**Quick Sort**

Quicksort works according to the “divide and conquer” principle:

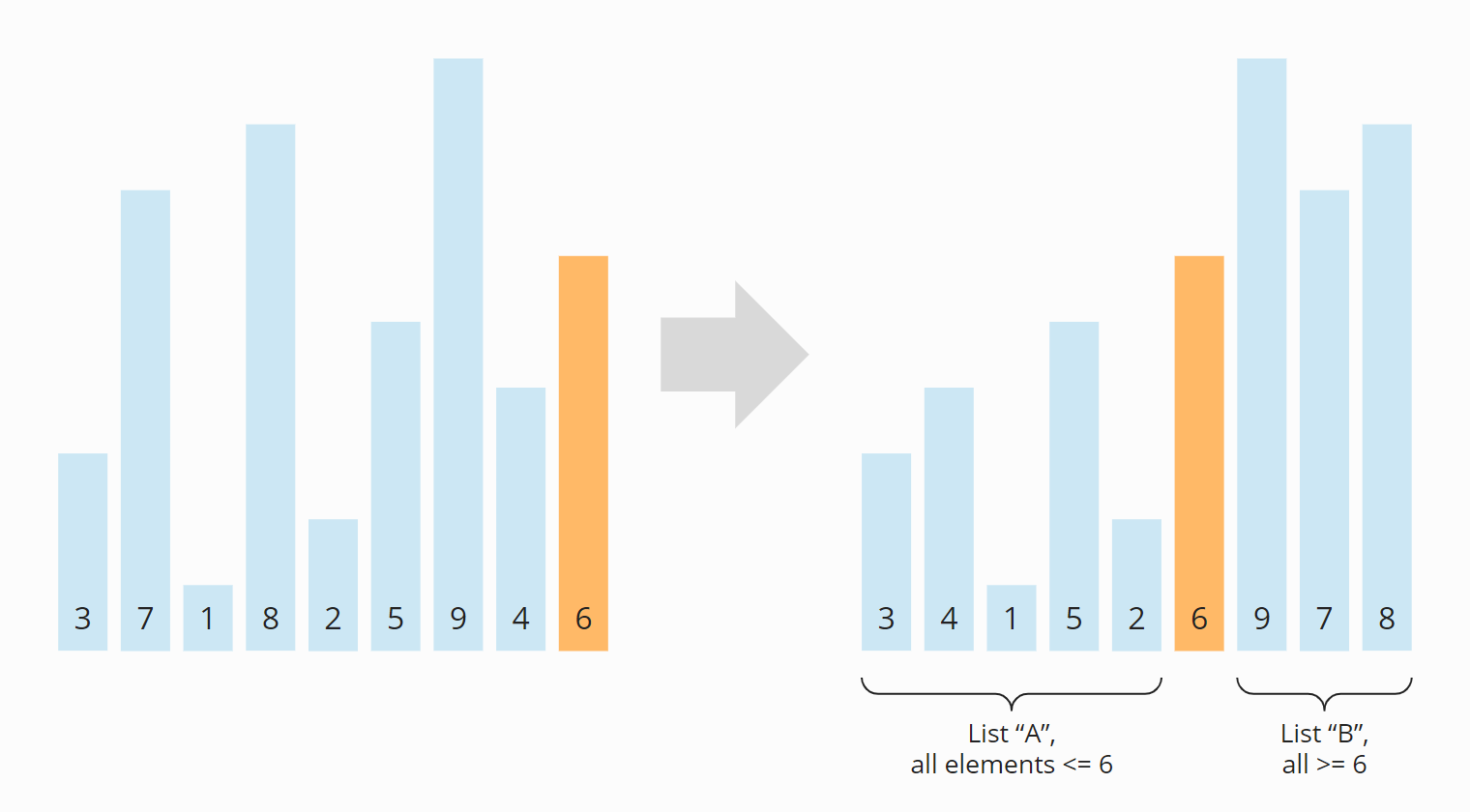
First, we divide the elements to be sorted into two sections – one with small elements (“A” in the following example) and one with large elements (“B” in the example).

The so-called pivot element determines which elements are small and which are large. The pivot element can be any element from the input array. (The pivot strategy determines which one is chosen, more on this later.)

The array is now rearranged so that:

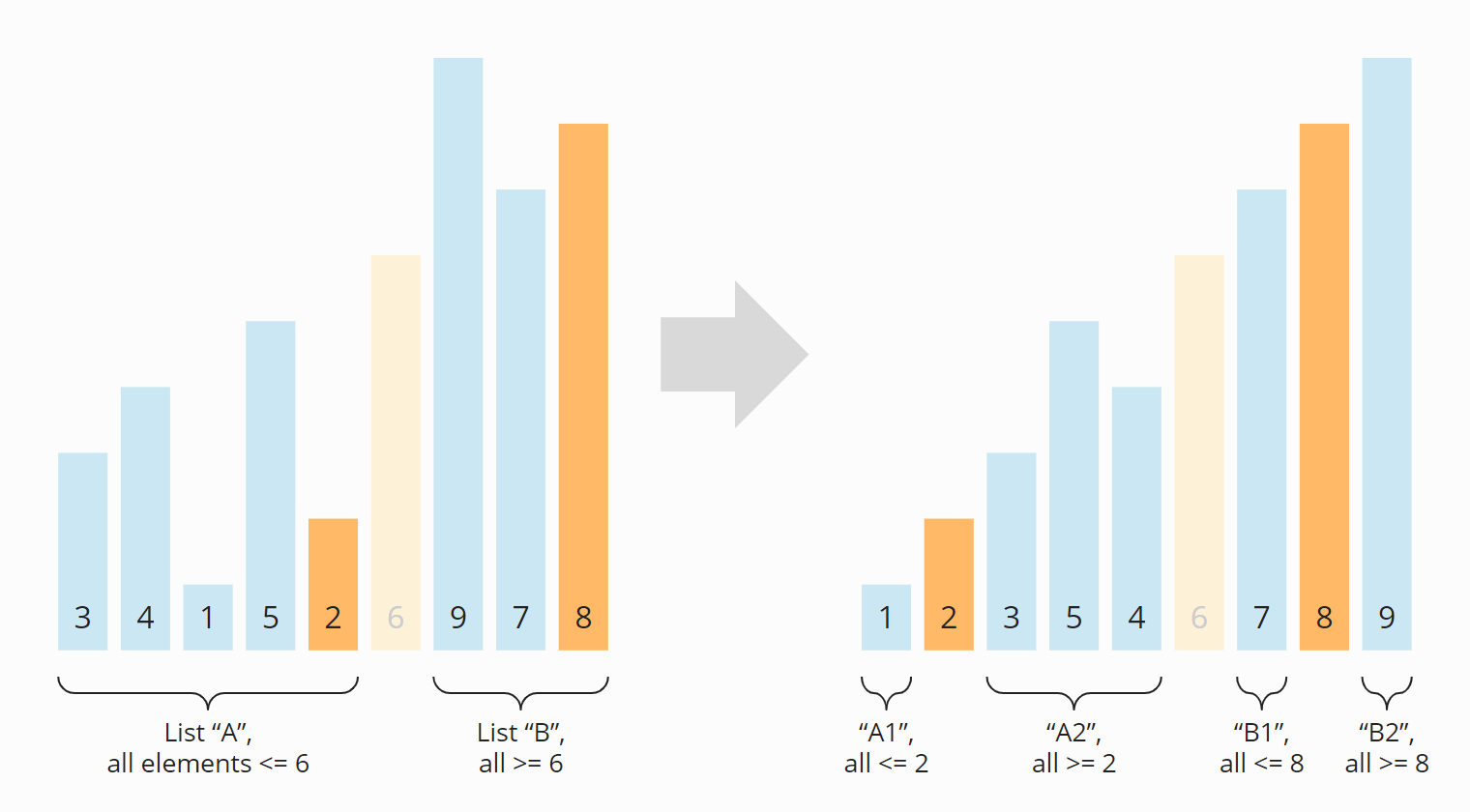
* the elements that are smaller than the pivot element end up in the left section,
* the elements that are larger than the pivot element end up in the right section,
* the pivot element is positioned between the two sections – which also is its final position.

In the following example, the elements [3, 7, 1, 8, 2, 5, 9, 4, 6] are sorted this way. As the pivot element, I chose the last element of the unsorted input array (the orange-colored 6):



This division into two subarrays is called partitioning.

The subarrays to the left and right of the pivot element are still unsorted after partitioning. These subarrays will now also bo partitioned. I drew the pivot element from the previous step, the 6,



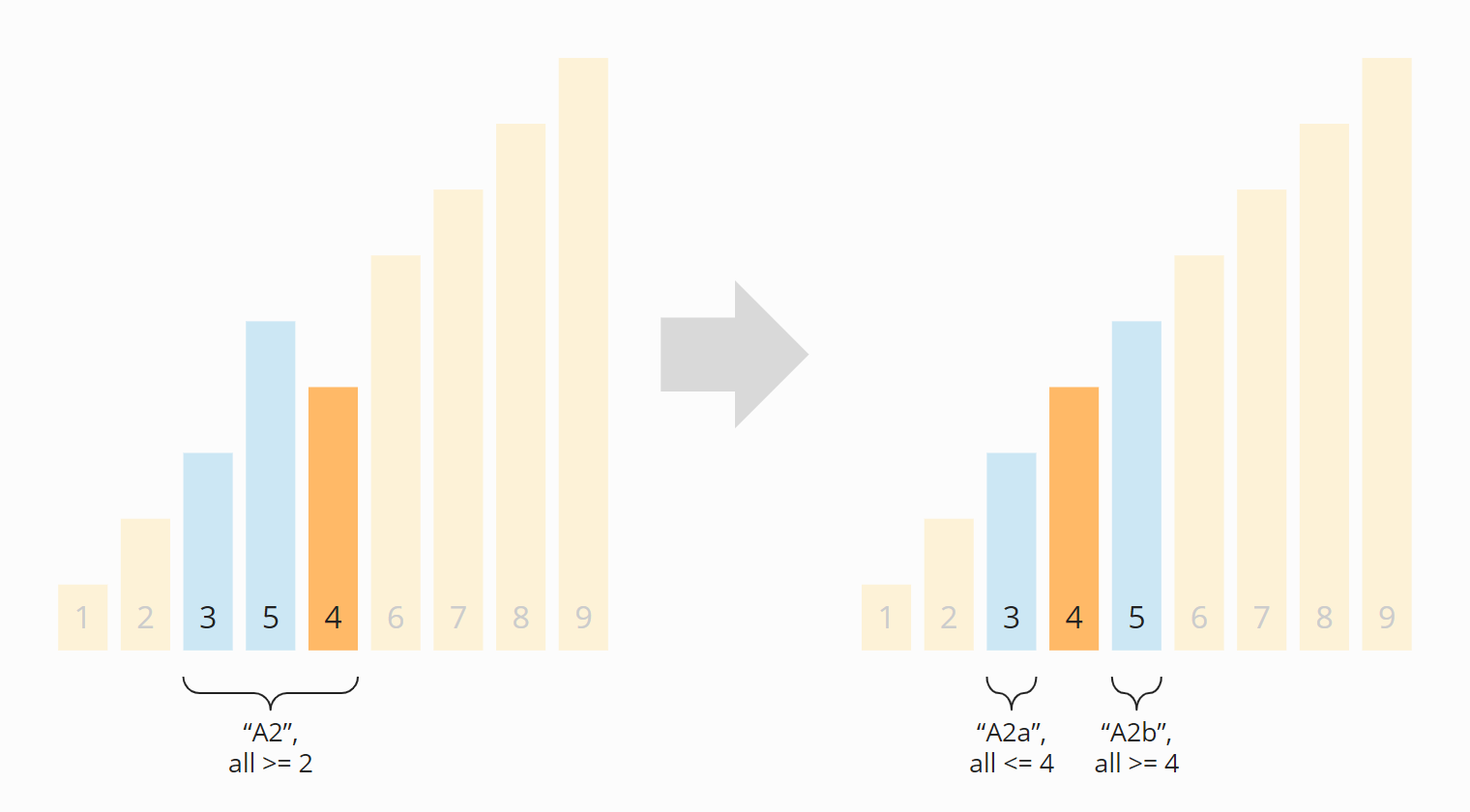
We now have four sections: Section

A turned into A1 and A2;

B turned into B1 and B2.

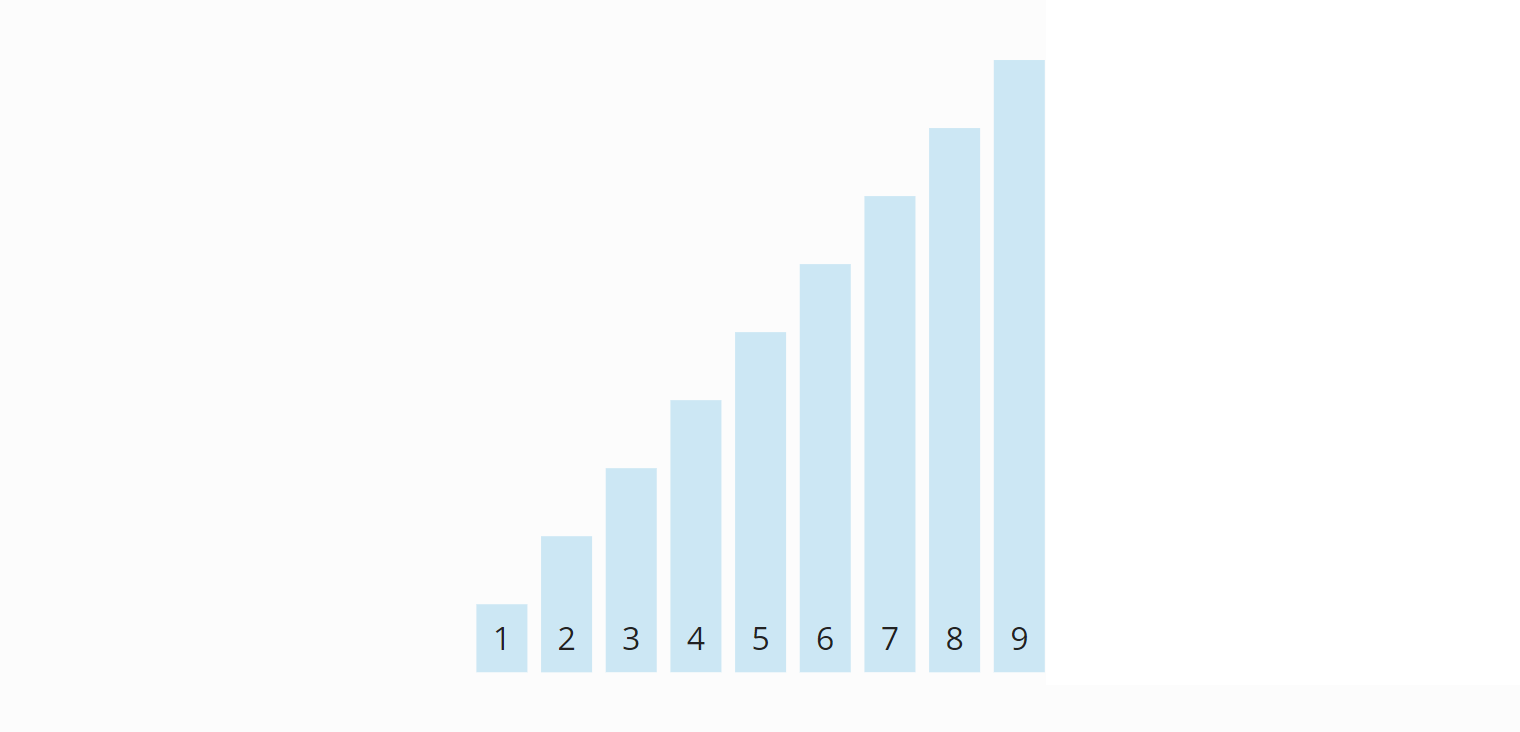
The sections A1, B1, and B2 consist of only one element and are therefore considered sorted (“conquered” in the sense of “divide and conquer”).

Now the subarray A2 is the only left to be partitioned:



The two partitions A2a and A2b that emerged from A2 in this step are again of length one. They are therefore considered sorted.

Thus, all subarrays are sorted – and so is the entire array:



The algorithm is, therefore, terminated.

partition(elements, left, right) {

pivot = elements[right];

1. i = left;

2. j = right - 1;

3. Repeat step 4 to 6 while (i < j) {

// Find the first element >= pivot

4. while (elements[i] < pivot)

. i= i+1

// Find the last element < pivot

5. while (j > left && elements[j] >= pivot)

j=j-1

6. // If the greater element is left of the lesser element, switch them

if (i < j) {

swap(elements[i], elements[j])

I = I + 1

J = J -1

// i == j means we haven't checked this index yet.

// Move i right if necessary so that i marks the start of the right array.

7. if (i == j && elements[i] < pivot)

I = I + 1

// Move pivot element to its final position

8. if (elements[i] != pivot) {

Swap (elements[i], elements[right])

9. return i;

quicksort(elements, left, right) {

// End of recursion reached?

1. if (left >= right) return;

2. pivotPos = partition(elements, left, right);

3. quicksort(elements, left, pivotPos - 1);

4. quicksort(elements, pivotPos + 1, right);

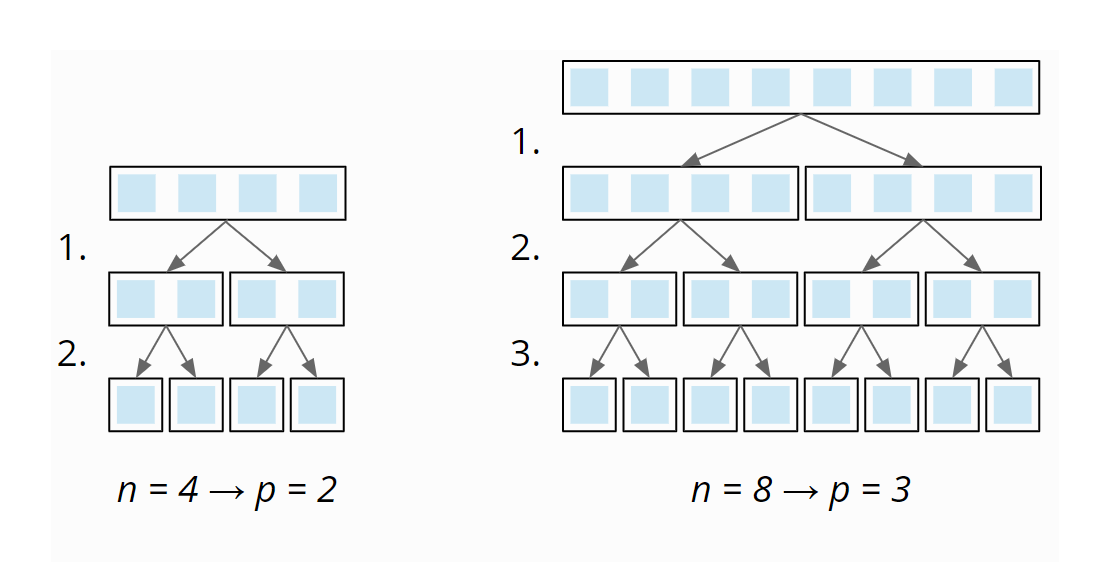
}

## Quicksort Time Complexity

### Best-Case Time Complexity

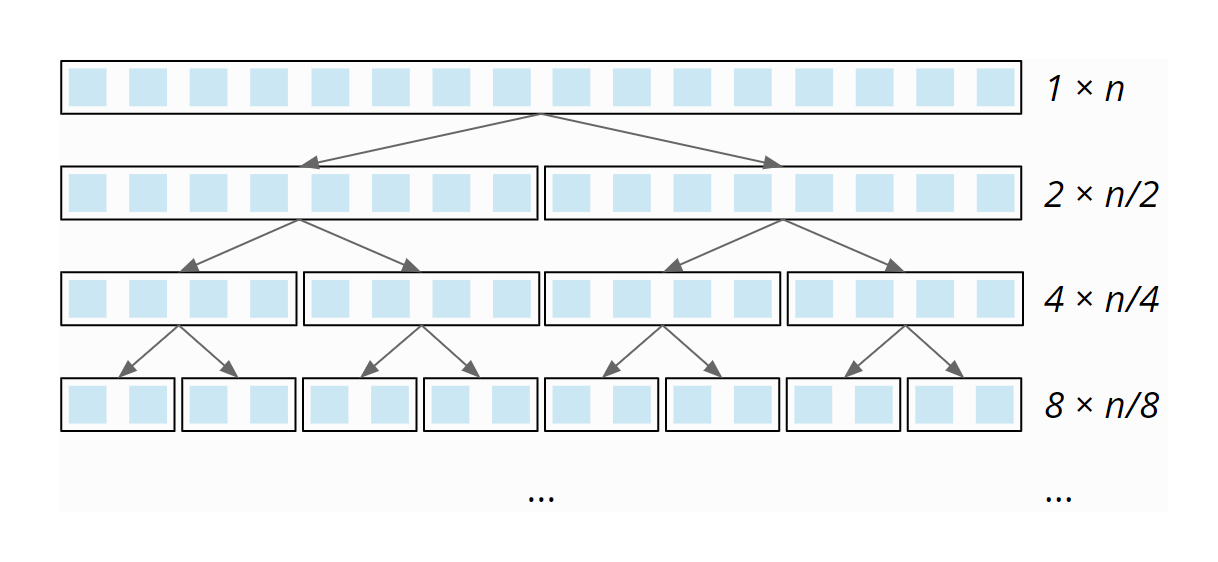
* Quicksort achieves optimal performance if we always divide the arrays and subarrays into two partitions of equal size.
* Because then, if the number of elements n is doubled, we only need one additional partitioning level p.

The following diagram shows that two partitioning levels are needed with four elements – and only one more with eight elements:



**So the number of partitioning levels is**log2 n**.**

At each partitioning level, we have to divide a total of n elements into left and right partitions (1 × n at the first level, 2 × n/2 at the second, 4 × n/4 at the third, etc.):



* This partitioning is done – due to the single loop within the partitioning – with linear complexity:
* When the array size doubles, the partitioning effort doubles as well. The total effort is, therefore, the same at all partitioning levels.

**So with**n**elements times**log2 n**partitioning levels.**

**The best-case time complexity of Quicksort is:**O(n log n)

### Average-Case Time Complexity

Omitting the complex calculation ( out of scope of the syllabus)

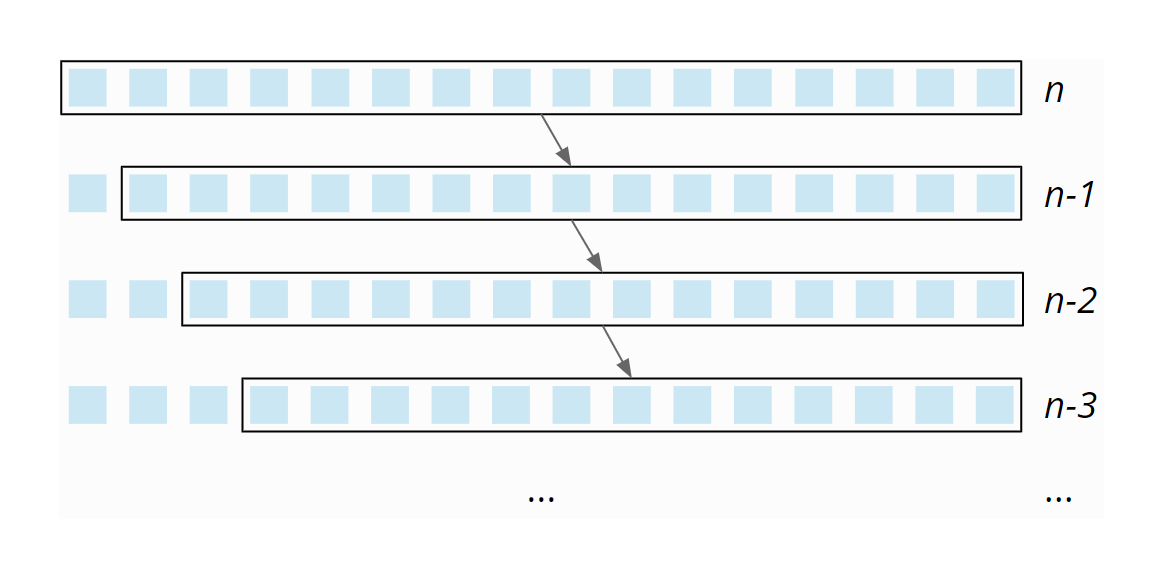
average number of comparison operations is 1.39 n × log2 n – (still quasilinear time)

**The best-case time complexity of Quicksort is also:**O(n log n)

### Worst-case Time Complexity

If the pivot element is always the smallest or largest element of the (sub)array (e.g. because our input data is already sorted and we always choose the last one as the pivot element), the array would not be divided into two approximately equally sized partitions, but one of length 0 (since no element is larger than the pivot element) and one of length n-1 (all elements except the pivot element).

Therefore we would need n partitioning levels with a partitioning effort of size n, n-1, n-2, etc.:



Size of partitioning effort decreases linearly from n to 0 – on average, it is, therefore, ½ n. Thus, with n partitioning levels, the total effort is n × ½ n = ½ n². Therefore:

The worst-case time complexity of Quicksort is: O(n²)

Note: In practice, the attempt to sort an array presorted in ascending or descending order using the pivot strategy “right element” would quickly fail due to a StackOverflowException, since the recursion would have to go as deep as the array is large.

### Space Complexity of Quicksort

For each recursion level, we need additional memory on the stack. In average and best case, the maximum recursion depth is limited by O(log n)

In the worst case, the maximum recursion depth is n.

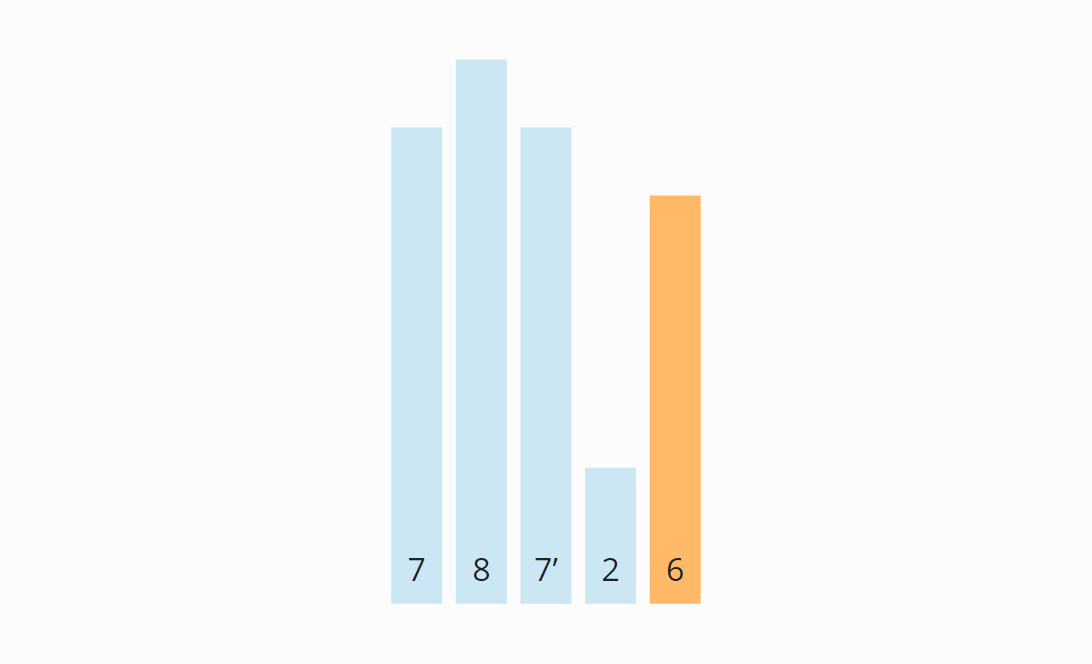
The additional memory requirement per recursion level is constant. Therefore:

 O(log n)

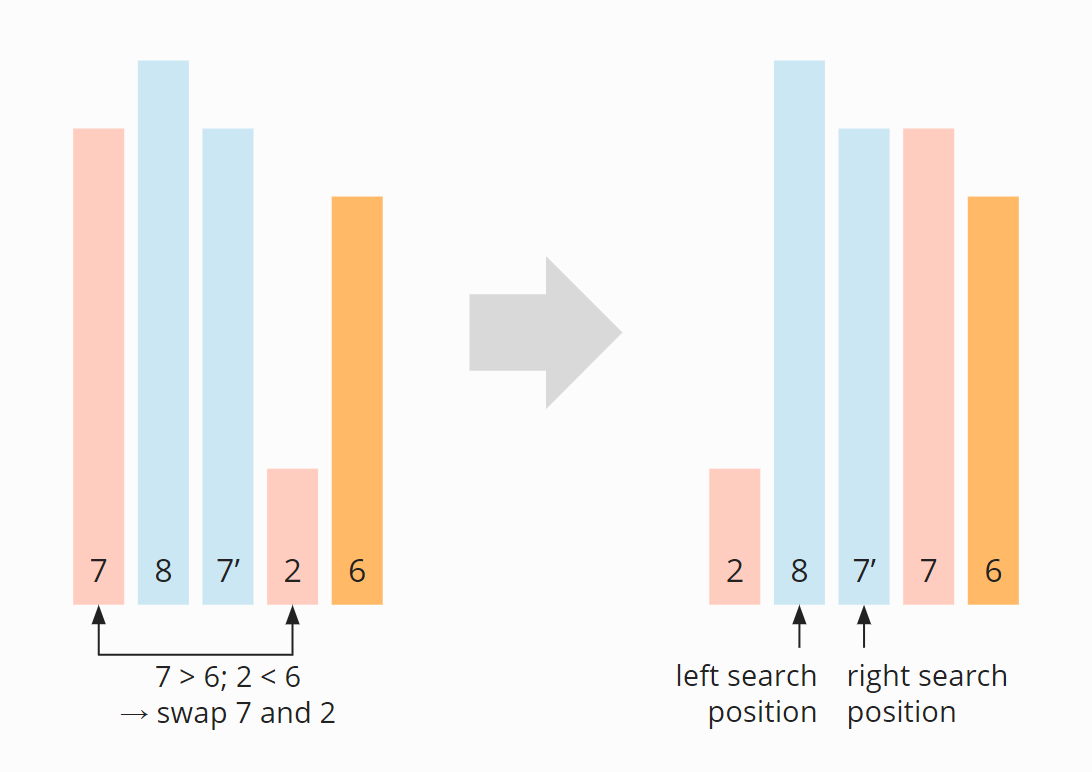
### Stability of Quicksort

Because of the way elements within the partitioning are divided into subsections, elements with the same key can change their original order.

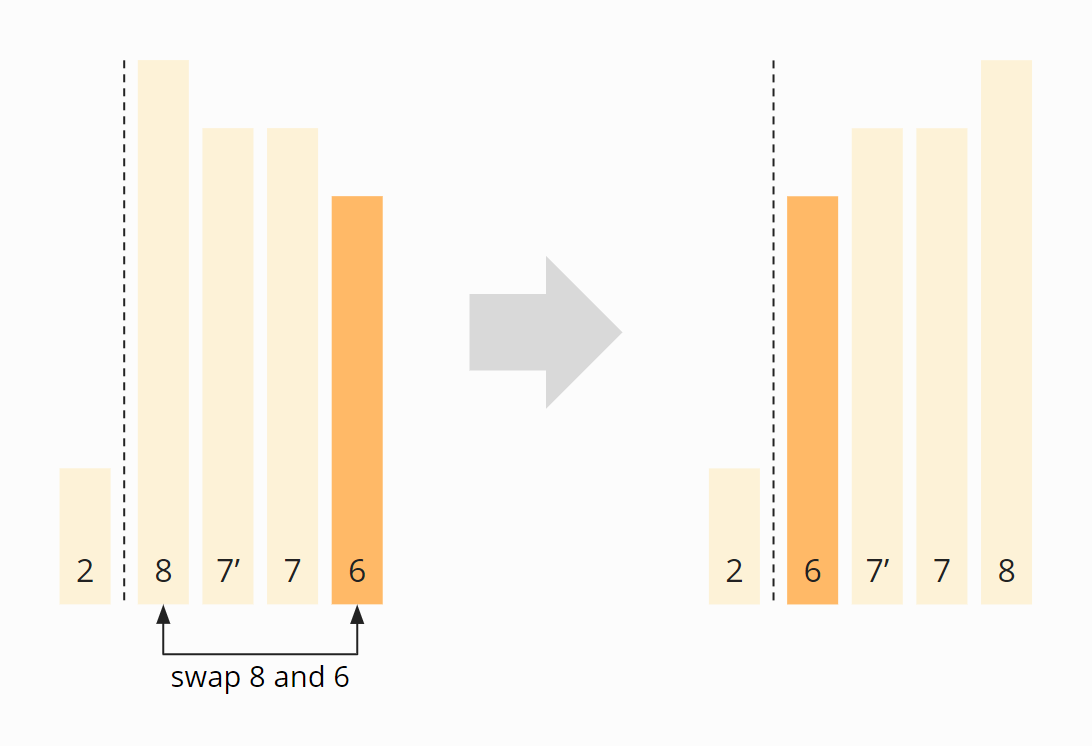
Here is a simple example: The array [7, 8, 7, 2, 6] should be partitioned with the pivot strategy “right element”. (I marked the second 7 as 7′ to distinguish it from the first one).



The first element from the left that is greater than 6 is the first 7. The first element from the right that is smaller than 6 is the 2. So the first 7 and the 2 must be swapped:



The first 7 is no longer ahead, but behind the second 7 (7′). This remains so even after the first element of the right partition (the 8) has been swapped with the pivot element (the 6):



**Quicksort is, therefore, not stable.**

### The Selection of Pivot Element

In the previous example, I selected the last element of a (sub)array as the pivot element. This strategy makes the algorithm particularly simple, but it can harm performance.

#### Advantage of the “Last Element” Pivot Strategy

The advantage is, as mentioned above, a simplified algorithm:

Since the pivot element is guaranteed to be in the right section in this strategy, we do not need to consider it in the comparison and exchange operations. Furthermore, in the final step of partitioning, we can safely swap the first element of the right section with the pivot element to set it to its final position.

#### Disadvantage of the “Last Element” Pivot Strategy

In practice, the strategy leads to problems with presorted input data. In an array sorted in ascending order, the pivot element would be the largest element in each iteration.

The array would no longer be split into two partitions of as equal size as possible, but into an empty one (since no element is larger than the pivot element), and one of the length n-1 (with all elements except the pivot element).

This would decrease performance significantly (see section [“Quicksort Time Complexity”](https://www.happycoders.eu/algorithms/quicksort/#Quicksort_Time_Complexity)).

With input data sorted in descending order, the pivot element would always be the smallest element, so partitioning would also create an empty partition and one of size n-1.

### Alternative Pivot Strategies

Alternative strategies for selecting the pivot element include:

* the middle element,
* a random element,
* the median of three, five, or more elements.

If you choose the pivot element in one of these ways, the probability increases that the subarrays resulting from the partitioning are as equally large as possible.

In the course of the article, I will explain how the choice of pivot strategy affects performance.

#### Why Not the Median?

In the best case, the pivot element divides the array into two equally sized parts. Then why not choose the median of all elements as the pivot element?

For the following reason: For determining the median, the array would first have to be sorted. But we are only just defining the sorting algorithm – so there is no way to access the median yet.

Example:

**Partitioning with Pivot point at first position (Alternative to the last element)**

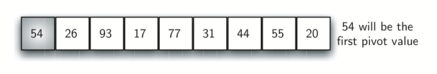


Figure 12: The First Pivot Value for a Quick Sort

Partitioning begins by locating two position markers—let’s call them leftmark and rightmark—at the beginning and end of the remaining items in the list (positions 1 and 8 in [Figure 13](https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html#fig-partitiona)). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. [Figure 13](https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html#fig-partitiona) shows this process as we locate the position of 54.

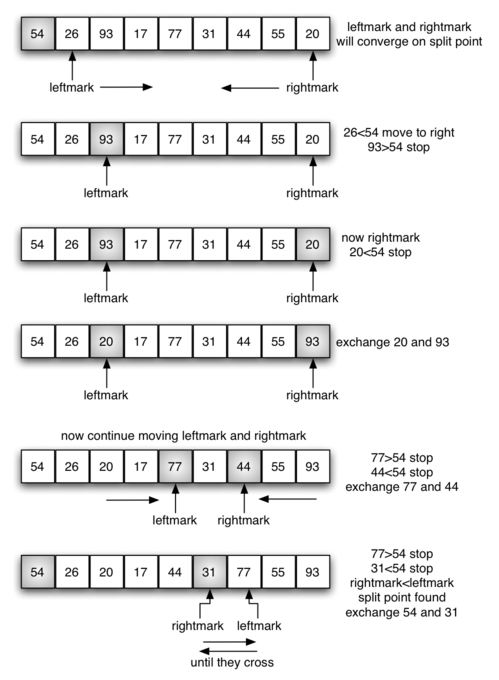


Figure 13: Finding the Split Point for 54

We begin by incrementing leftmark until we locate a value that is greater than the pivot value. We then decrement rightmark until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.

At the point where rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place ([Figure 14](https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html#fig-partitionb)). In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.

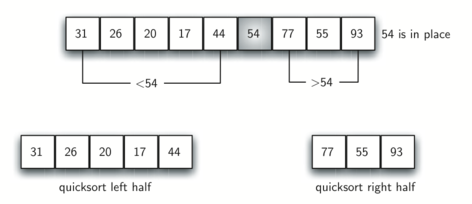


Figure 14: Completing the Partition Process to Find the Split Point for 54[¶](https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html#id3)

## Optimized Quicksort (Self Study)

## 1. Combination With Insertion Sort

For very small arrays, Insertion Sort is faster than Quicksort. So these algorithms are often combined in practice. This means that (sub)arrays above a specific size are not further partitioned, but sorted with Insertion Sort.

## 2. Dual-Pivot Quicksort

Quicksort can be further optimized by using two pivot elements instead of one. When partitioning, the elements are then divided into:

* elements smaller than the smaller pivot element,
* elements greater than or equal to the smaller pivot element and smaller than the larger pivot element,
* elements larger than/equal to the larger pivot element.

Here too, we have different pivot strategies, for example:

* Left and right element: For presorted elements, this leads – analogous to the regular Quicksort – to two partitions remaining empty and one partition containing n-2 elements. This, in turn, results in quadratic time and StackOverflowExceptions even with comparatively small n.
* Elements at the positions “one third” and “two thirds”: This is comparable to the strategy “middle element” in the regular Quicksort.

**Quicksort Applications**

Quicksort is implemented when

* the programming language is good for recursion
* time complexity matters
* space complexity matters

**Merge Sort**

Merge sort is a sorting technique based on divide and conquer technique.

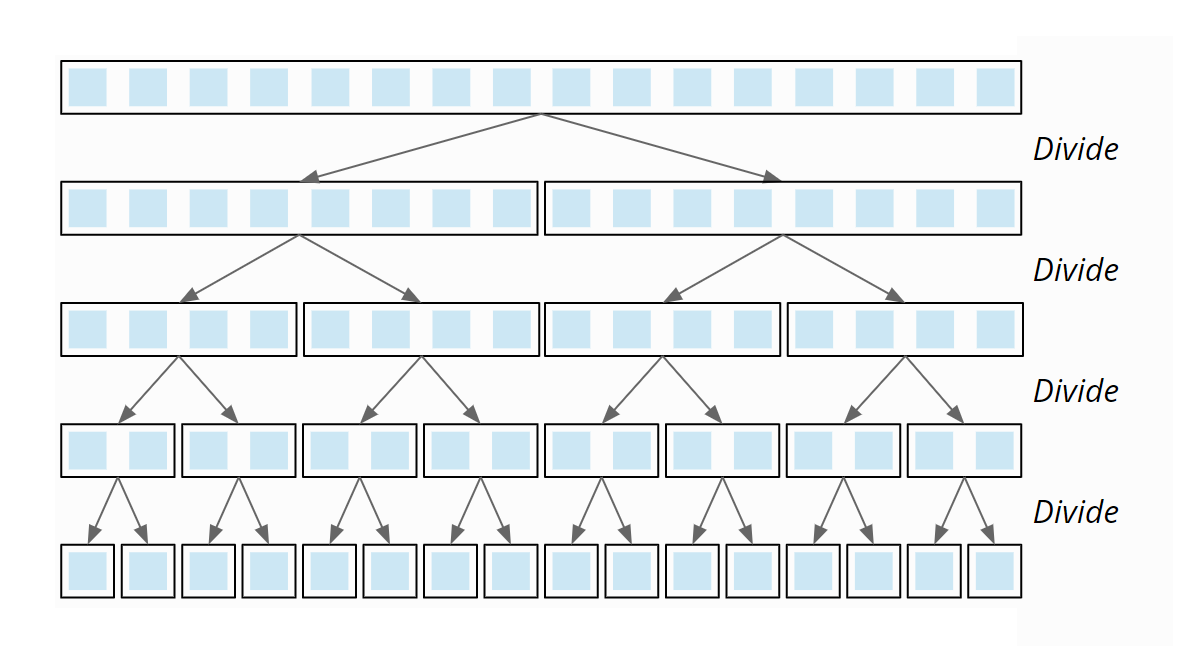
**With worst-case time complexity being Ο(n log n), it is one of the most respected algorithms.**

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

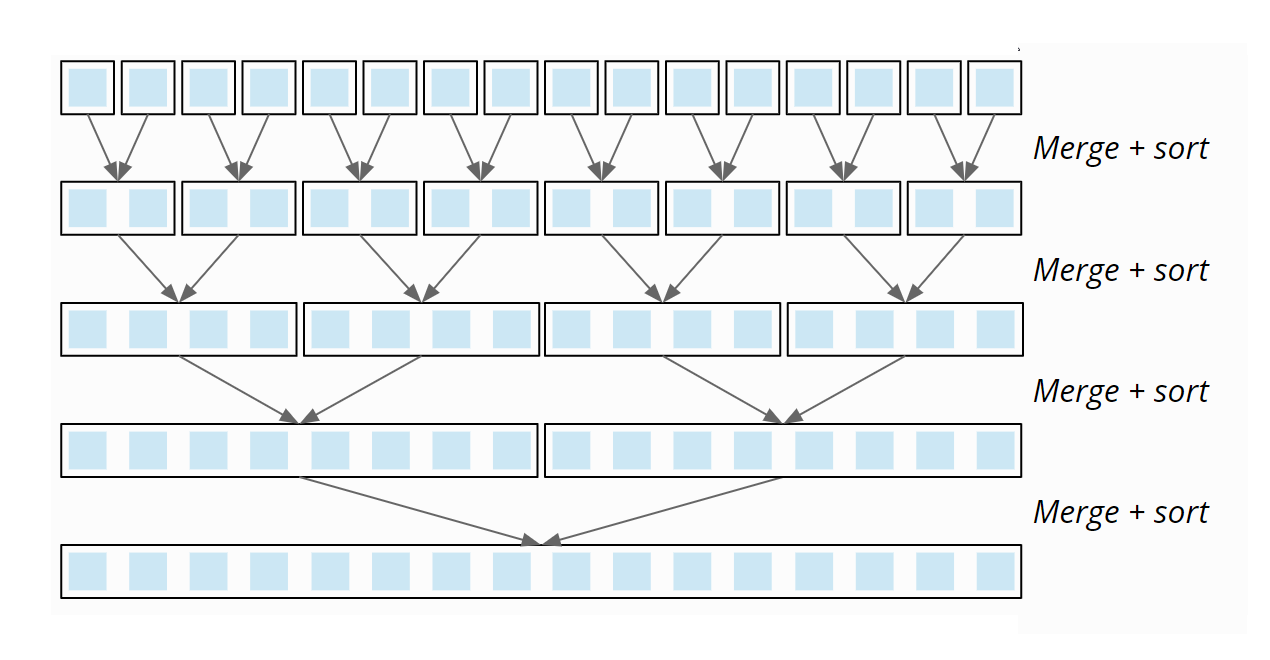
This does not change the sequence of appearance of items in the original.

## How Merge Sort Works?

First, we divide the elements to be sorted into two halves. The resulting subarrays are then divided again – and again until subarrays of length 1 are created:



Now two subarrays are merged so that a sorted array is created from each pair of subarrays. In the last step, the two halves of the original array are merged so that the complete array is sorted.



Example1: Given Array:

Unsorted Array

First divides the whole array iteratively into equal halves unless the atomic values are achieved.

Merge Sort Division

Merge Sort Division

Further divide these arrays and we achieve atomic value which can no more be divided.

Merge Sort Division

Combine them in sorted order (exactly the same manner as they were broken down)

Merge Sort Combine

Continue combining until all data values placed in a sorted order.

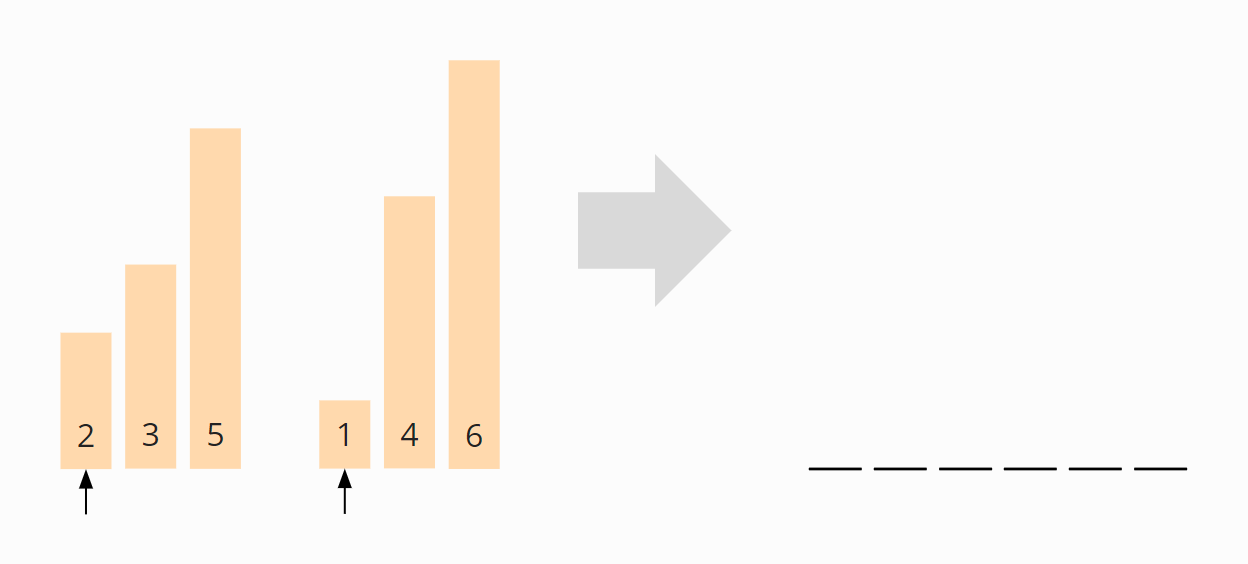
Merge Sort Combine

After the final merging, the list should look like this −

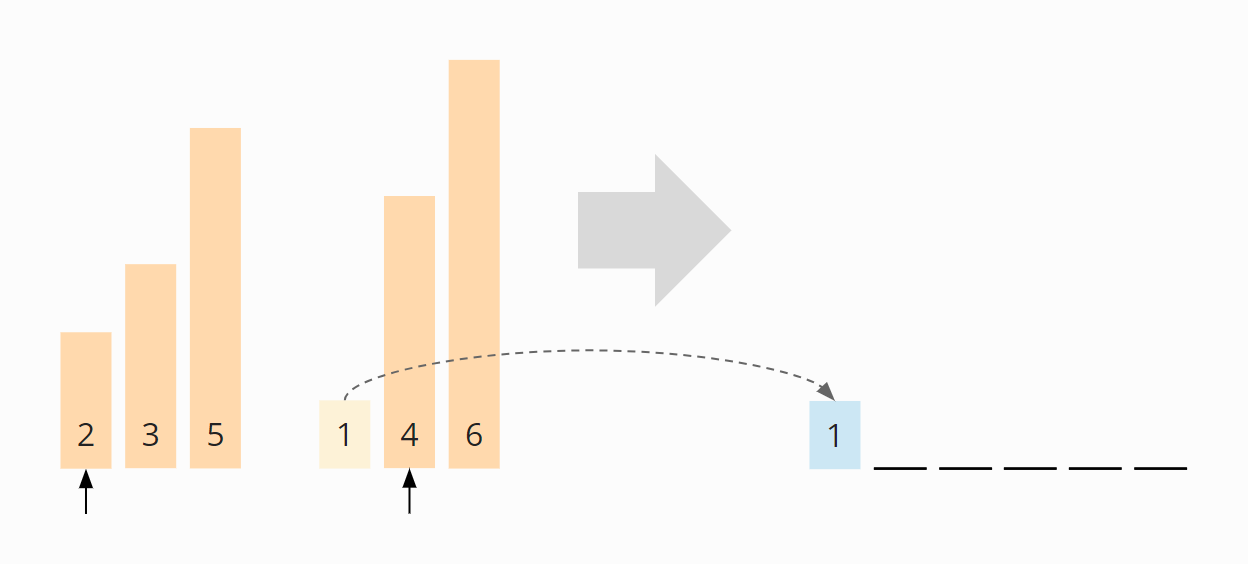
Merge Sort

**Example of Merging :**

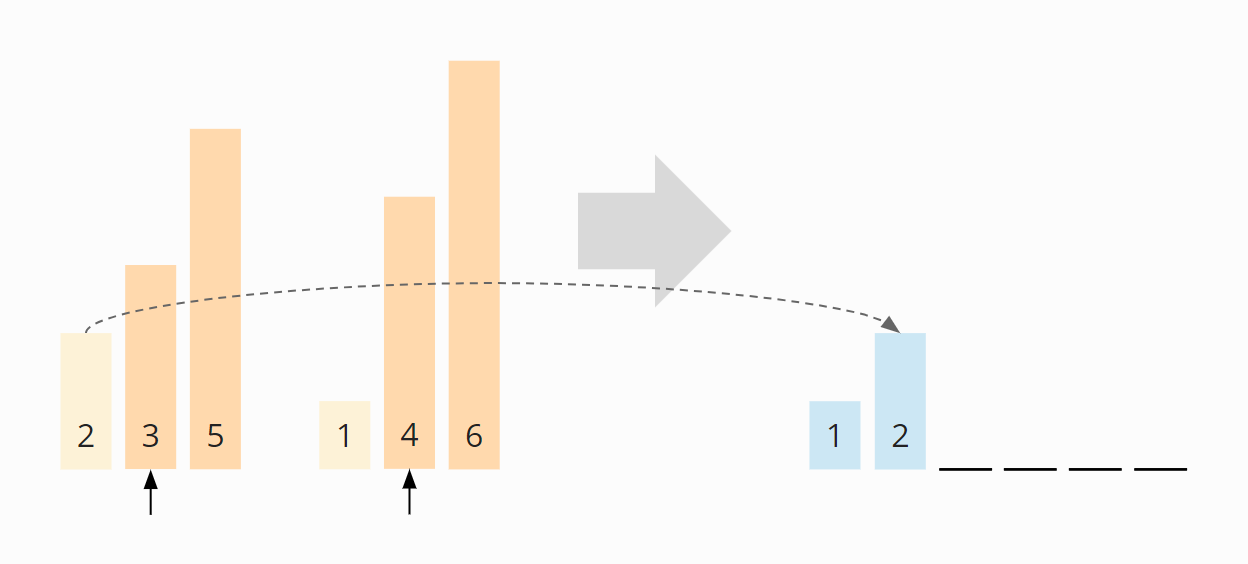
The merging itself is simple: For both arrays, we define a merge index, which first points to the first element of the respective array. The easiest way to show this is to use an example (the arrows represent the merge indexes):



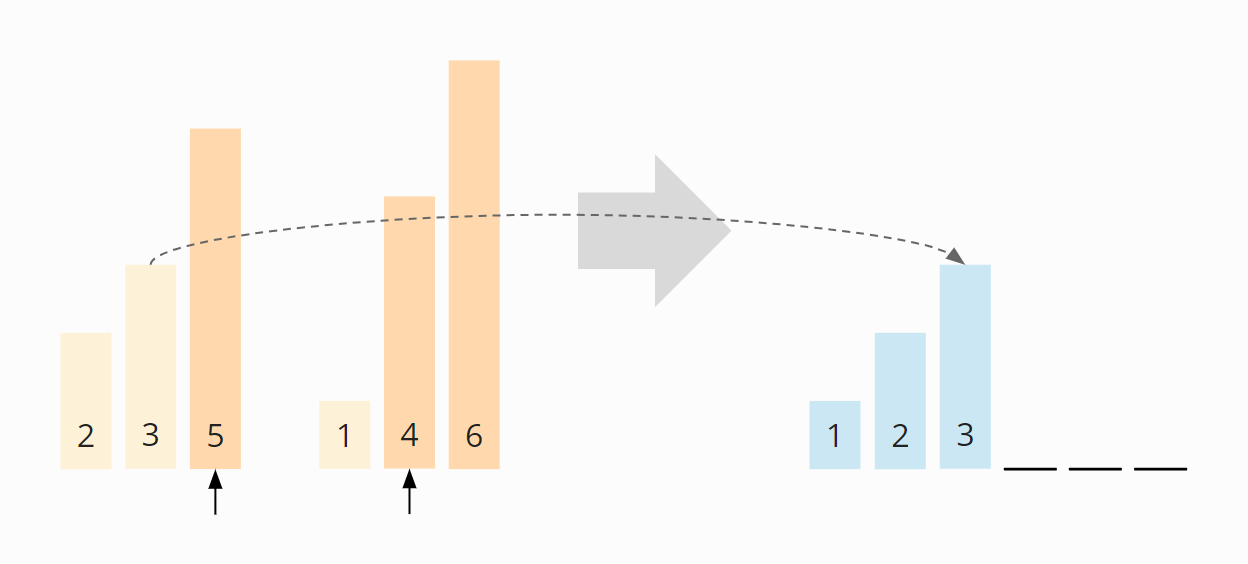
The elements over the merge pointers are compared. The smaller of the two (1 in the example) is appended to a new array, and the pointer to that element is moved one field to the right:



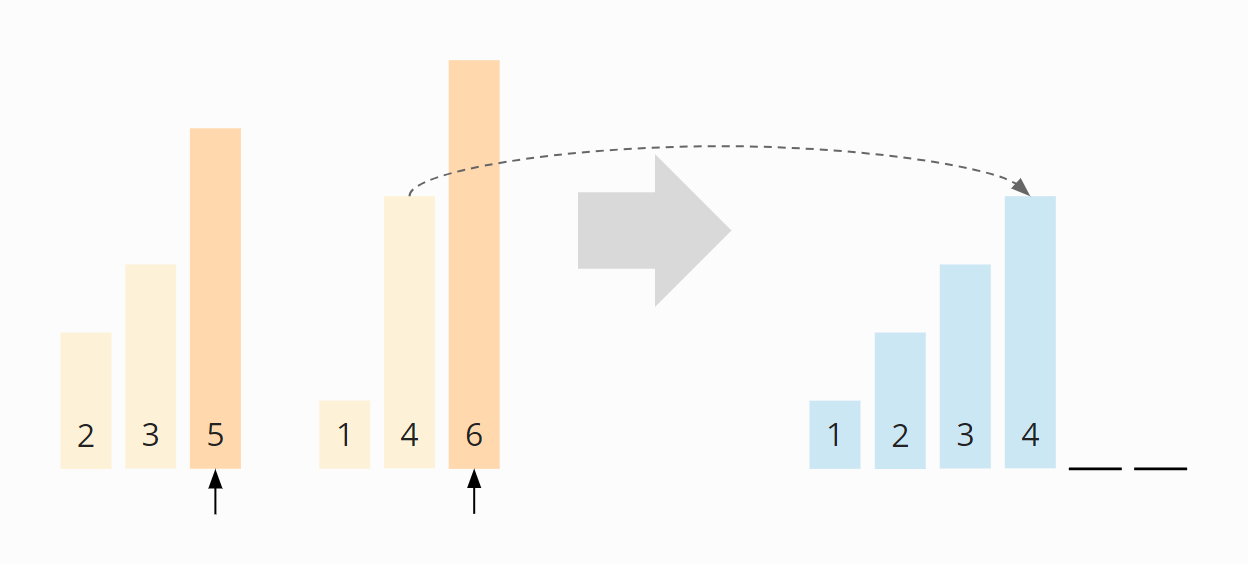
Now the elements above the pointers are compared again. This time the 2 is smaller than the 4, so we append the 2 to the new array:



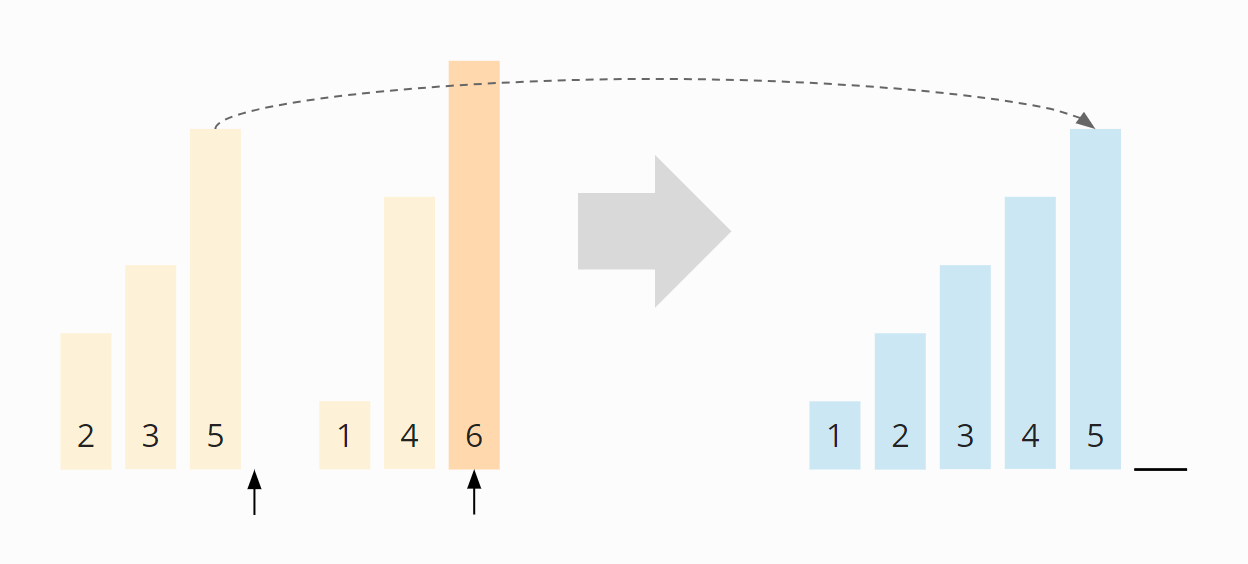
Now the pointers are on the 3 and the 4. The 3 is smaller and is appended to the target array:



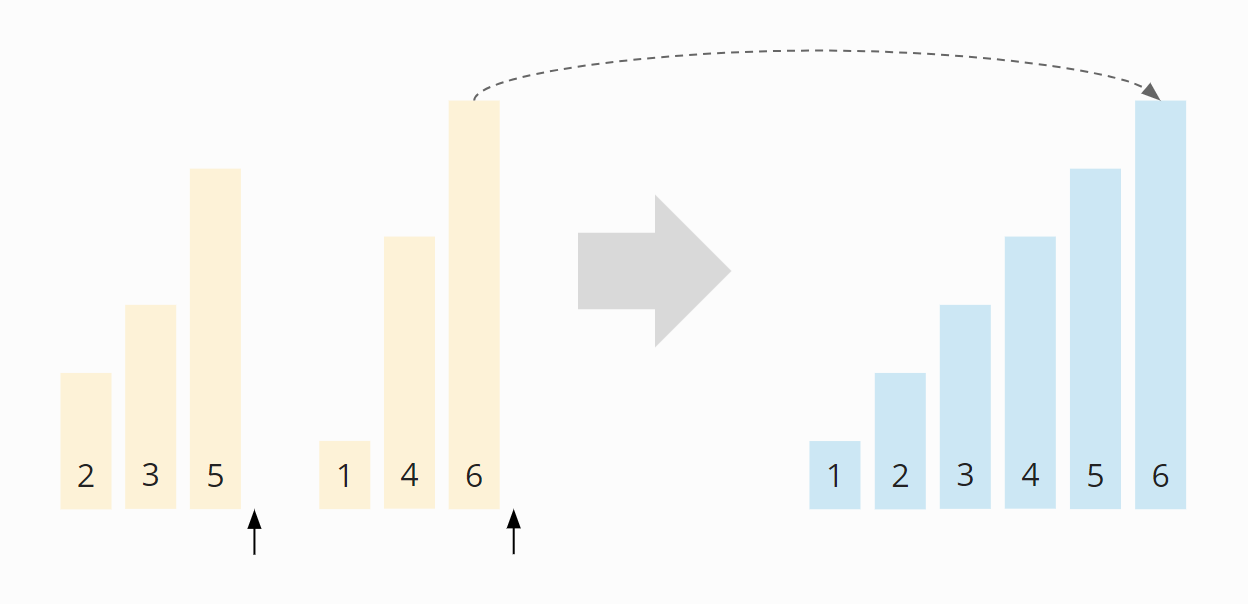
Now the 4 is the smallest element:



Now the 5:



And in the final step, the 6 is appended to the new array:

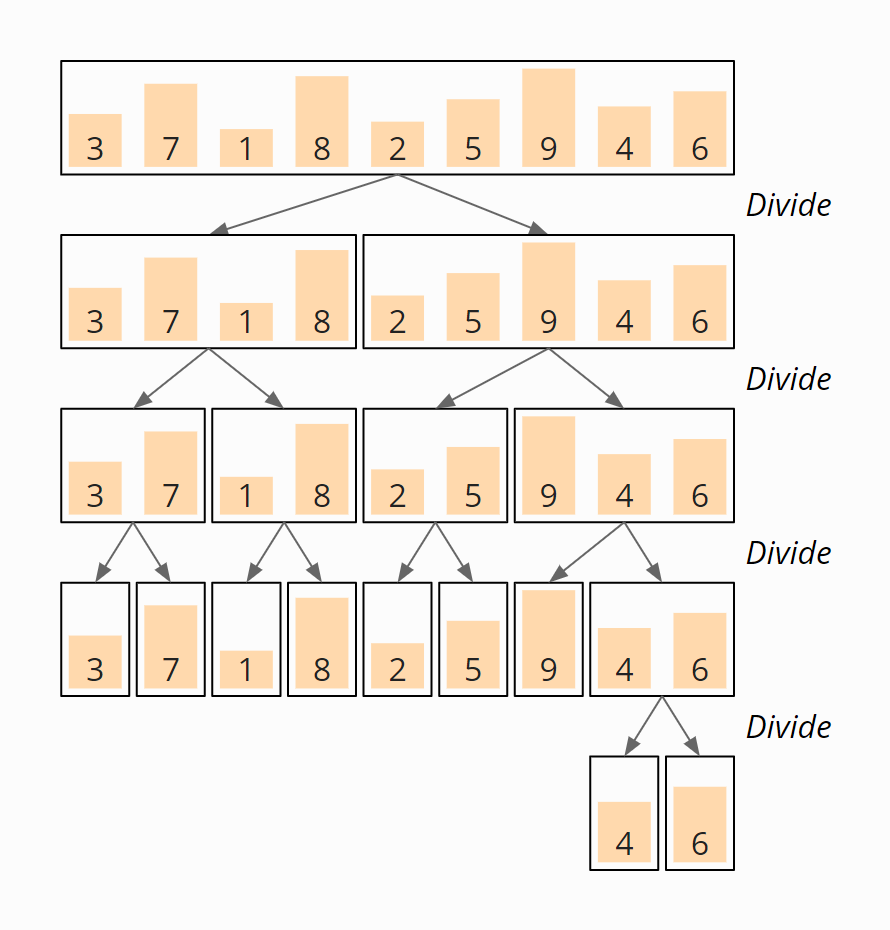


The two sorted subarrays were merged to the sorted final array.

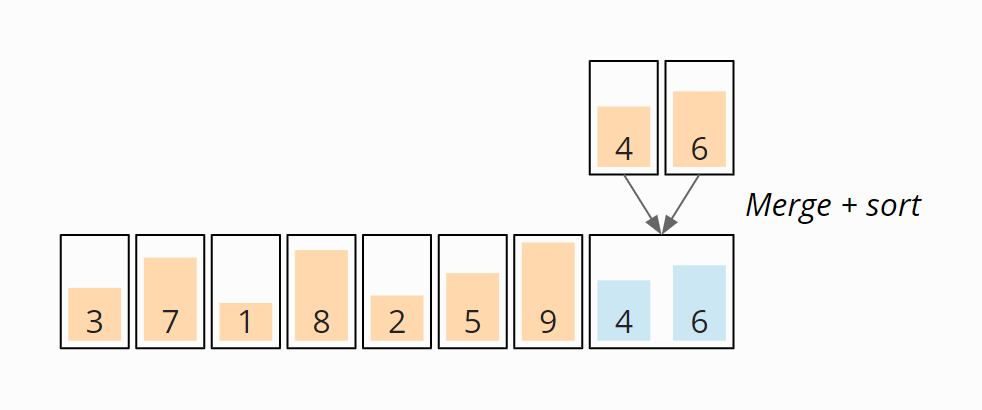
**One More Example (Odd numbered list):**

We want to sort the array [3, 7, 1, 8, 2, 5, 9, 4, 6] known from the previous parts of the series.

The array is divided until arrays of length 1 are created. The order of the elements does not change:

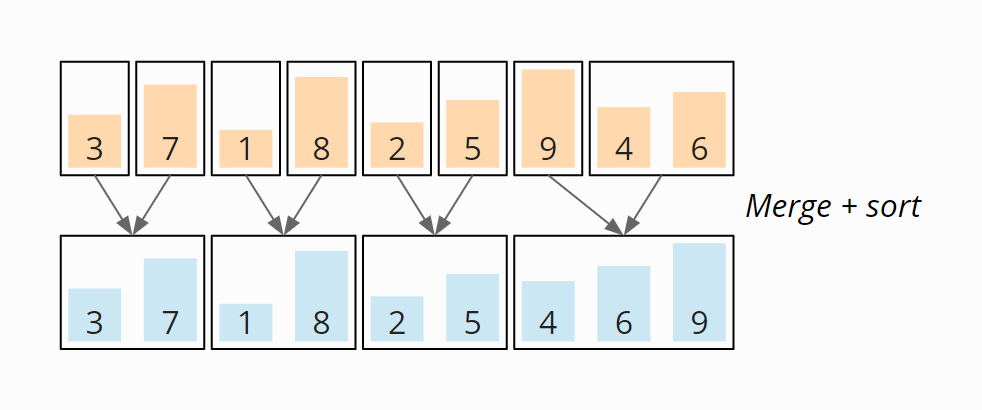


Now the subarrays are merged in the reverse direction according to the principle described above. In the first step, the 4 and the 6 are merged to the subarray [4, 6]:

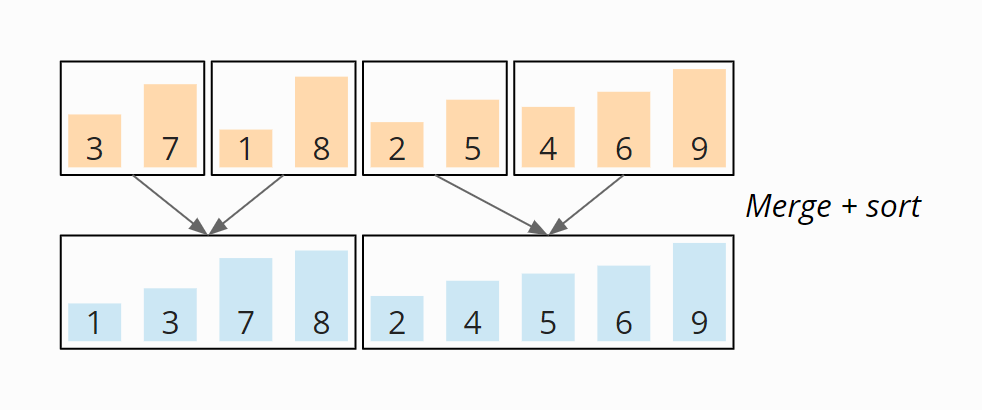


Next, the 3 and the 7 are merged to the subarray [3, 7], 1 and 8 to the subarray [1, 8], the 2 and the 5 become [2, 5]. Up to this point, the merged elements were coincidentally in the correct order and were therefore not moved.

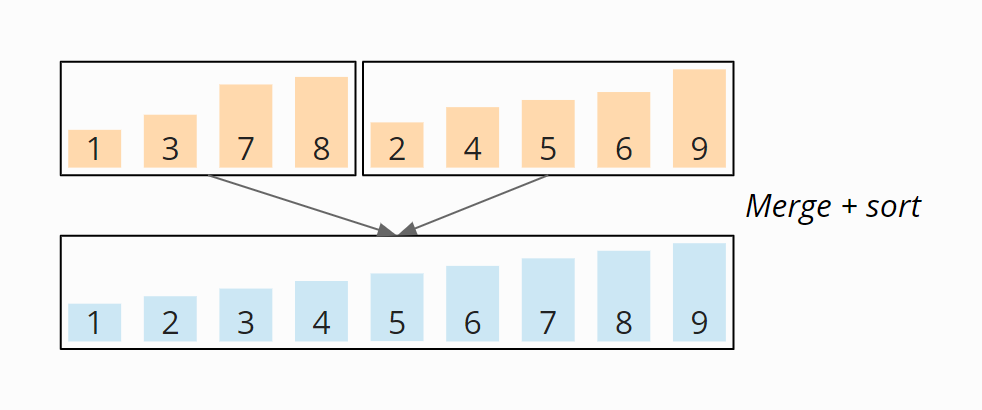
That’s changing now: The 9 is merged with the subarray [4, 6] – moving the 9 to the end of the new subarray [4, 6, 9]:



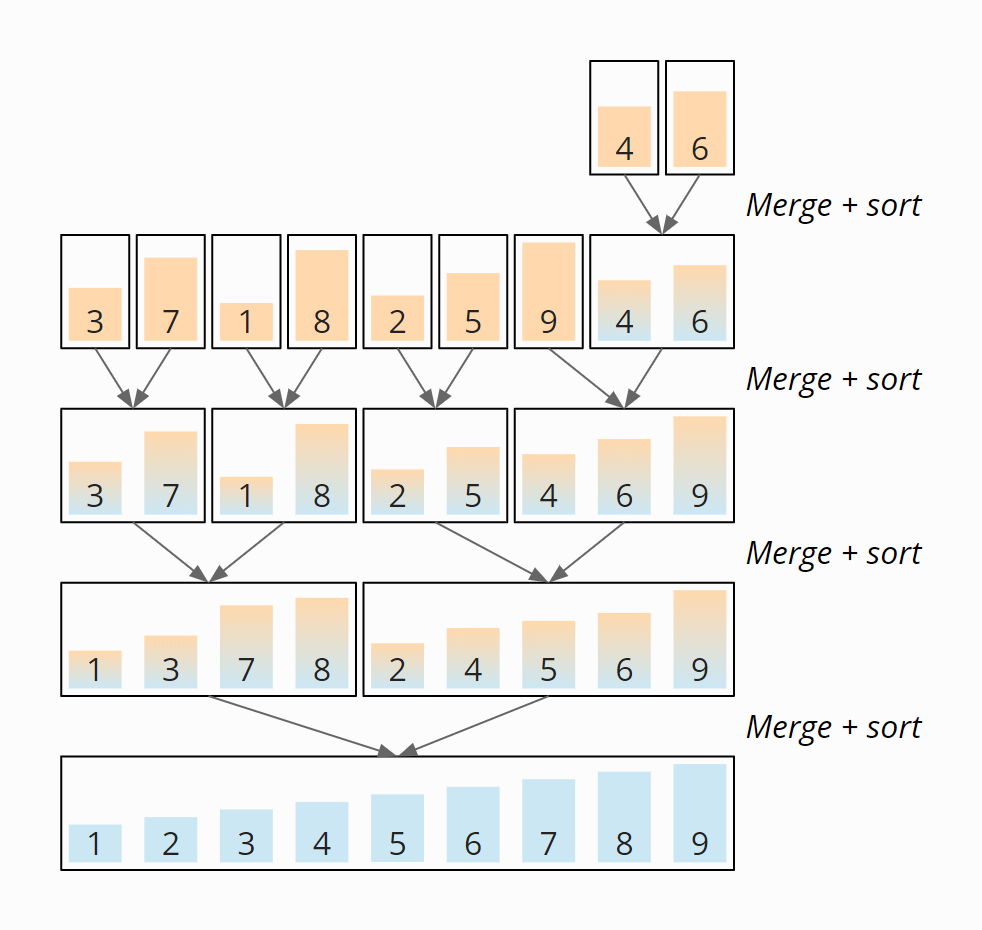
[3, 7] and [1, 8] are now merged to [1, 3, 7, 8]. [2, 5] and [4, 6, 9] become [2, 4, 5, 6, 9]:



And in the last step, the two subarrays [1, 3, 7, 8] and [2, 4, 5, 6, 9] are merged to the final result:



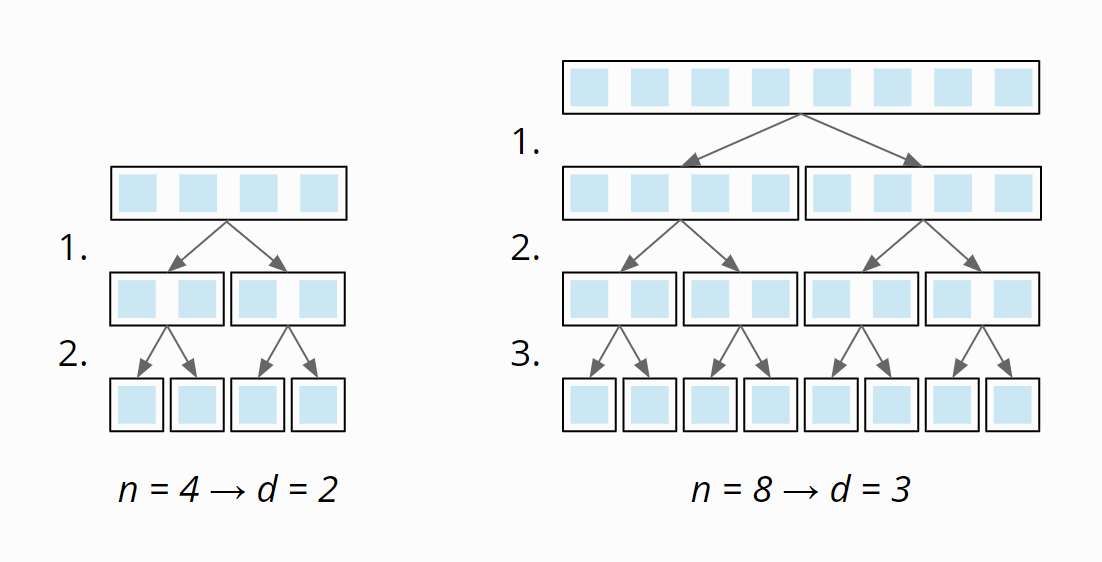
In the end, we get the sorted array [1, 2, 3, 4, 5, 6, 7, 8, 9]. The following diagram shows all merge steps summarized in an overview:



## Merge Sort Time Complexity

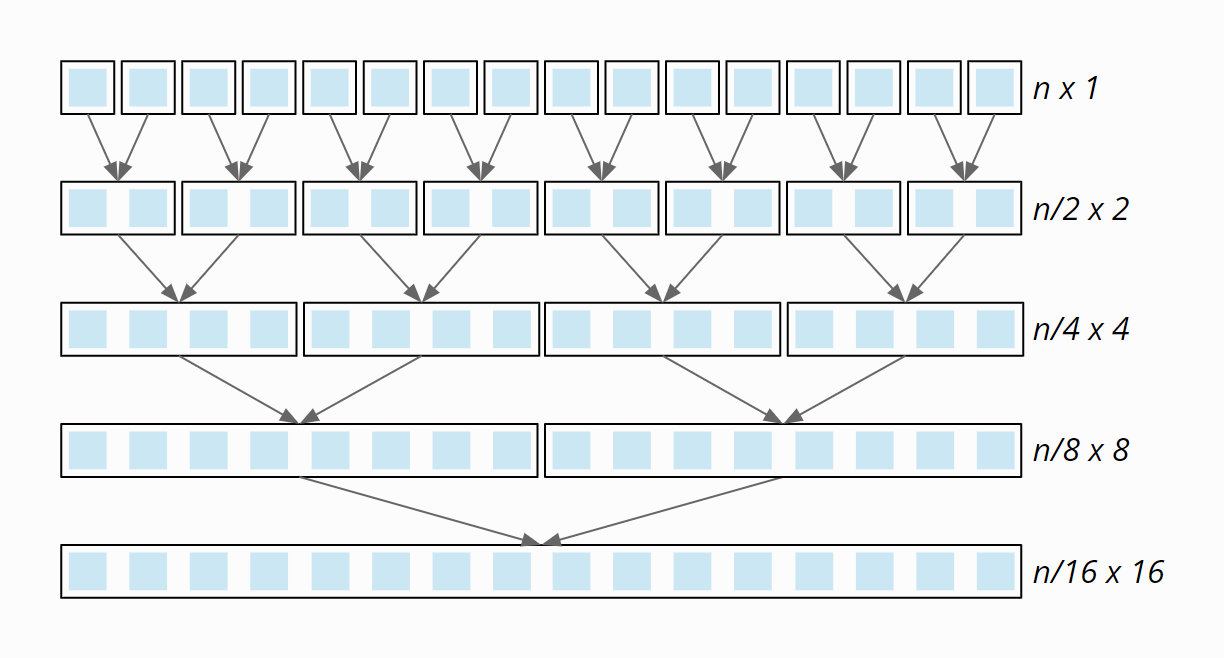
We denote with n the number of elements; in our example n = 6.

Since we repeatedly divide the (sub)arrays into two equally sized parts, if we double the number of elements n, we only need one additional step of divisions d. The following diagram demonstrates that for four elements, two division steps are needed, and for eight elements, only one more:



**Thus the number of division stages is**log2 n**.**

On each merge stage, we have to merge a total of n elements (on the first stage n × 1, on the second stage n/2 × 2, on the third stage n/4 × 4, etc.):



The merge process does not contain any nested loops, so it is executed with linear complexity:

* If the array size is doubled, the merge time doubles, too. The total effort is, therefore, the same at all merge levels.

So we have n elements times log2 n division and merge stages. Therefore:

The time complexity of Merge Sort is: O(n log n)

And that is regardless of whether the input elements are presorted or not. Merge Sort is therefore no faster for sorted input elements than for randomly arranged ones.

Basic **Algorithm**

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

**Step 1** − if it is only one element in the list it is already sorted, return.

**Step 2** − divide the list recursively into two halves until it can no more be divided.

**Step 3** − merge the smaller lists into new list in sorted order.

### Detailed Algorithm

**Main ()**

1. length = elements.length;

2, sorted = mergeSort(elements, 0, length - 1);

3, print sorted

4. end

**mergeSort( elements, left, right)**

// End of recursion reached?

1. if (left == right)

return (elements[left])

2. middle = left + (right - left) / 2;

3. leftArray = mergeSort(elements, left, middle);

4. rightArray = mergeSort(elements, middle + 1, right);

5. return merge(leftArray, rightArray);

merge( leftArray, rightArray)

1. leftLen = leftArray.length;

2. rightLen = rightArray.length;

3. target = new int[leftLen + rightLen];

4. targetPos = 0;

5. leftPos = 0;

6.rightPos = 0;

// As long as both arrays contain elements...

7. Repeat step 8 to 10 while (leftPos < leftLen && rightPos < rightLen) {

// Which one is smaller?

8. leftValue = leftArray[leftPos];

9. rightValue = rightArray[rightPos];

10. if (leftValue <= rightValue)

target[targetPos++] = leftValue;

leftPos++;

else

target[targetPos++] = rightValue;

rightPos++;

// Copy the rest

11. repeat step 12 while (leftPos < leftLen)

12. target[targetPos++] = leftArray[leftPos++];

13. repeat step 14 while (rightPos < rightLen) {

14 target[targetPos++] = rightArray[rightPos++];

return target;

## Other Characteristics of Merge Sort

**Space Complexity of Merge Sort**

* In the merge phase, elements from two subarrays are copied into a newly created target array.
* In the very last merge step, the target array is exactly as large as the array to be sorted.
* Thus, we have a linear space requirement:
* If the input array is twice as large, the additional storage space required is doubled. Therefore:

The space complexity of Merge Sort is: O(n)

### Stability of Merge Sort

Merge Sort is stable sorting

### Parallelizability of Merge Sort

There are basically two approaches to parallelize Merge Sort:

* Recursive calls of mergeSort() can be executed in parallel; however, today’s multi-core CPUs cannot be fully utilized in the final merge stages.
* The merge() method itself can be parallelized.

## In-Place Merge Sort (Self study)

One approach is the following:

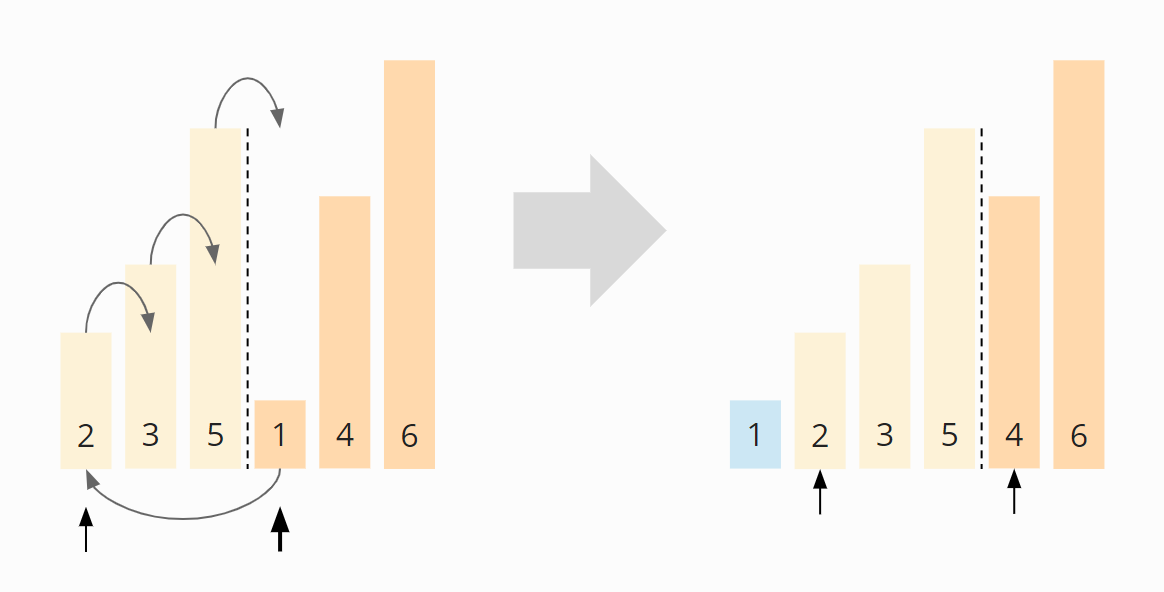
* If the element above the left merge pointer is less than or equal to the element above the right merge pointer, the left merge pointer is moved one field to the right.
* Otherwise, all elements from the first pointer to, but excluding, the second pointer are moved one field to the right, and the right element is placed in the field that has become free. Then both pointers are shifted one field to the right, as well as the end position of the left subarray.

### In-Place Merge Sort – Example

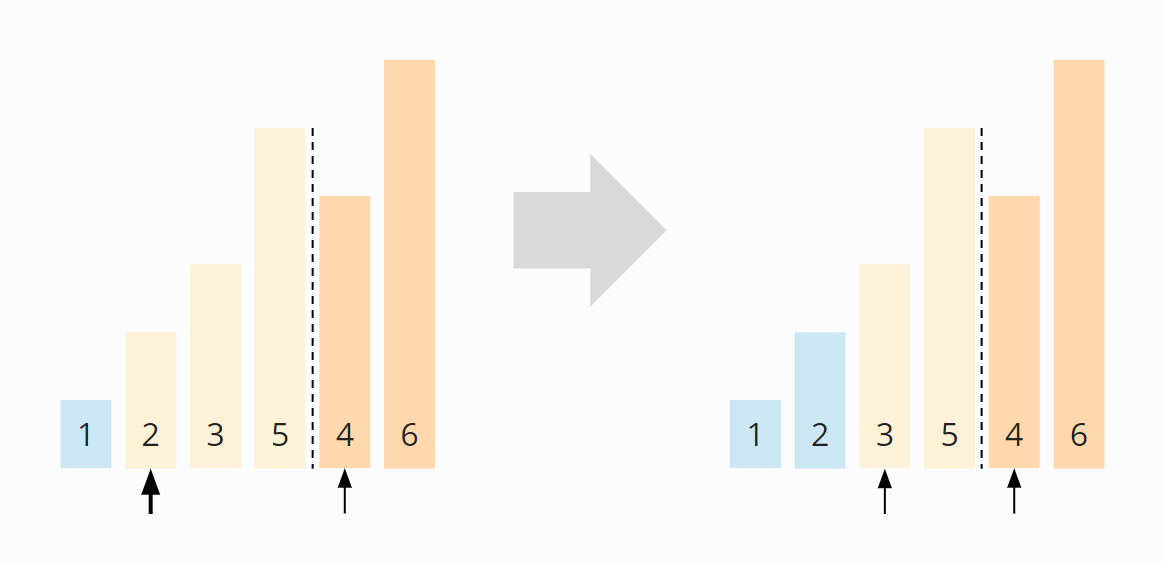
The following example shows this in-place merge algorithm using the example from above – merging the subarrays [2, 3, 5] and [1, 4, 6].

The left part array is colored yellow, the right one orange, and the merged elements blue.

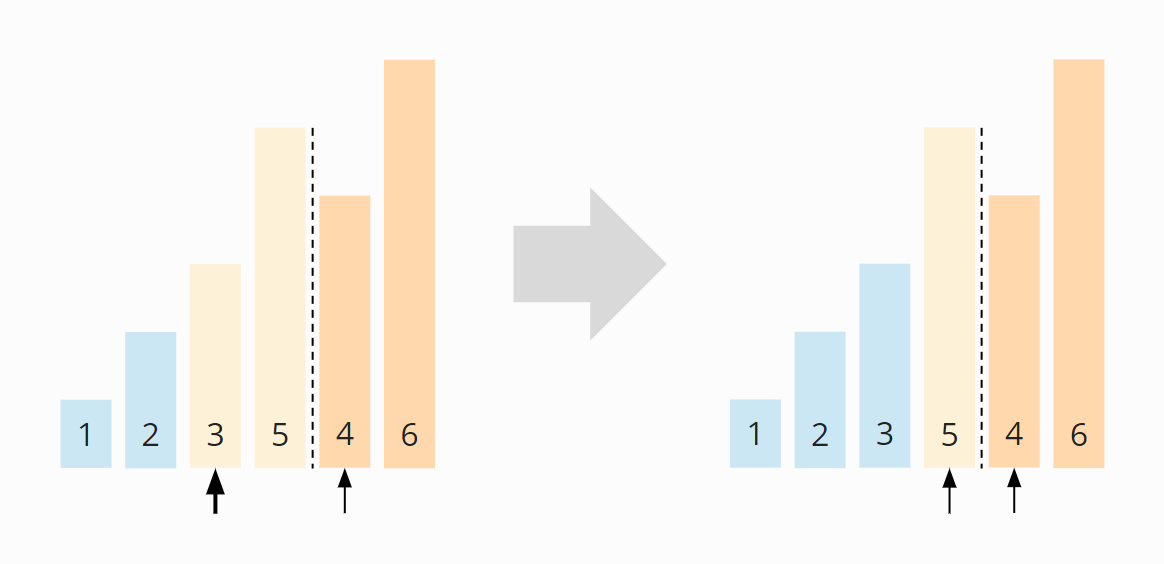
In the first step, the second case occurs right away: The right element (the 1) is smaller than the left one. Therefore, all elements of the left subarray are shifted one field to the right, and the right element is placed at the beginning:



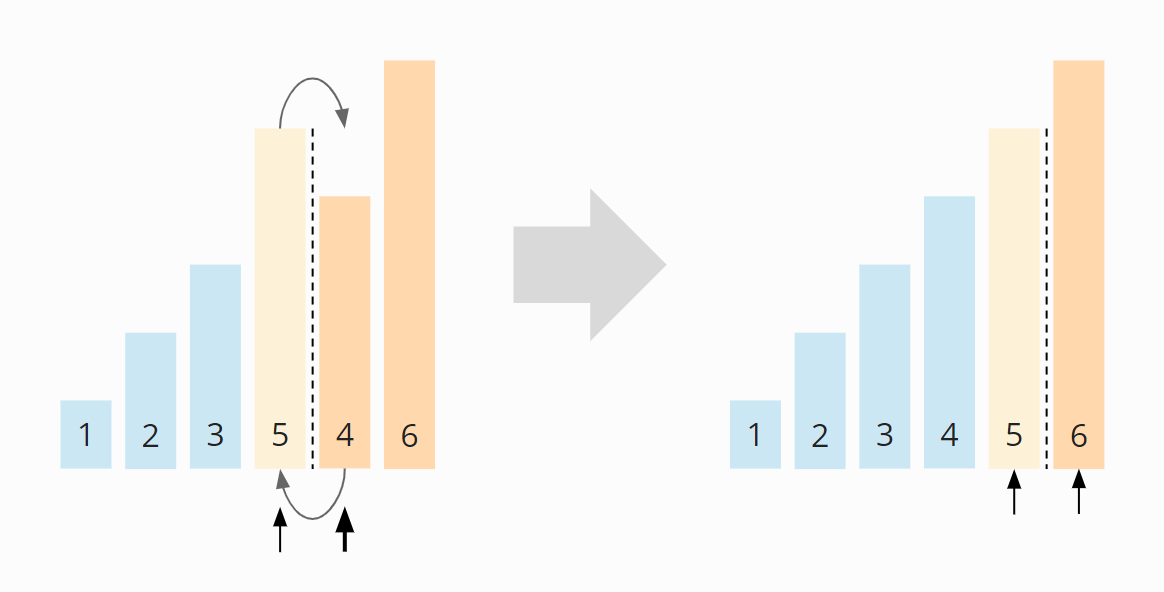
In the second step, the left element (the 2) is smaller, so the left search pointer is moved one field to the right:



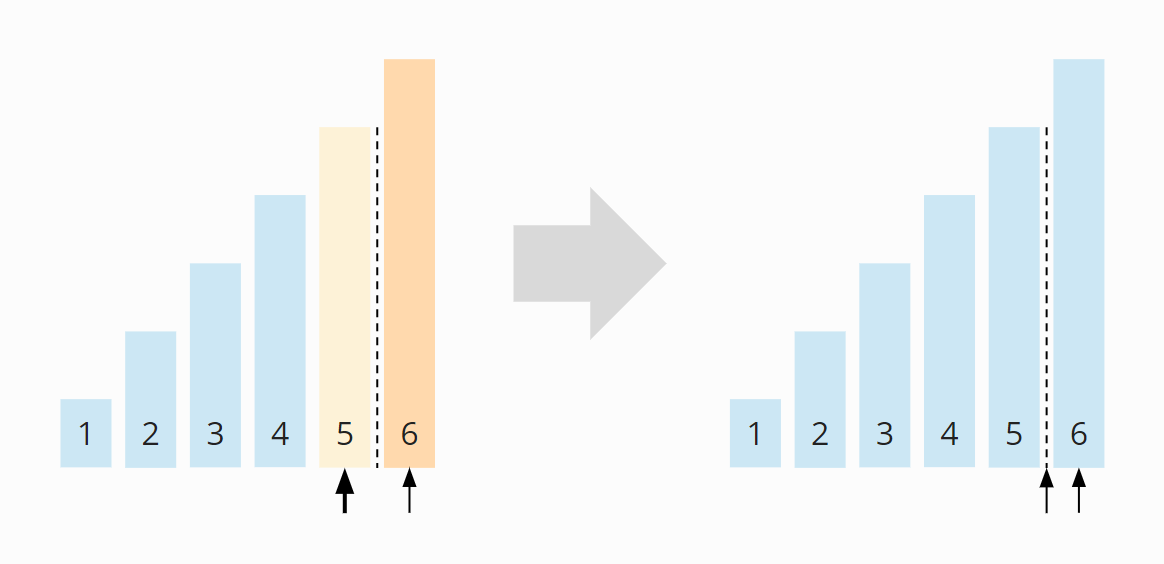
In the third step, again, the left element (the 3) is smaller, so we move the left search pointer once more:



In the fourth step, the right element (the 4) is smaller than the left one. So the remaining part of the left area (only the 5) is moved one field to the right, and the right element is placed on the free field:



In the fifth step, the left element (the 5) is smaller. The left search pointer is moved one position to the right and has thus reached the end of the left section:



The in-place merge process is now complete.

### In-Place Merge Sort – Time Complexity

We have now executed the merge phase without any additional memory requirements – but we have paid a high price: Due to the two nested loops, the merge phase now has an average and worst-case time complexity of O(n²) – instead of previously O(n).

The total complexity of the sorting algorithm is, therefore, O(n² log n) – instead of O(n log n). The algorithm is, therefore, no longer efficient.

Only in the best case, when the elements are presorted in ascending order, the time complexity within the merge phase remains O(n) and that of the overall algorithm O(n log n). In this case, the inner loop, which shifts the elements of the left subarray to the right, is never executed.