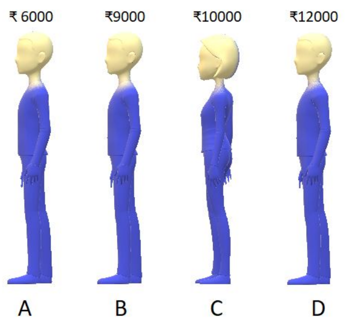
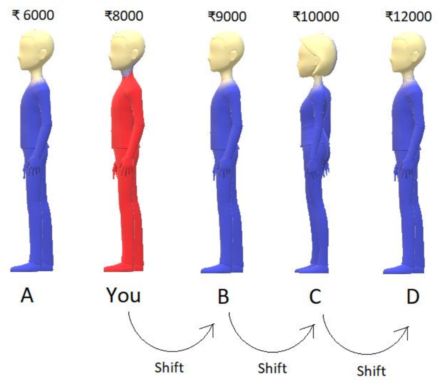
# Insertion Sort Algorithm

Suppose you were to stand in a queue where people are already sorted on the basis of the amount of money they have. Person with the least amount is standing in the front and the person with the largest sum in his pocket stands last. The below illustration describes the given situation.



Problem arises when you suppose you have ₹8000 in your pocket, and you want to be a part of this queue. You don't know where to stand. So, now you start from the last and keep asking the person standing there whether he has more money than you or less money than you. If you find someone with more money, you simply ask him/her to shift backward. And the moment you find a person having less money than you, you stand just behind him/her. So, after doing all this, you find a position in the 2nd place in the queue. The final situation is:



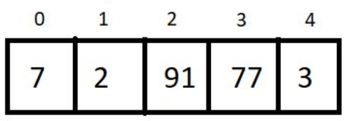
Now, suppose these were not the people but the numbers in an array. We would keep comparing two numbers, and if we find a number greater than the number we want to insert, we shift it backward. And the moment we find a number smaller, we insert the element at the vacant space just behind the smaller number.

And basically, what did we learn? We learned to insert an element in a sorted array. Although it felt very intuitive to just put yourself in the second position, what would you do if the queue had a thousand people? Not easy, right? And this is where we need a proper algorithm.

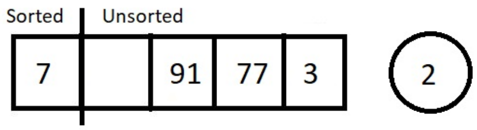
## Insertion Sort Algorithm:

Let’s just take an array, and use the insertion sort algorithm to sort its elements in increasing order.

Consider the given array below:

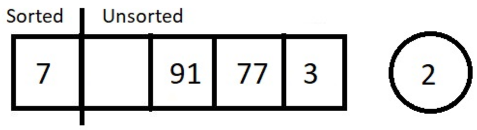


And what have we already learned? We have learned to put an arbitrary element inside a sorted array, using the insertion method we saw above. **And an array of a single element is always sorted.** So, what we have now is an array of length 5 with a subarray of length 1 already sorted.

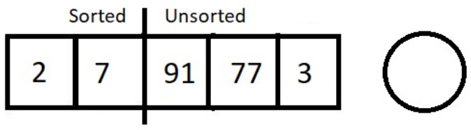


Moving from the left to the right, we will pluck the first element from the unsorted part, and insert it in the sorted subarray. This way at each insertion, our sorted subarray length would increase by 1 and unsorted subarray length decreases by 1. Let’s call each of these insertions and the traversal of the sorted subarray to find the best position, a pass.

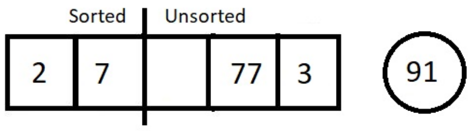
So, let’s start with pass 1, which is to insert 2 in the sorted array of length 1.



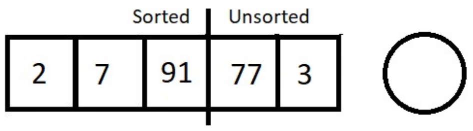
So, we plucked the first element from the unsorted part. Let’s insert element 2 at its correct position, which is before 7. And this increases the size of our sorted array.



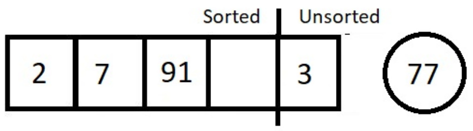
Let’s proceed to the next pass.



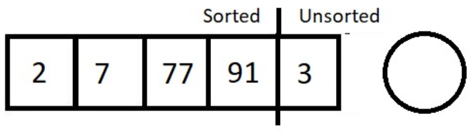
The next element we plucked out was 91. And its position in the sorted array is at the last. So that would cause zero shifting. And our array would look like this.



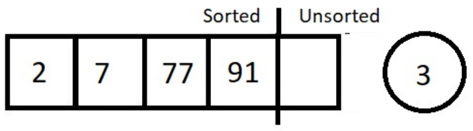
Our sorted subarray now has size 3, and unsorted subarray is now of length 2. Let’s proceed to the next pass which would be to traverse in this sorted array of length 3 and insert element 77.



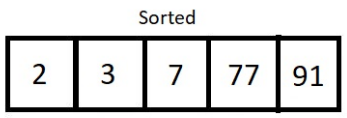
We started checking its best fit, and found the place next to element 7. So this time it would cause just a single shift of element 91.



As a result, we are left with a single element in the unsorted subarray. Let’s pull that out too in our last pass.



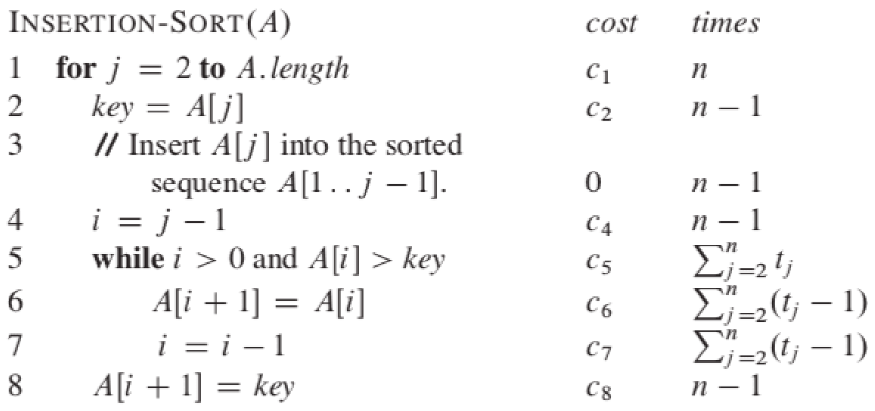
Since our new element to insert is the element 3, we started checking for its position from the back. The position is, no doubt, just next to element 2. So, we shifted elements 7, 77, and 91. Those were the only three shifts.  And the final sorted we received is illustrated below.



So, this was the main procedure behind the insertion sort algorithm.

## Analysis:

Now, let's analyze the performance of insertion sort:



| INSERTION-SORT($A$) | cost | times |
|---|---|---|
| 1 **for** $j = 2$ **to** $A.length$ | $c_1$ | $n$ |
| 2 $\quad key = A[j]$ | $c_2$ | $n-1$ |
| 3 $\quad$ // Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$. | 0 | $n-1$ |
| 4 $\quad i = j-1$ | $c_4$ | $n-1$ |
| 5 $\quad$ **while** $i > 0$ and $A[i] > key$ | $c_5$ | $\sum_{j=2}^{n} t_j$ |
| 6 $\qquad A[i+1] = A[i]$ | $c_6$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 7 $\qquad i = i - 1$ | $c_7$ | $\sum_{j=2}^{n} (t_j - 1)$ |
| 8 $\quad A[i+1] = key$ | $c_8$ | $n-1$ |

Steps 1, 2, 4 and 8 will run **n-1** times (from second to the last element).

Step 5 will run tj times (assumption) for n-1 elements (second till last). Similarly, steps 6 and 7 will run tj-1 times for n-1 elements.

Summing up, the total cost for insertion sort is:

Now we analyze the worst and best case for Insertion Sort.

### Worst case:

The array is reverse sorted. The inner while loop executes i times during the i-th iteration of the outer loop and hence, while condition will be checked i-1 times for comparing key with all elements in the sorted array.

The explanation for the first summation is simple - the sum of numbers from 1 to n is n(n+1)/2, since the summation starts from 2 and not 1, we subtract 1 from the result. We can simplify the second summation similarly by replacing n by n-1 in the first summation.

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

We can express this worst-case running time as **an^2+bn+c** where a, b and c are constants.

## Best case:

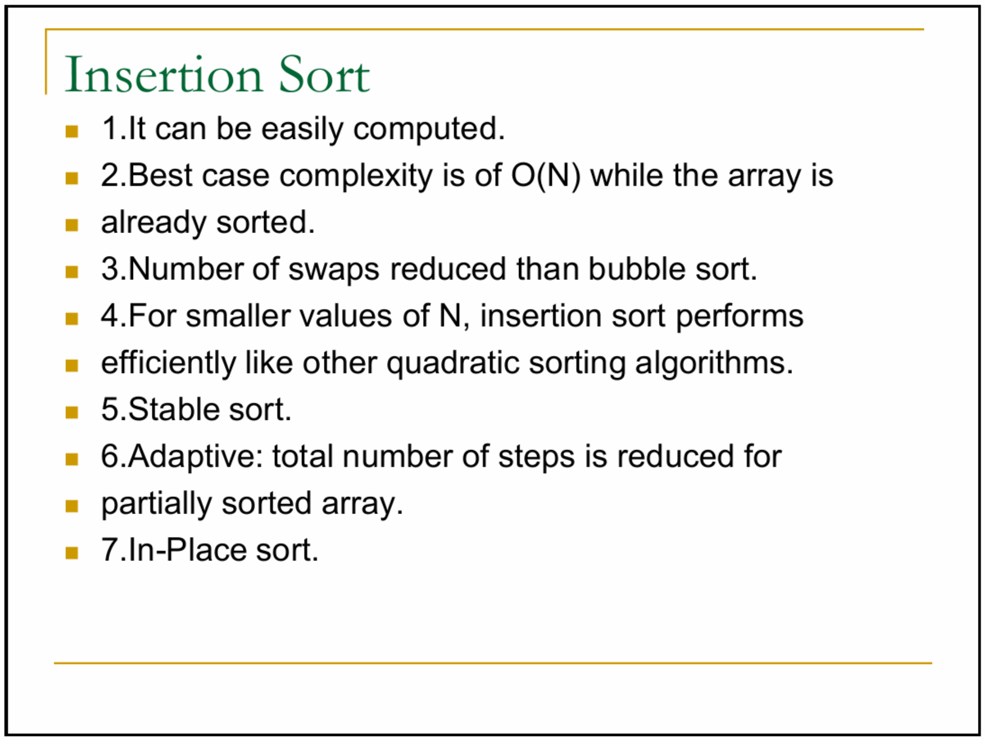
The array is already sorted. tj will be 1 for each element as while condition will be checked once and fail because A[i] is not greater than **key**.

Hence cost for steps 1, 2, 4 and 8 will remain the same. Cost for step 5 will be n-1 and cost for step 6 and 7 will be 0. So cost for best case is:

We can express this running time as **an+b** where a and b are constants.

## Summary of Time Complexities

- **Best Case:** O(n) – When the array is already sorted.
- **Worst Case:** O(n^2) – When the array is sorted in reverse order.
- **Average Case:** O(n^2) – When elements are in a random order.



## Insertion Sort

- 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.