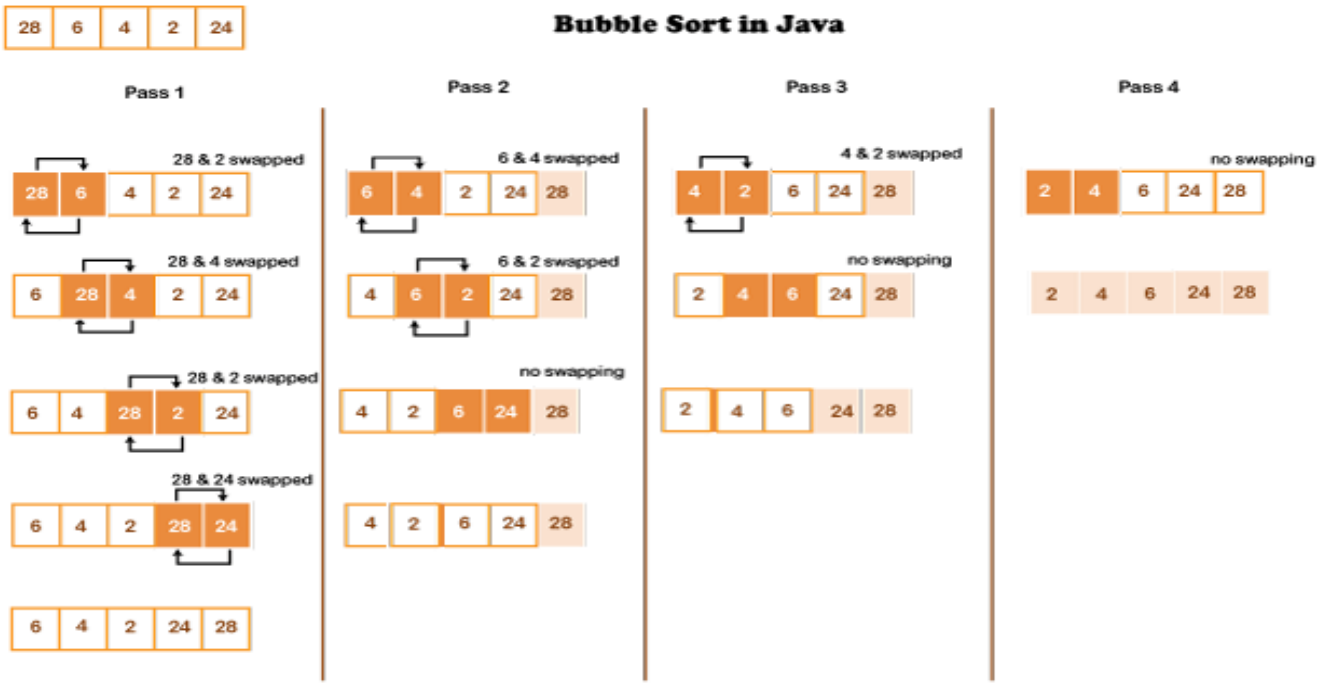
Bubble Sort

- **Bubble Sort** is one of the simplest sorting algorithms, primarily used for educational purposes due to its ease of implementation and understanding.
- Despite its simplicity, it's not the most efficient sorting algorithm for large datasets.
- However, comprehending Bubble Sort provides a foundational understanding of sorting algorithms, making it a valuable concept for beginners in programming and computer science.

How Bubble Sort Works?

Bubble Sort works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The name "Bubble Sort" comes from the way smaller elements bubble to the top of the list during each pass.



Bubble Sort Algorithm

1. Start with the first element of the array.
2. Compare the current element with the next element.
3. If the current element is greater than the next element, swap them.

4. Move to the next element and repeat steps 2 and 3 until the end of the array.
5. Repeat steps 1-4 until no more swaps are needed, indicating the array is sorted.

Let's implement the above algorithm in a Java program.

File Name: BubbleSortExample.java

```
1. public class BubbleSortExample {
2. //function that sorts the array
3.     static void bubbleSort(int[] arr) {
4.         int n = arr.length;
5.         int temp = 0;
6.          for(int i=0; i < n; i++){
7.                  for(int j=1; j < (n-i); j++){
8.                          if(arr[j-1] > arr[j]){
9.                                  //swap elements
10.                                 temp = arr[j-1];
11.                                 arr[j-1] = arr[j];
12.                                 arr[j] = temp;
13.                         }
14.}
15.       }
16.   }
17.   public static void main(String[] args) {
18.             int arr[] ={3,60,35,2,45,320,5};
19.             System.out.println("Array Before Bubble Sort");
20.             for(int i=0; i < arr.length; i++){
21.                     System.out.print(arr[i] + " ");
```

```
22.             }
23.             System.out.println();
24.             bubbleSort(arr);//sorting array elements using bubble sort
25.             System.out.println("Array After Bubble Sort");
26.             for(int i=0; i < arr.length; i++){
27.                     System.out.print(arr[i] + " ");
28.             }
29.     }
30.}
```

Output:

Array Before Bubble Sort

3 60 35 2 45 320 5

Array After Bubble Sort

2 3 5 35 45 60 320

Complexity Analysis

Bubble Sort has a time complexity of $O(n^2)$ in the worst and average cases, where n is the number of elements in the array. Because, in the worst case, for each element in the array, we may need to compare it with every other element. Despite its simplicity, Bubble Sort is not suitable for large datasets due to its poor time complexity.

Advantages of Bubble Sort

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space to sort an array.
- The elements with the same key value maintain their relative order in the sorted output because it is a stable algorithm.

Disadvantages of Bubble Sort

- Bubble sort has a time complexity of $O(N^2)$ that makes it very slow for large data sets.

- It can limit the efficiency of the algorithm in certain cases.
- It is a comparison-based sorting algorithm. It means that it requires a comparison operator to determine the relative order of elements in the input data set.

Conclusion

Bubble sort, although less efficient for large datasets, serves as a basic framework that helps understand more complex systems. Its simplicity makes it a valuable tool for beginners in programming and computer science to understand the conceptual and algorithmic complexity of programming. Using Bubble Sort in Java gives you a deeper understanding of the sorting algorithm and lays the foundation for finding more efficient alternatives.

INSERTION SORT

What is Insertion Sort?

Insertion sort is one of the comparison sort algorithms used to sort elements by iterating on one element at a time and placing the element in its correct position.

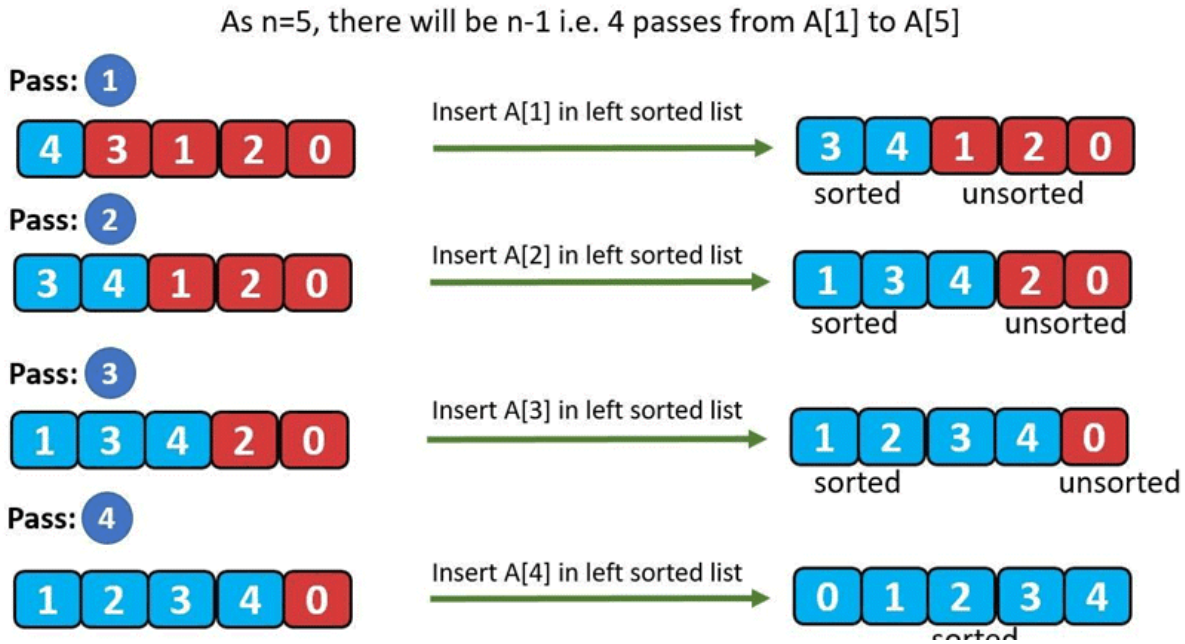
Each element is sequentially inserted in an already sorted list. The size of the already sorted list initially is one. The insertion sort algorithm ensures that the first k elements are sorted after the kth iteration.

Characteristics of Insertion Sort Algorithm

The Algorithm for Insertion Sort has following important Characteristics:

- It is a stable sorting technique, so it does not change the relative order of equal elements.
- It is efficient for smaller data sets but not effective for larger lists.
- Insertion Sort is adaptive, which reduces its total number of steps if it is partially sorted. Array is provided as input to make it efficient.

- The insert operation discussed above is the backbone of the insertion sort. The insert procedure is executed on every element, and in the end, we get the sorted list.



Complexity of Insertion Sort

Space Complexity

The insertion sort doesn't require extra space to sort the elements, the space complexity is constant, i.e., O(1).

Time Complexity

As insertion sort iterates each element simultaneously, it requires N-1 passes to sort N elements. For each pass, it may make zero swaps if the elements are already sorted, or swap if the elements are arranged in descending order.

- For pass 1, the minimum swaps required are zero, and the maximum swaps required are 1.
- For pass 2, the minimum swaps required are zero, and the maximum swaps required are 2.
- For pass N, the minimum swap required is zero, and the maximum swaps required are N.

- The minimum swap is zero, so the best time complexity is O(N) for iterating N passes.
- Total maximum swaps are (1+2+3+4+..+N) i. N(N+1)/2, the worst time complexity is O(N^2).

Here is the important time complexity of insertion sort:

- **Worst Case Complexity**: O(n2): Sorting an array in descending order when it is in ascending order is the worst-case scenario.
- **Best Case Complexity:** O(n): Best Case Complexity occurs when the array is already sorted, the outer loop runs for n number of times whereas the inner loop does not run at all. There is only n number of comparisons. So, in this case, complexity is linear.
- **Average Case Complexity:** O(n2): It happens when the elements of an array occur in the jumbled order, which is neither ascending nor descending.

Summary

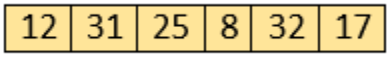
- Insertion sort is a sorting algorithm method that is based on the comparison.
- It is a stable sorting technique, so it does not change the relative order of equal elements.
- On every element, the insert operation is used to insert the element in the sorted sub-list.
- Insertion sort is an in-place sorting algorithm.
- The worst and average time complexity of insertion sort is quadratic, i.e., O(N^2).
- Insertion sort does not require any auxiliary space

Working of Insertion sort Algorithm

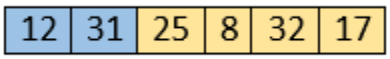
Now, let's see the working of the insertion sort Algorithm.

To understand the working of the insertion sort algorithm, let's take an unsorted array. It will be easier to understand the insertion sort via an example.

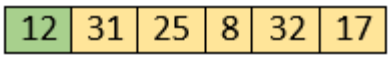
Let the elements of array are -



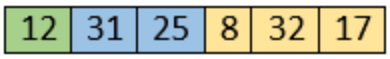
Initially, the first two elements are compared in insertion sort.

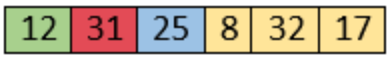


Here, 31 is greater than 12. That means both elements are already in ascending order. So, for now, 12 is stored in a sorted sub-array.



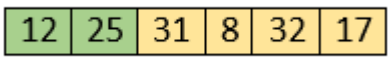
Now, move to the next two elements and compare them.



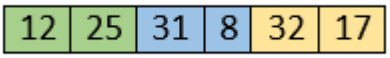


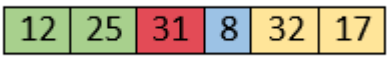
Here, 25 is smaller than 31. So, 31 is not at correct position. Now, swap 31 with 25. Along with swapping, insertion sort will also check it with all elements in the sorted array.

For now, the sorted array has only one element, i.e. 12. So, 25 is greater than 12. Hence, the sorted array remains sorted after swapping.

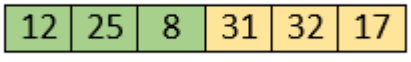


Now, two elements in the sorted array are 12 and 25. Move forward to the next elements that are 31 and 8.

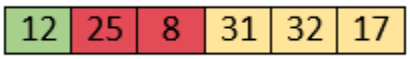




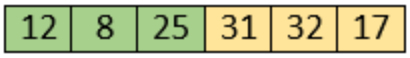
Both 31 and 8 are not sorted. So, swap them.



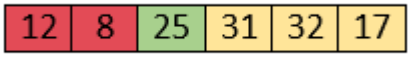
After swapping, elements 25 and 8 are unsorted.



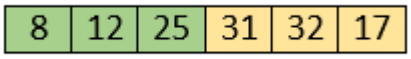
So, swap them.



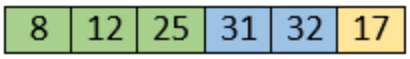
Now, elements 12 and 8 are unsorted.



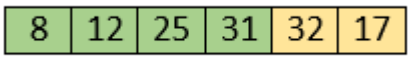
So, swap them too.



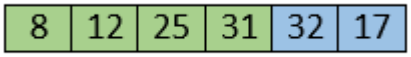
Now, the sorted array has three items that are 8, 12 and 25. Move to the next items that are 31 and 32.



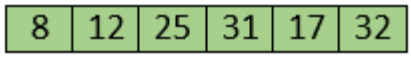
Hence, they are already sorted. Now, the sorted array includes 8, 12, 25 and 31.



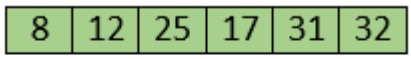
Move to the next elements that are 32 and 17.



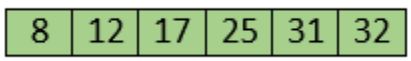
17 is smaller than 32. So, swap them.



Swapping makes 31 and 17 unsorted. So, swap them too.



Now, swapping makes 25 and 17 unsorted. So, perform swapping again.



Now, the array is completely sorted.

Insertion Sort Pseudo code

```
void sort(int arr[])
   {
      int n = arr.length;
      for (int i = 1; i < n; ++i) {
         int key = arr[i];
         int j = i - 1;

         /* Move elements of arr[0..i-1], that are
            greater than key, to one position ahead
            of their current position */
         while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
         }
         arr[j + 1] = key;
      }
   }
```

Selection Sort

Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the **smallest (or largest)** element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

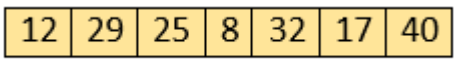
1. First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among remaining elements (or second smallest) and swap it with the second element.
3. We keep doing this until we get all elements moved to correct position.

Working of Selection sort Algorithm

Now, let's see the working of the Selection sort Algorithm.

To understand the working of the Selection sort algorithm, let's take an unsorted array. It will be easier to understand the Selection sort via an example.

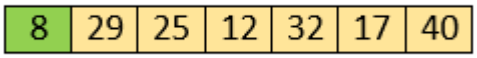
Let the elements of array are -



Now, for the first position in the sorted array, the entire array is to be scanned sequentially.

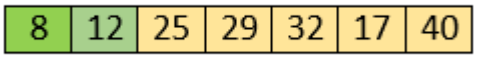
At present, **12** is stored at the first position, after searching the entire array, it is found that **8** is the smallest value.

So, swap 12 with 8. After the first iteration, 8 will appear at the first position in the sorted array.

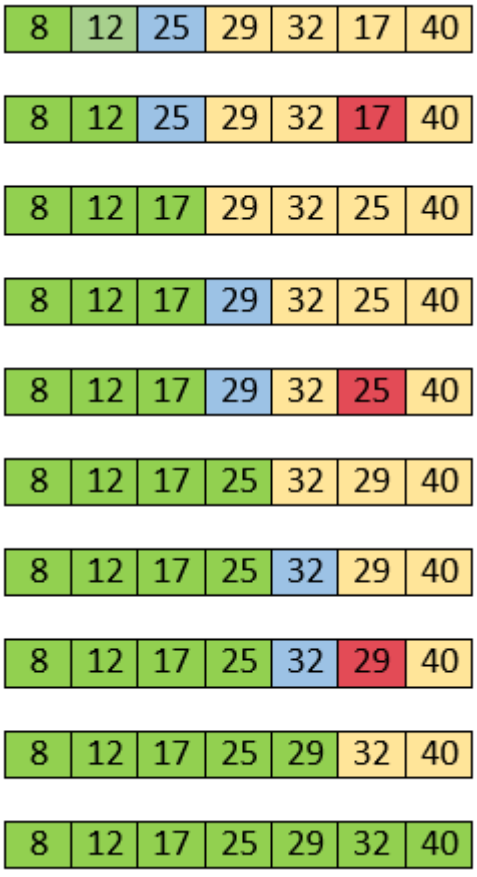


For the second position, where 29 is stored presently, we again sequentially scan the rest of the items of unsorted array. After scanning, we find that 12 is the second lowest element in the array that should be appeared at second position.

Now, swap 29 with 12. After the second iteration, 12 will appear at the second position in the sorted array. So, after two iterations, the two smallest values are placed at the beginning in a sorted way.



The same process is applied to the rest of the array elements. Now, we are showing a pictorial representation of the entire sorting process.



Now, the array is completely sorted.

Selection sort complexity

Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

1. Time Complexity

Case	Time Complexity
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$

```
import java.util.Arrays;

class Test{

   static void selectionSort(int[] arr){
      int n = arr.length;
      for (int i = 0; i < n - 1; i++) {
         // Assume the current position holds
         // the minimum element
         int min_idx = i;
         // Iterate through the unsorted portion
         // to find the actual minimum
         for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
               // Update min_idx if a smaller element
               // is found
               min_idx = j;
```

```
            }
        }

        // Move minimum element to its
        // correct position
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}

static void printArray(int[] arr){
    for (int val : arr) {
        System.out.print(val + " ");
    }
    System.out.println();
}

public static void main(String[] args){
    int[] arr = { 64, 25, 12, 22, 11 };

    System.out.print("Original array: ");
    printArray(arr);

    selectionSort(arr);
```

```
System.out.print("Sorted array: ");
```