Insertion sort

Algorithm description

Roughly speaking, **insertion sort** is a simple sorting algorithm that sorts one element at a time. It automatically divides the array into the **sorted** and the **unsorted** part, as we will explain below. At each iteration, the algorithm moves an element from the unsorted part to the appropriate position in the sorted one, until it sorts all array elements.
Formally, the algorithm works as follows:

1. Assume the leftmost element belongs to the sorted part of the array, and all remaining elements are in the unsorted part;
2. Choose the first element from the unsorted part and insert that element into the sorted list at its proper position;
3. Repeat step 2, until all elements are sorted.

How to perform **insertion** at step 2? Compare the selected element with its left neighbor: if it is greater than or equal to it, the element is already in its proper position. Otherwise, move the neighbor one position to the right to make space for the current element. Now, it has a new left neighbor: repeat the same procedure until our element reaches the desired position, i.e. is greater than or equal to the element on its left, and finally, insert it there.

We can notice that after every step, the sorted part literally gets sorted. The fact that at every step we move one unsorted element to the sorted part, guarantees that we will end up with a sorted array after

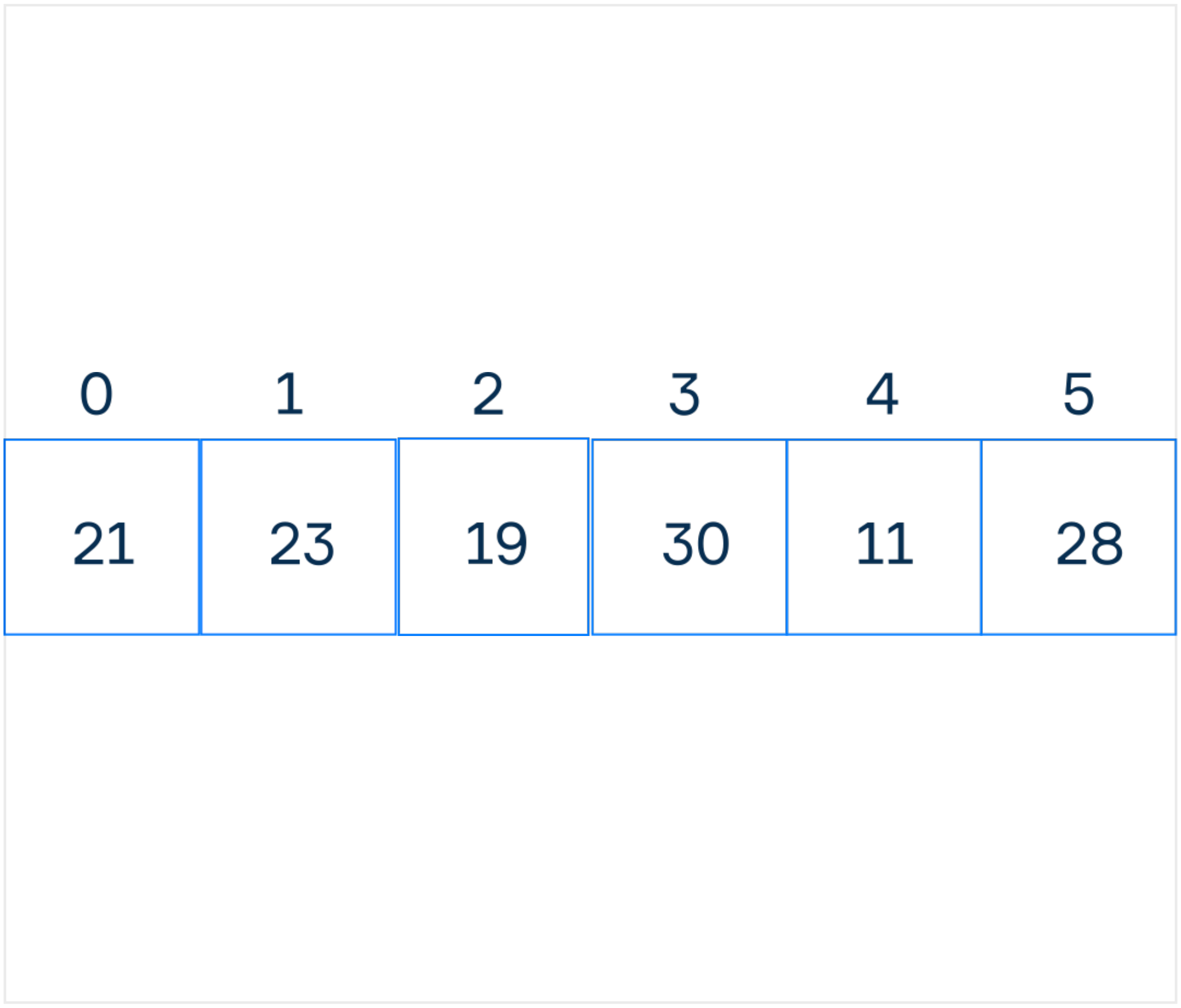
$n-1$

 steps, where

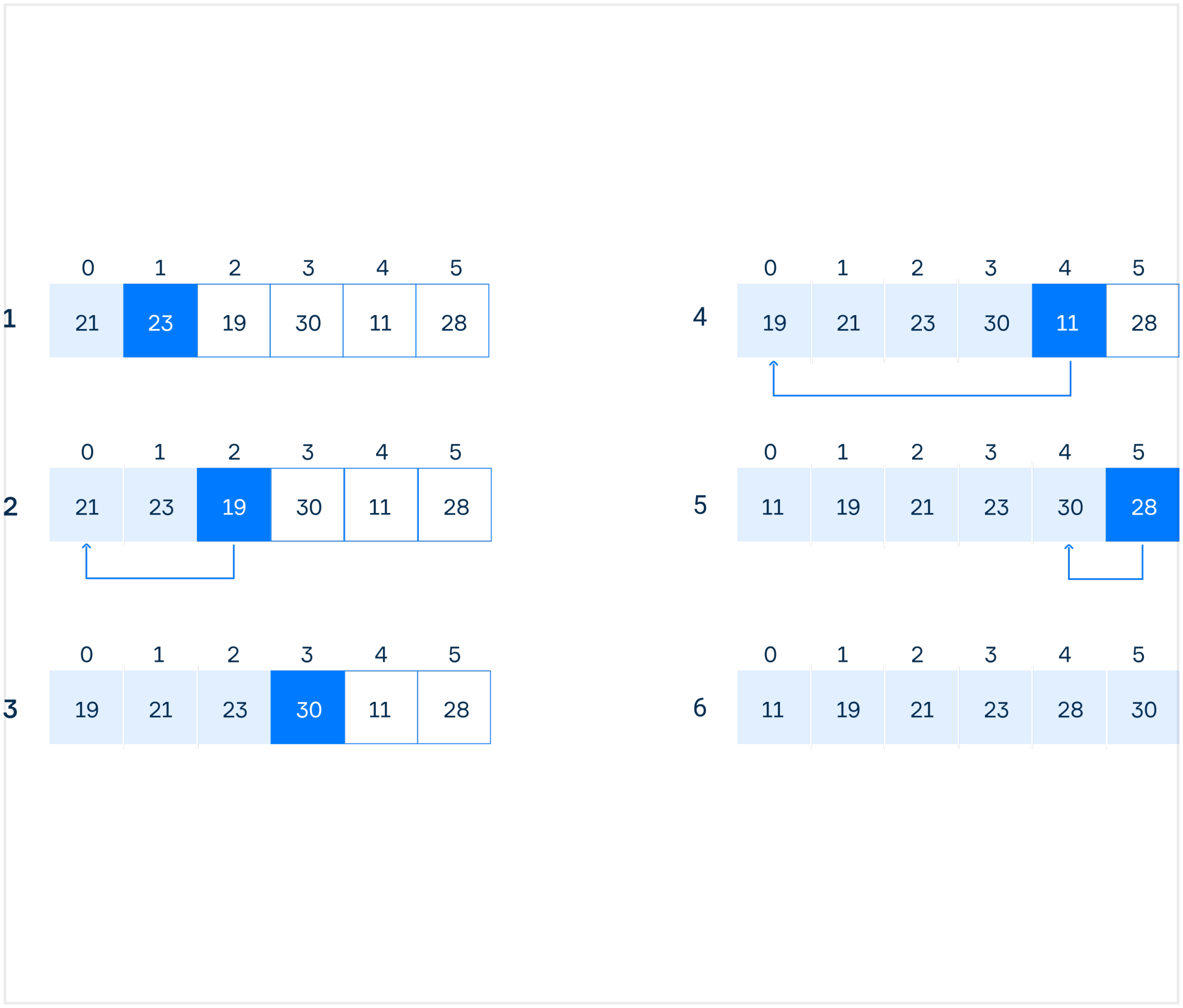
n

 denotes the number of elements. In other words, it proves that the algorithm works correctly.

Example



The array has six elements, the first element's index is 0, and the last element has the index 5. The following image illustrates how the insertion sort algorithm works:



Let's explain in detail the first few steps:

1. The sorted part includes only a single element with the index 0, since it's the leftmost element in the array. Let's look at the second one (23). It's greater than the last element in the sorted part, which means we don't move it.
2. Now, the sorted part includes elements with the indexes 0-1. We consider the element with the index 2 (19

). It's smaller than the last element in the sorted part, hence, we move the last element, namely 23
   , to the right. Still, the new neighbor, 21
   , is greater than our element, therefore, we push 21
   to the right and insert 19
   in this position. Now, after moving, it has index 0.

Similarly, we keep applying the steps of the algorithm to the rest of the elements, as shown in the illustration above. At step 6, the unsorted part is empty, which means the whole array is sorted. Now you can be proud of yourself, since you've completed the professor's task!

Pseudocode

```
i = 1                          // we skip index 0, as it is within the sorted part
while i < length(A):           // we will check all the elements in the unsorted part
    x = A[i]                   // save the value of the selected element
    j = i - 1                  // take the index of the left neighbor
    while j >= 0 and A[j] > x:   // comparing our selected element with its left neighbors
        A[j+1] = A[j]          // if our element is lower, push the neighbor to the right
        j = j - 1              // move left to the next neighbor
    A[j+1] = x                 // the proper position has been reached, insert our element there
    i = i + 1                  // examine the next element in the unsorted part

// N.B. We could simply swap x and A[j] in line 6, but this would take more time,
// since we would need to perform extra assignments during each iteration.
// Instead, we perform only one assignment in line 6.
```

Pseudocode

```
i = 1                          // we skip index 0, as it is within the sorted part
while i < length(A):           // we will check all the elements in the unsorted part
    x = A[i]                   // save the value of the selected element
    j = i - 1                  // take the index of the left neighbor
    while j >= 0 and A[j] > x:   // comparing our selected element with its left neighbors
        A[j+1] = A[j]          // if our element is lower, push the neighbor to the right
        j = j - 1              // move left to the next neighbor
    A[j+1] = x                 // the proper position has been reached, insert our element there
    i = i + 1                  // examine the next element in the unsorted part

// N.B. We could simply swap x and A[j] in line 6, but this would take more time,
// since we would need to perform extra assignments during each iteration.
// Instead, we perform only one assignment in line 6.
```

Complexity and other features

Every algorithm has its advantages and shortcomings, and insertion sort is not an exception. Our algorithm works pretty fast with already sorted arrays. It takes $O(n)$ time, because the elements are already in their proper positions, so we don't need to move anything. In pseudocode language, this means that the inner while cycle will not be executed. Another great advantage of the insertion sort algorithm is its simple implementation and the human-like logic behind it. Also, it is worth mentioning that our algorithm is stable and in-place, meaning that it doesn't require any extra memory.

With the insertion sort's features we've mentioned, we can conclude that it is advisable to use this algorithm for arrays that contain few elements or are almost fully sorted. Such examples are sets of data produced by another program, sorted datasets, in which we need to append a few new elements, error log files, etc.
However, quite the opposite happens with the algorithm's efficiency in the average and worst cases. When this happens, insertion sort's time complexity is quadratic, since both while loops will be executed. For example, if the array is in reverse order, we need to compare every pair of elements and make
$N/(N-1)/2$
 comparisons, which is
$O(n^2)$
Due to its time complexity, it is not recommended to use insertion sort for large datasets. This is why sorting methods in programming languages are based on algorithms other than insertion sort, as we will see in the following topics.