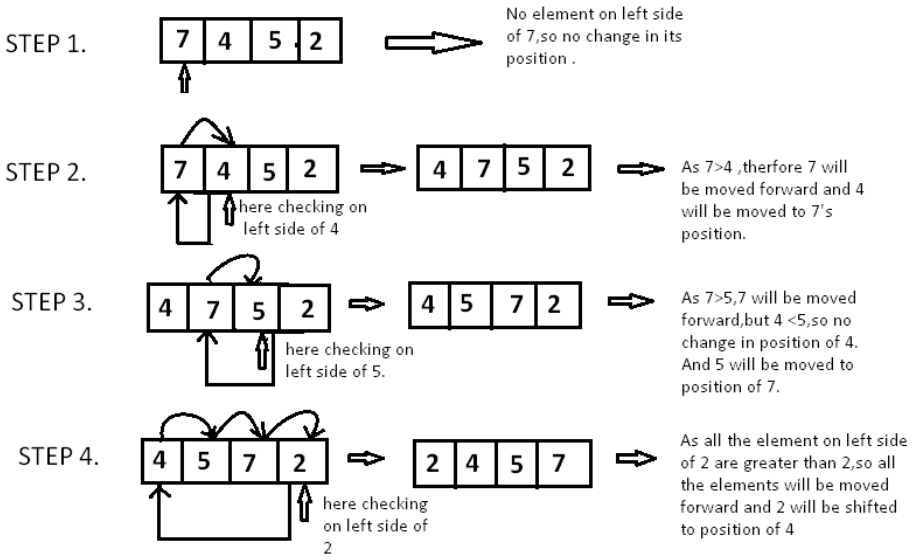
Title: Insertion Sort

Introduction: Insertion sort is an algorithm that constructs the final sorted list by gradually inserting elements in the correct order. It starts by assuming that the first item in the list is already sorted, then it compares each subsequent item with the ones before it. If an item is smaller than the ones before it, it is inserted into the correct position in the already sorted portion of the list. This process is repeated for all remaining items in the list.

Mechanism: The mechanism of insertion sort can be broken down into the following steps:

I. Start with the second element in the list and consider it as the key element.
II. Compare the key element with the element before it in the list. If the key element is smaller, swap it with the element before it.
III. Repeat step 2 for all elements before the key element.
IV. Continue this process for all elements in the list, moving from left to right.
V. When all elements have been compared and inserted into the correct position, the list will be sorted in ascending order.
VI. An important aspect to note is that the left side of the array after a certain index i is considered as sorted and the right side as unsorted. As we move through the array, the elements on the right are inserted into the correct position on the left, thus at the end of the array, we have a sorted array.

For example: Taking an array [7, 4, 5, 2].



Pseudo Code:

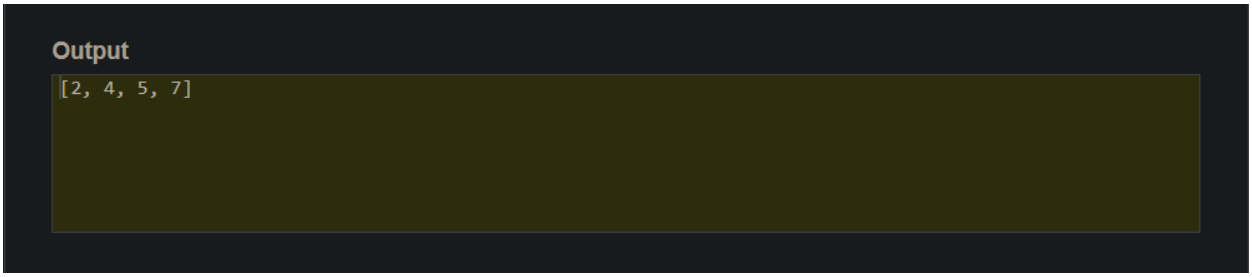
```
procedure insertionSort(A: list of sortable items)
   for i = 1 to length(A) - 1 do
      key = A[i]
      j = i - 1
      while j >= 0 and A[j] > key
         A[j+1] = A[j]
         j = j - 1
      end while
      A[j+1] = key
   end for
end procedure
```

Code:

1. Iterative:

```
import java.util.Arrays;
class Codechef {
   public static void insertionSort(int[] arr) {
      for (int i = 1; i < arr.length; i++) {
         int value = arr[i];
         int j = i;
         while (j > 0 && arr[j - 1] > value) {
            arr[j] = arr[j - 1];
            j--;
         }
         arr[j] = value;
      }
   }
```

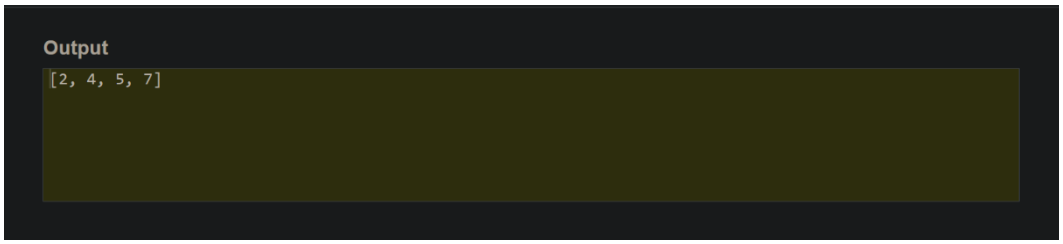
```
    public static void main(String[] args) {
        int[] arr = { 7, 4, 5, 2 };
        insertionSort(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```



2. Recursive:

```
import java.util.Arrays;
class Codechef {
    public static void insertionSortRec(int arr[]) {
        insertionSortRec(arr, arr.length);
    }
    public static void insertionSortRec(int arr[], int n) {
        if (n <= 1)
            return;
        insertionSortRec( arr, n - 1 );
        int last = arr[n - 1];
        int j = n - 2;
        while (j >= 0 && arr[j] > last) {
            arr[j + 1] = arr[j];
            j--;
        }
```

```
        arr[j + 1] = last;
    }
    public static void main(String[] args) {
        int[] arr = { 7, 4, 5, 2 };
        insertionSortRec(arr);
        System.out.println(Arrays.toString(arr));
    }
}
```



Time complexity Proof:

Insertion Sort(A)	**Cost**	**Times**
1. for j=2 to A. length	c1	n
2. key =A[j]	c2	n - 1
3. // Insert A[j] into the sorted sequence A[1...j-1].	0	n - 1
4. i = j - 1	c4	n-1
5. while i > 0 and A[j] > key	c5	$\sum_{j-2}^{n}$ tj
6. A[i+1] = A[i]	c6	$\sum_{j-2}^{n}(tj-1)$
7. i = i – 1	c7	$\sum_{j-2}^{n}(tj-1)$
8. A[i+1] = key	c8	n – 1

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) .$$

Best case (already sorted):

$t_j = 1$

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

In the form of an + b, i.e., **linear time: Ω(n).**

Worst case (already sorted):

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

In the form of an2 + bn + c, i.e., **quadratic time: O(n2).**

Average case:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) . \end{aligned}$$

In the form of an2 + bn + c, i.e., **quadratic time: Θ(n2).**

Space complexity analysis:

The space complexity of the insertion sort algorithm is O(1), as it sorts the elements in place and doesn't require any additional memory space. The algorithm uses a single key variable to temporarily store the current element being compared and a single variable to store the index of the element before it.

In terms of space complexity, Insertion Sort is considered as an in-place sorting algorithm, meaning it only uses a small, constant amount of extra memory space. This makes it efficient in terms of space usage, especially when working with large data sets.

Advantages:

- **Simple and easy to understand:** The insertion sort algorithm is relatively easy to understand, implement and debug, making it suitable for small datasets or for educational purposes.
- **In-place sorting:** As mentioned before, Insertion sort is an in-place sorting algorithm, which means it does not require additional memory space. This makes it efficient in terms of space usage, especially when working with large data sets.
- **Adaptive:** Insertion sort is adaptive, meaning it performs well in certain scenarios where the data is already partially sorted.
- **Efficient for small datasets:** Insertion sort has a time complexity of O(n) in the best case scenario when the input list is already sorted and a time complexity of O(n^2) in the worst case scenario. For small datasets, it can be faster than more complex algorithms with better average time complexity.
- **Stable:** Insertion sort preserves the relative order of elements with equal keys, meaning it is a stable sorting algorithm.
- **Online:** Insertion sort can sort a list as it receives it, it means it can sort the elements "online" as they appear.

Disadvantage:

- **Slow for large datasets:** Insertion sort has a time complexity of O(n^2) in the worst case scenario and O(n) in the best case scenario, which makes it less efficient for large datasets.
- **Not efficient for large data sets:** For large datasets the time complexity of O(n^2) makes the algorithm slow, and other algorithms like merge sort or quick sort are more suitable.
- **Not suitable for large data sets with complex data structures**: Insertion sort is not well-suited for large data sets with complex data structures, such as linked lists, because it requires repeatedly shifting elements in memory.

- **Not suitable for parallel processing:** Because of the sequential nature of the algorithm, it is not well suited for parallel processing.
- **Not efficient for datasets with many elements with the same value:** Because of the way the algorithm compares the elements, it takes more time to sort a list with many elements with the same value.
- **Not efficient for datasets with a lot of data with a few large values:** Because of the way the algorithm compares the elements, it takes more time to sort a list with many elements with large values.

Real life applications:

1. **Sorting playing cards:** Insertion sort can be used to sort a deck of playing cards by their rank or suit.
2. **Organizing personal finances:** Insertion sort can be used to sort financial transactions, such as bank statements or credit card statements, by date or amount.
3. **Sorting small lists:** Insertion sort can be used to sort small lists, such as lists of names, phone numbers, or addresses, in an address book or personal database.
4. **Sorting library books:** Insertion sort can be used to sort a library's collection of books by title, author, or publication date.
5. **Sorting collections of small items:** Insertion sort can be used to sort small collections, such as coins, stamps, or small toys, by value, date, or other criteria.

Optimizations and advancements:

1. **Bidirectional Insertion Sort:** A variation of the insertion sort algorithm that sorts the array from both the front and the back. This optimization can reduce the number of movements required to sort the array, leading to faster performance.
2. **Shell Sort:** A sorting algorithm that uses the principles of insertion sort, but with a more advanced approach to choosing the gap between elements to be compared. This optimization can result in a faster running time compared to traditional insertion sort.
3. **Adaptive Insertion Sort:** An optimization that uses the properties of partially sorted arrays to reduce the number of swaps required by the insertion sort algorithm.
4. **In-Place Insertion Sort:** An optimization that sorts the input array in place, without using any additional memory space to store intermediate results. This optimization can result in a more memory-efficient implementation of insertion sort.
5. **Parallel Insertion Sort:** An optimization that divides the input array into multiple parts and sorts each part in parallel, reducing the overall time required to sort the array.