sort

In this, at first, the smallest element is sent to the first position.

Then, the next smallest element is searched in the remaining array and is placed at the second position. This goes on until the algorithm reaches the final element and places it in the right position.

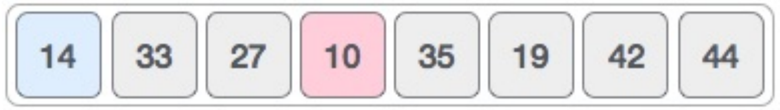

Selection Sort
Green = Sorted
Blue = Current minimum
Find minimum elements in unsorted array and swap if required (element not at correct location already).
5 -1 2 0 3
-1 5 2 0 3
-1 0 2 5 3
-1 0 2 5 3
-1 0 2 3 5

How Selection Sort Works?
Consider the following depicted array as an example.

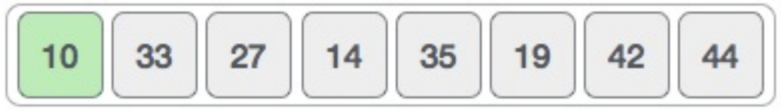


For the first position in the sorted list, the whole list is scanned sequentially.

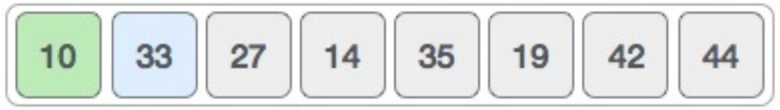
The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

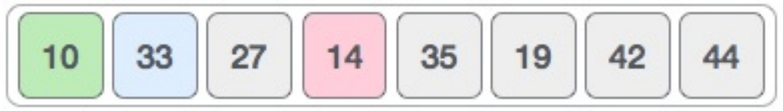


For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

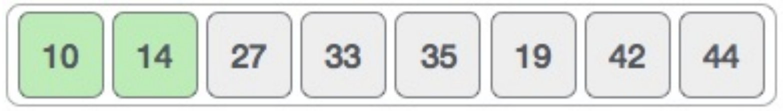


- Selection sort is a simple sorting algorithm.
- This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end.
- Initially, the sorted part is empty and the unsorted part is the entire list.
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array.
- This process continues moving unsorted array boundary by one element to the right.
- This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n2), where n is the number of items.

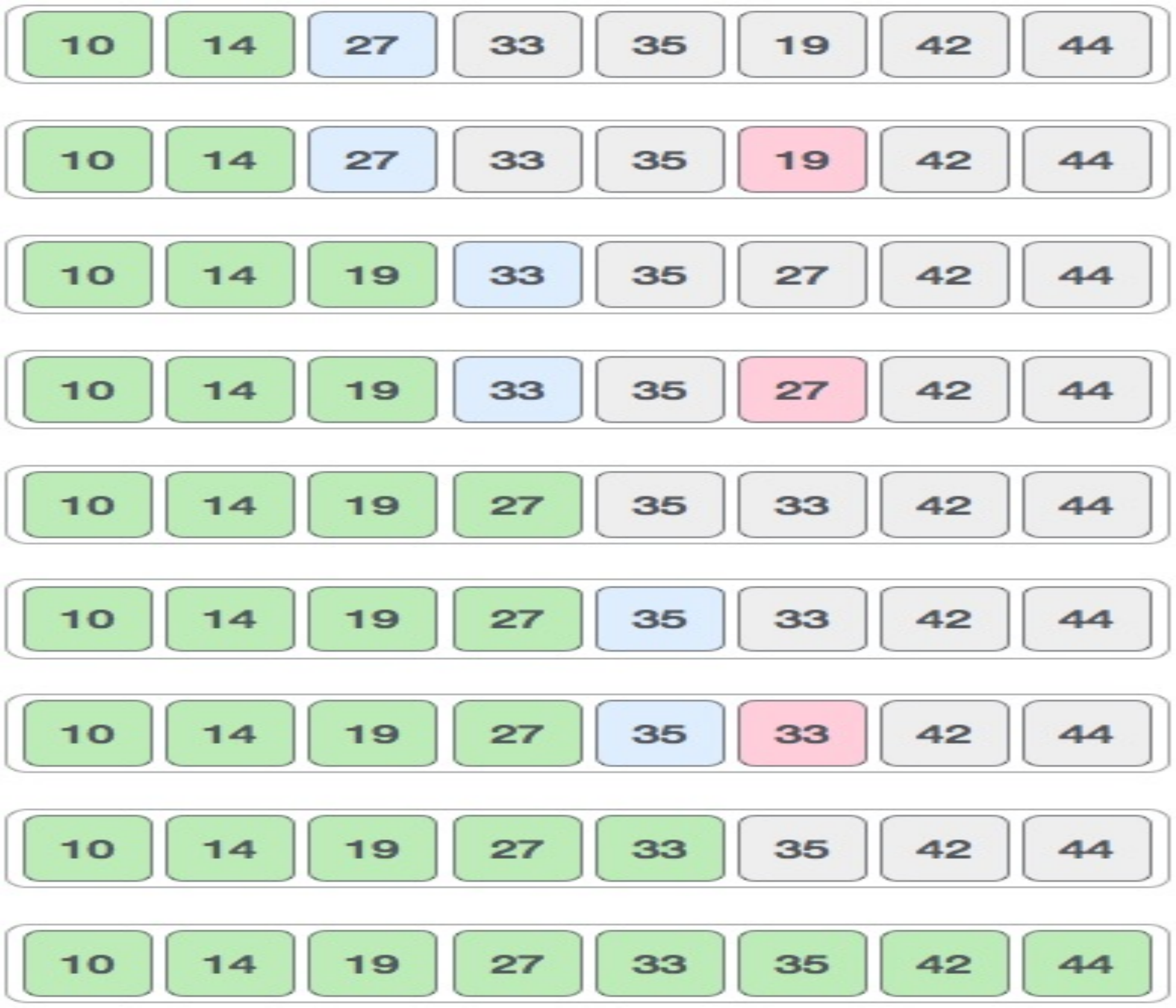
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.
Following is a pictorial depiction of the entire sorting process −



## Algorithm

Step 1 − Set MIN to location 0

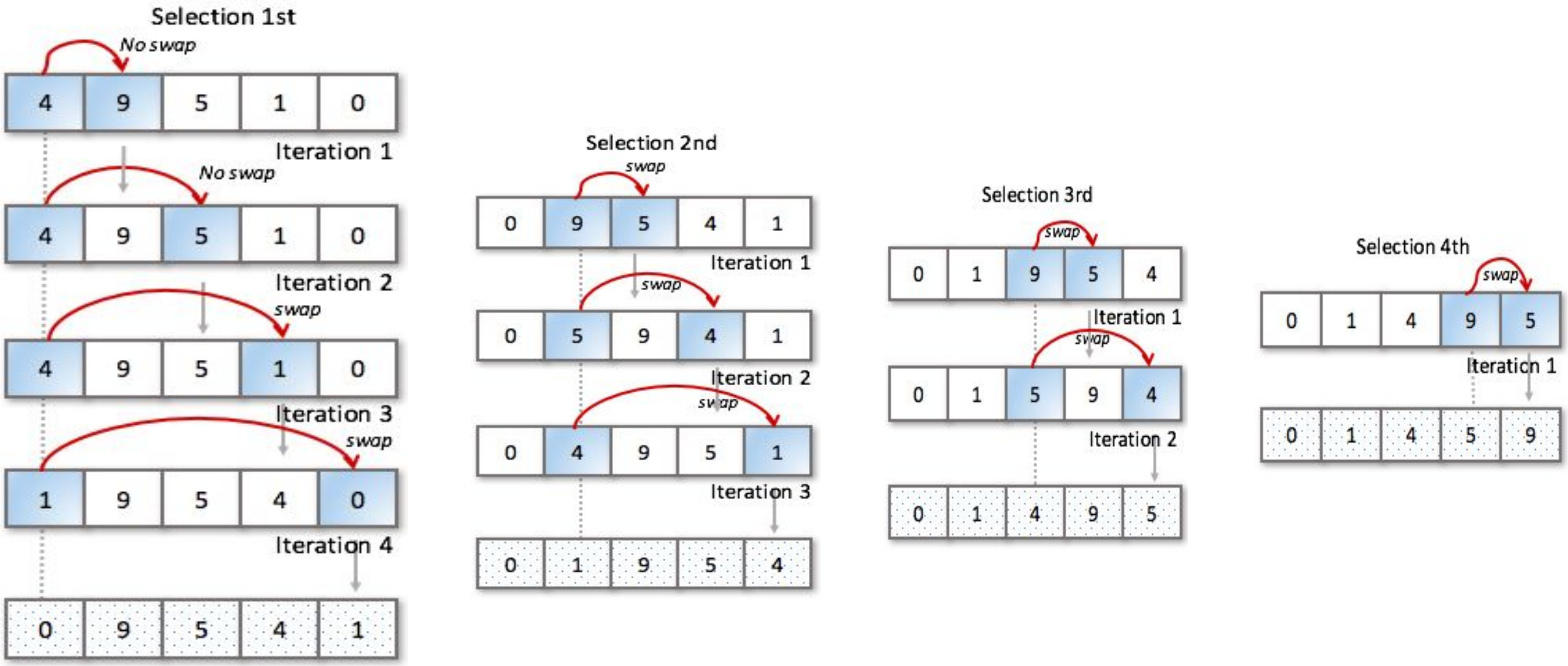
Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted

## Steps and Iterations



**Selection sort Big(o)**

Selection sort of N elements can take (N - 1) steps and (N - 1) iterations in each steps. Thus resultant is (N-1)*(N-1).

This sorting algorithm is not however the best in performance when count of the elements are large.

Time complexities of Selection sort is Big(o) = N^2. This sorting is well suited for small number of elements and it is easy the implement in C or any other programming languages.

**Python program for selection sort implementation**

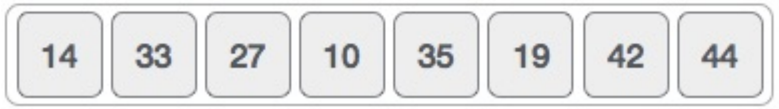
```
import sys

X = [6, 25, 10, 28, 11]

for i in range(len(X)):
        min_idx = i

    for j in range(i+1, len(X)):
        if X[min_idx] > X[j]:
            min_idx = j
    X[i], X[min_idx] = X[min_idx], X[i]

print (“The sorted array is”)

for i in range(len(X)):
    print(“%d” %X[i]),
```

## Insertion Sort

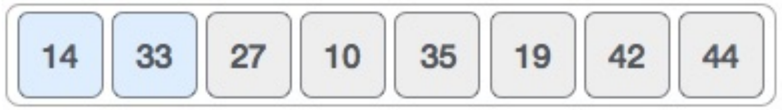
- This is an in-place comparison-based sorting algorithm.
- Here, a sub-list is maintained which is always sorted.
- For example, the lower part of an array is maintained to be sorted.
- An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, **insertion sort**.
- The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array).
- This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$, where **n** is the number of items.

## How Insertion Sort Works?

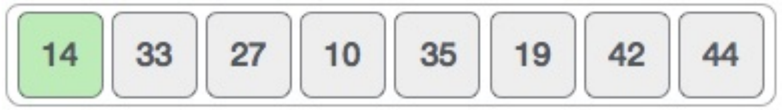
We take an unsorted array for our example.



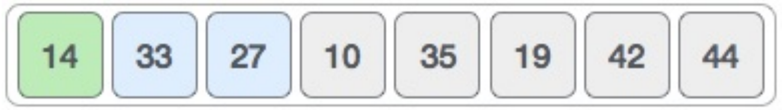
Insertion sort compares the first two elements.



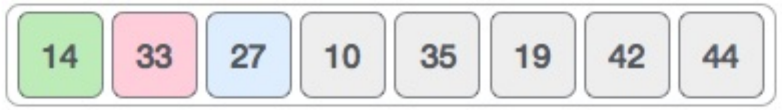
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



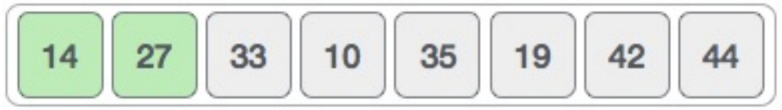
Insertion sort moves ahead and compares 33 with 27.



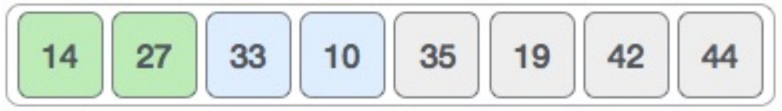
And finds that 33 is not in the correct position.



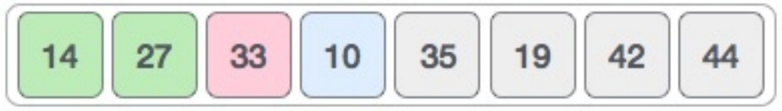
It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.



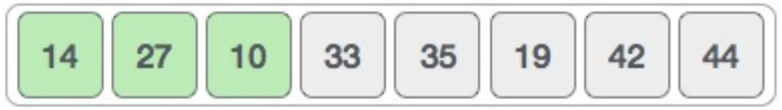
By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.



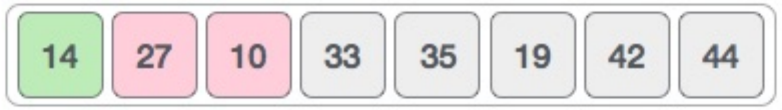
These values are not in a sorted order.



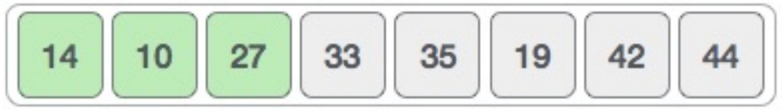
So we swap them.



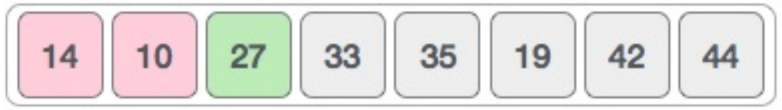
However, swapping makes 27 and 10 unsorted.



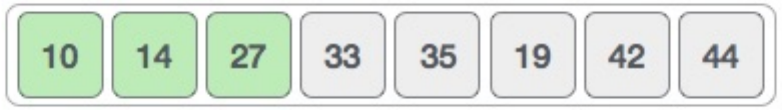
Hence, we swap them too.



Again we find 14 and 10 in an unsorted order.



We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.



This process goes on until all the unsorted values are covered in a sorted sub-list.

**Algorithm- sample 1**

To sort an array of size n in ascending order:

1: Iterate from arr[1] to arr[n] over the array.

2: Compare the current element (key) to its predecessor.

3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

**Algorithm- sample 2**

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

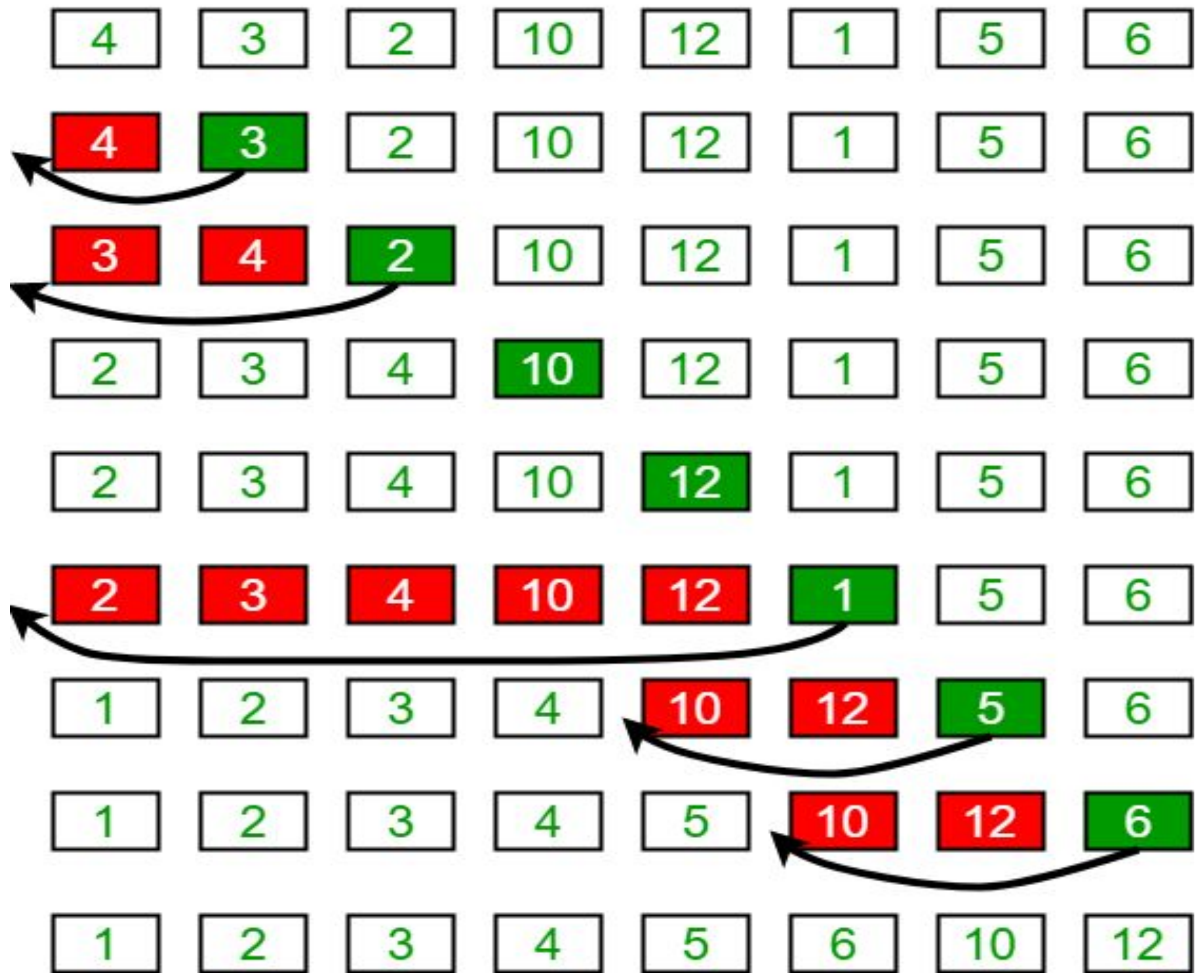
Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the  value to be sorted

Step 5 − Insert the value

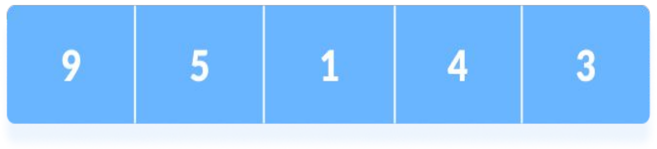
Step 6 − Repeat until list is sorted

## Insertion Sort Execution Example



## How Insertion Sort Works?

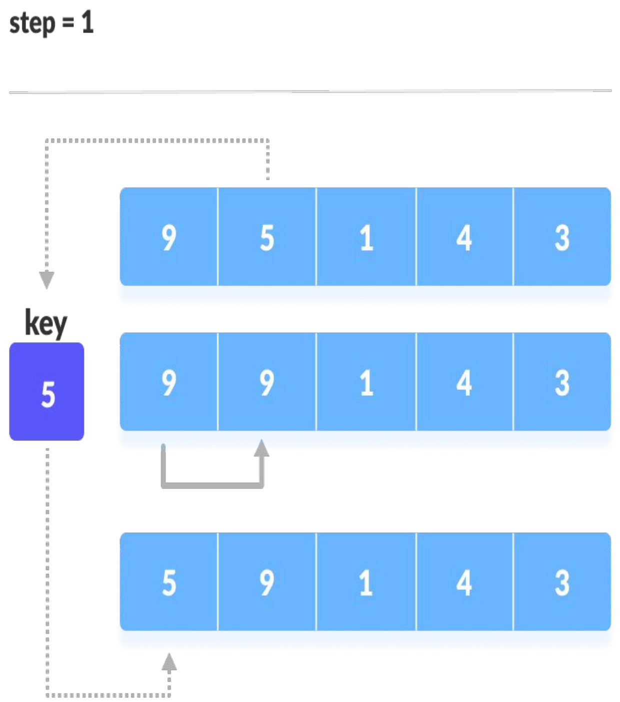
Suppose we need to sort the following array.



Initial array

The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

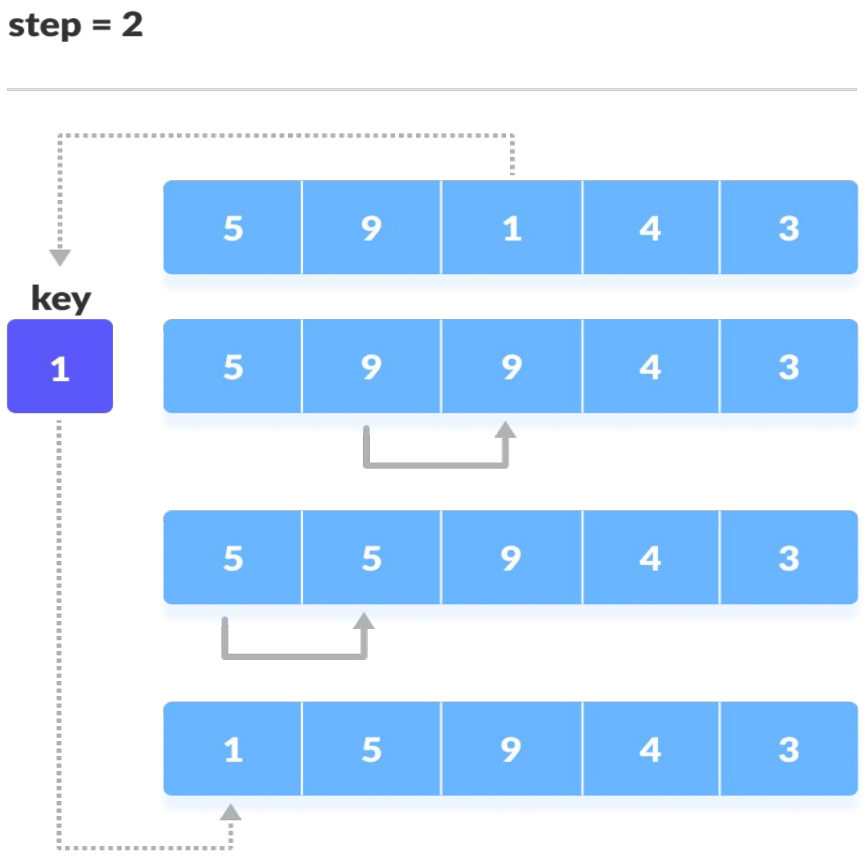
Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.



If the first element is greater than key, then key is placed in front of the first element.

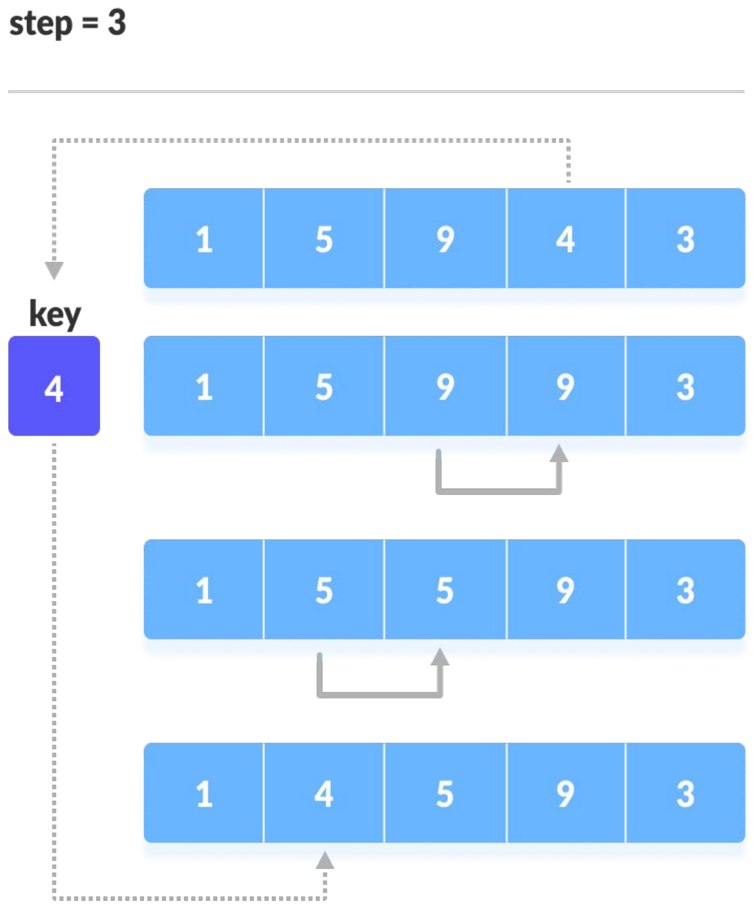
2. Now, the first two elements are sorted.

Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.

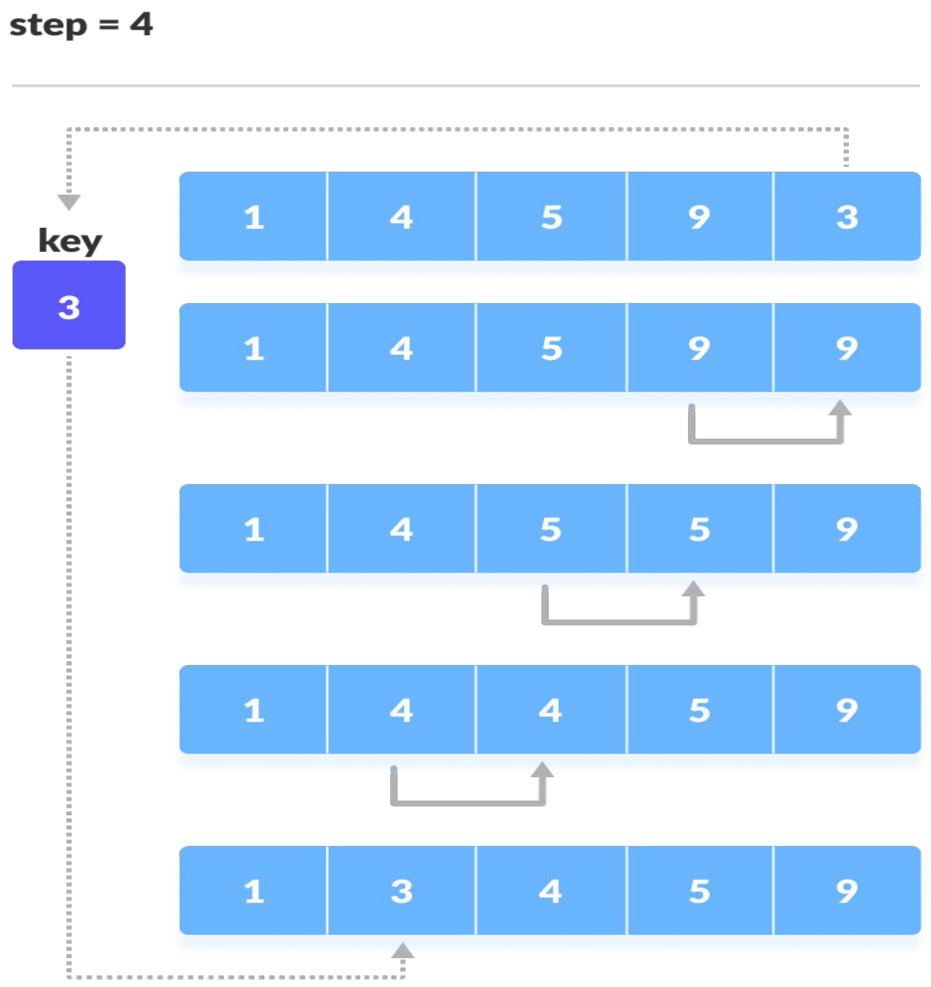


Place 1 at the beginning

3. Similarly, place every unsorted element at its correc position.



Place 4 behind 1



Place 3 behind 1 and the array is sorted

```
# Function to do insertion sort (sample 1)
def insertionSort(arr):

    # Traverse through 1 to len(arr)
    for i in range(1, len(arr)):

        key = arr[i]

        # Move elements of arr[0..i-1], that are
        # greater than key, to one position ahead
        # of their current position
        j = i-1
        while j >= 0 and key < arr[j] :
                arr[j + 1] = arr[j]
                j -= 1
        arr[j + 1] = key


# Driver code to test above
arr = [12, 11, 13, 5, 6]
insertionSort(arr)
for i in range(len(arr)):
    print ("% d" % arr[i])
```

```
# Insertion sort in Python (sample 2)

def insertionSort(array):

    for step in range(1, len(array)):
        key = array[step]
        j = step - 1

        # Compare key with each element on the left of it until an
element smaller than it is found
        # For descending order, change key<array[j] to key>array[j].
        while j >= 0 and key < array[j]:
            array[j + 1] = array[j]
            j = j - 1

        # Place key at after the element just smaller than it.
        array[j + 1] = key

data = [9, 5, 1, 4, 3]
insertionSort(data)
print('Sorted Array in Ascending Order:')
print(data)
```

**Time Complexities:**

**Worst Case Complexity:** O(n2)
Suppose, an array is in ascending order, and you want to sort it in descending order. In this case, worst case complexity occurs.

Each element has to be compared with each of the other elements so, for every nth element, (n-1) number of comparisons are made.

Thus, the total number of comparisons = n*(n-1) ~ n2
**Best Case Complexity:** O(n)
When the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. So, there are only n number of comparisons. Thus, complexity is linear.

**Average Case Complexity:** $O(n^2)$
It occurs when the elements of an array are in jumbled order (neither ascending nor descending).
**Space Complexity**
Space complexity is O(1) because an extra variable key is used.

## Insertion Sort Applications

The insertion sort is used when:

- the array is has a small number of elements
- there are only a few elements left to be sorted