<https://www.happycoders.eu/algorithms/quicksort/>

# Quicksort – Algorithm, Source Code, Time Complexity

In this [article series on sorting algorithms](https://www.happycoders.eu/algorithms/sorting-algorithms/), after three relatively easy-to-understand methods ([Insertion Sort](https://www.happycoders.eu/algorithms/insertion-sort/), [Selection Sort](https://www.happycoders.eu/algorithms/selection-sort/), [Bubble Sort](https://www.happycoders.eu/algorithms/bubble-sort/)), we come to the more complex – and much more efficient algorithms.

We start with Quicksort ("Sort" is not a separate word here, so not "Quick Sort"). This article:

* describes the Quicksort algorithm,
* shows its Java source code,
* explains how to derive its time complexity,
* tests whether the performance of the Java implementation matches the expected runtime behavior,
* introduces various algorithm optimizations (combination with Insertion Sort and Dual-Pivot Quicksort)
* and measures and compares their speed.

You can find the source code for the article series in this [GitHub repository](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide" \t "https://www.happycoders.eu/algorithms/quicksort/_blank).

**Contents**  [hide](https://www.happycoders.eu/algorithms/quicksort/)

**[1 Quicksort Algorithm](https://www.happycoders.eu/algorithms/quicksort/" \l "Quicksort_Algorithm)**

[1.1 Quicksort Partitioning](https://www.happycoders.eu/algorithms/quicksort/" \l "Quicksort_Partitioning)

[1.2 The Pivot Element](https://www.happycoders.eu/algorithms/quicksort/" \l "The_Pivot_Element)

[1.3 Alternative Pivot Strategies](https://www.happycoders.eu/algorithms/quicksort/" \l "Alternative_Pivot_Strategies)

**[2 Quicksort Java Source Code](https://www.happycoders.eu/algorithms/quicksort/" \l "Quicksort_Java_Source_Code)**

[2.1 Source Code for Alternative Pivot Strategies](https://www.happycoders.eu/algorithms/quicksort/" \l "Source_Code_for_Alternative_Pivot_Strategies)

**[3 Quicksort Time Complexity](https://www.happycoders.eu/algorithms/quicksort/" \l "Quicksort_Time_Complexity)**

[3.1 Best-Case Time Complexity](https://www.happycoders.eu/algorithms/quicksort/" \l "Best-Case_Time_Complexity)

[3.2 Average-Case Time Complexity](https://www.happycoders.eu/algorithms/quicksort/" \l "Average-Case_Time_Complexity)

[3.3 Worst-case Time Complexity](https://www.happycoders.eu/algorithms/quicksort/" \l "Worst-case_Time_Complexity)

**[4 Java Quicksort Runtime](https://www.happycoders.eu/algorithms/quicksort/" \l "Java_Quicksort_Runtime)**

[4.1 Runtime Measurement of the Quicksort Algorithm Variants](https://www.happycoders.eu/algorithms/quicksort/" \l "Runtime_Measurement_of_the_Quicksort_Algorithm_Variants)

[4.2 Runtime Measurements for Different Pivot Strategies and Array Sizes](https://www.happycoders.eu/algorithms/quicksort/" \l "Runtime_Measurements_for_Different_Pivot_Strategies_and_Array_Sizes)

**[5 Quicksort Optimized: Combination With Insertion Sort](https://www.happycoders.eu/algorithms/quicksort/" \l "Quicksort_Optimized_Combination_With_Insertion_Sort)**

[5.1 Quicksort/Insertion Sort Source Code](https://www.happycoders.eu/algorithms/quicksort/" \l "QuicksortInsertion_Sort_Source_Code)

[5.2 Quicksort/Insertion Sort Performance](https://www.happycoders.eu/algorithms/quicksort/" \l "QuicksortInsertion_Sort_Performance)

**[6 Dual-Pivot Quicksort](https://www.happycoders.eu/algorithms/quicksort/" \l "Dual-Pivot_Quicksort)**

[6.1 Dual-Pivot Quicksort Source Code](https://www.happycoders.eu/algorithms/quicksort/" \l "Dual-Pivot_Quicksort_Source_Code)

[6.2 Dual-Pivot Quicksort Performance](https://www.happycoders.eu/algorithms/quicksort/" \l "Dual-Pivot_Quicksort_Performance)

[6.3 Dual-Pivot Quicksort Combined With Insertion Sort](https://www.happycoders.eu/algorithms/quicksort/" \l "Dual-Pivot_Quicksort_Combined_With_Insertion_Sort)

**[7 Comparing All Quicksort Optimizations](https://www.happycoders.eu/algorithms/quicksort/" \l "Comparing_All_Quicksort_Optimizations)**

**[8 Further Characteristics of Quicksort](https://www.happycoders.eu/algorithms/quicksort/" \l "Further_Characteristics_of_Quicksort)**

[8.1 Space Complexity of Quicksort](https://www.happycoders.eu/algorithms/quicksort/" \l "Space_Complexity_of_Quicksort)

[8.2 Stability of Quicksort](https://www.happycoders.eu/algorithms/quicksort/" \l "Stability_of_Quicksort)

[8.3 Parallelizability of Quicksort](https://www.happycoders.eu/algorithms/quicksort/" \l "Parallelizability_of_Quicksort)

**[9 Quicksort vs. Merge Sort](https://www.happycoders.eu/algorithms/quicksort/" \l "Quicksort_vs_Merge_Sort)**

**[10 Conclusion](https://www.happycoders.eu/algorithms/quicksort/" \l "Conclusion)**

## Quicksort Algorithm

Quicksort works according to the "divide and conquer" principle:

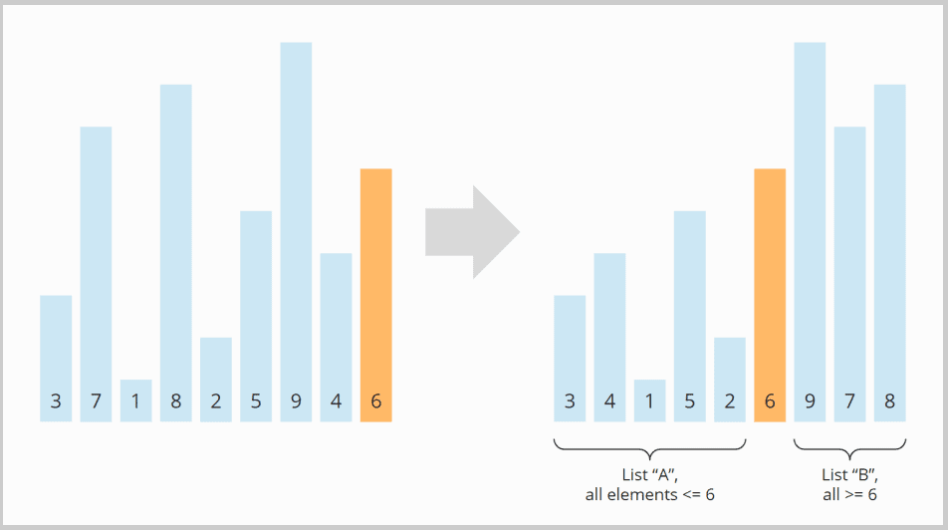
First, we divide the elements to be sorted into two sections - one with small elements ("A" in the following example) and one with large elements ("B" in the example).

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:

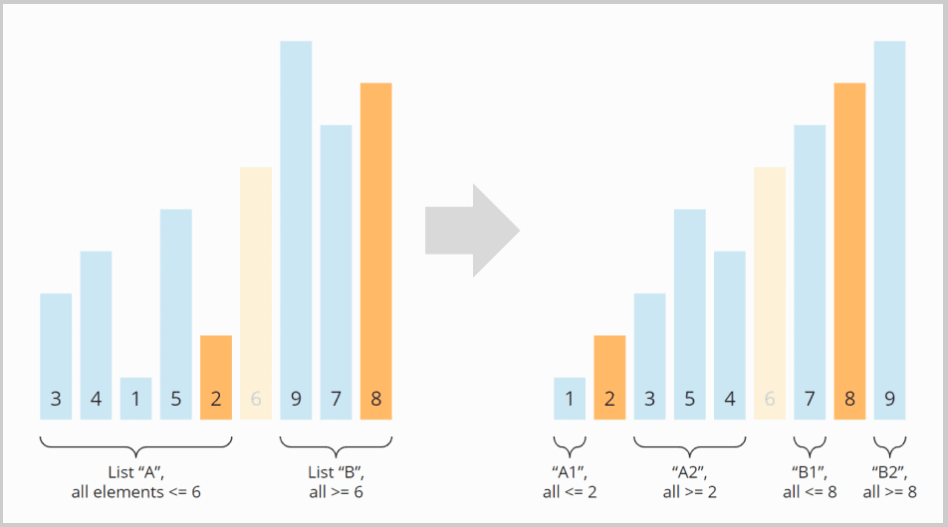
* the elements that are smaller than the pivot element end up in the left section,
* the elements that are larger than the pivot element end up in the right section,
* the pivot element is positioned between the two sections - which also is its final position.

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):

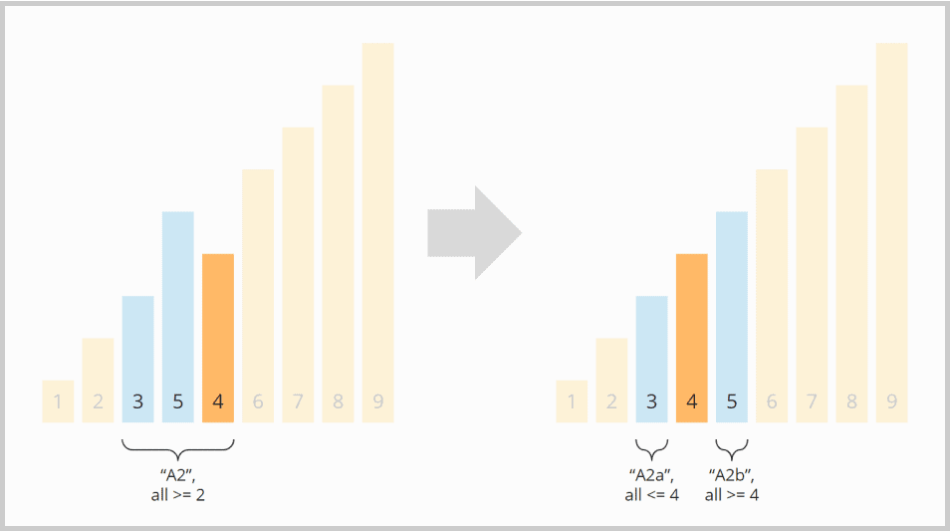


This division into two subarrays is called partitioning. You will learn precisely how partitioning works in the next section. Before that, I will show you how the higher-level algorithm continues.

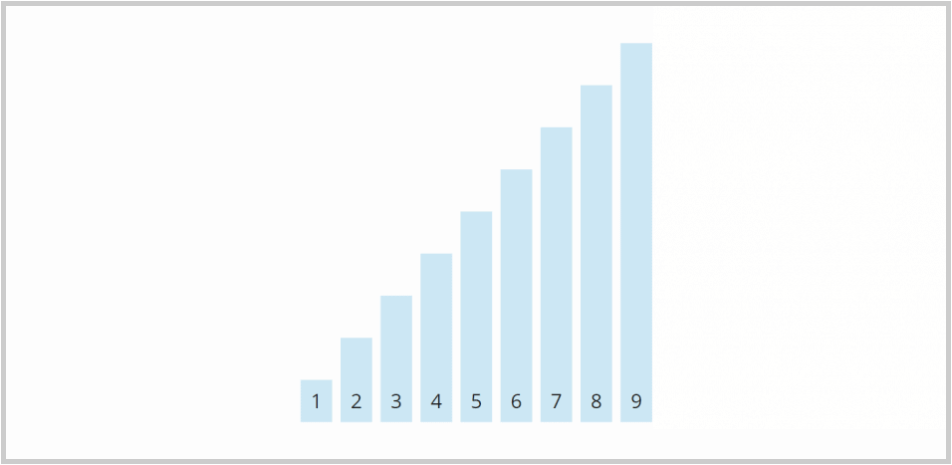
The subarrays to the left and right of the pivot element are still unsorted after partitioning. These subarrays will now also bo partitioned. I drew the pivot element from the previous step, the 6, semi-transparent to make the two subarrays easier to recognize:



After partitioning again, we 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.

The next section will explain how the division of an array into two sections – the partitioning – works.

### Quicksort Partitioning

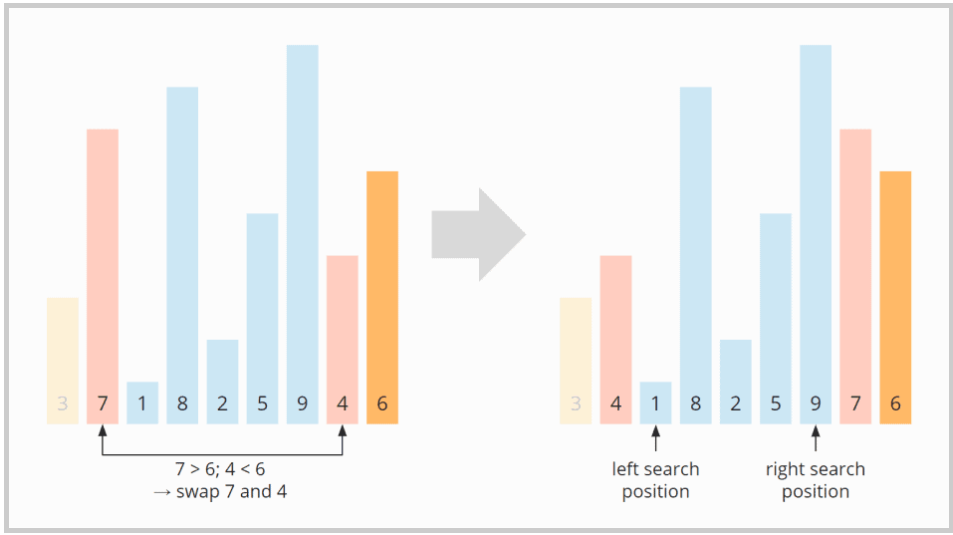
We divide the array into two partitions by searching for elements larger than the pivot element starting from the left – and for elements smaller than the pivot element starting from the right.

These elements are then swapped with each other. We repeat this until the left and right search positions have met or passed each other.

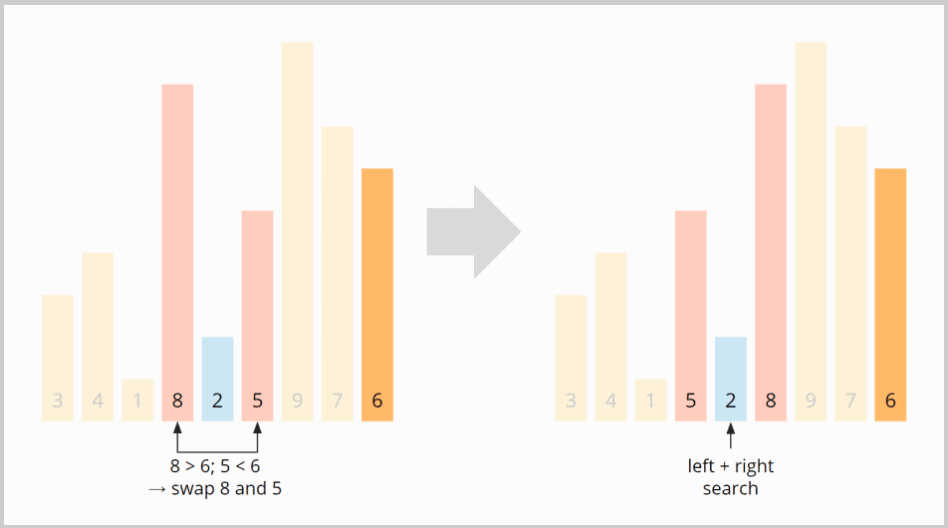
In the example from above this works as follows:

* The first element from the left, which is larger than pivot element 6, is 7.
* The first element from the right, which is smaller than the 6, is the 4.
* We swap the 7 and the 4.

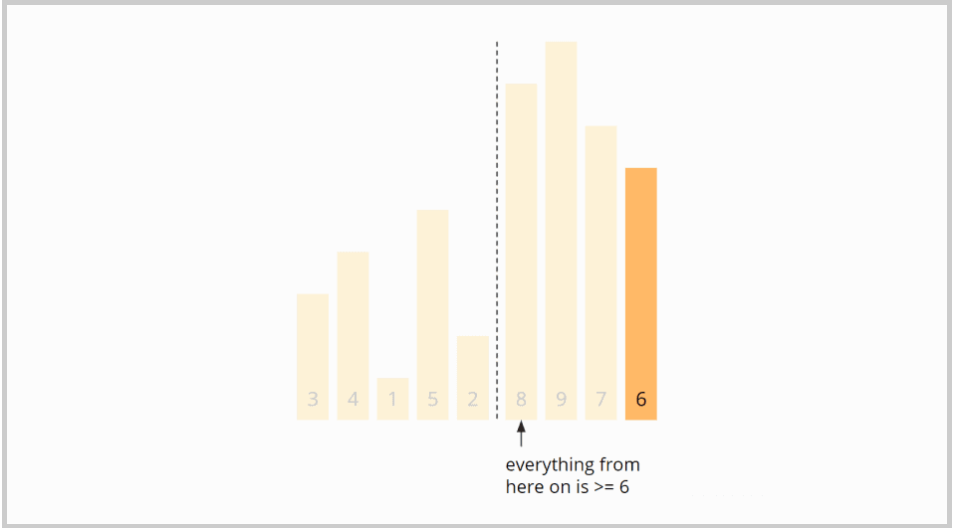
The 3 was already on the correct side (less than 6, so on the left). I filled it with a weaker color because we don’t have to look at it any further.



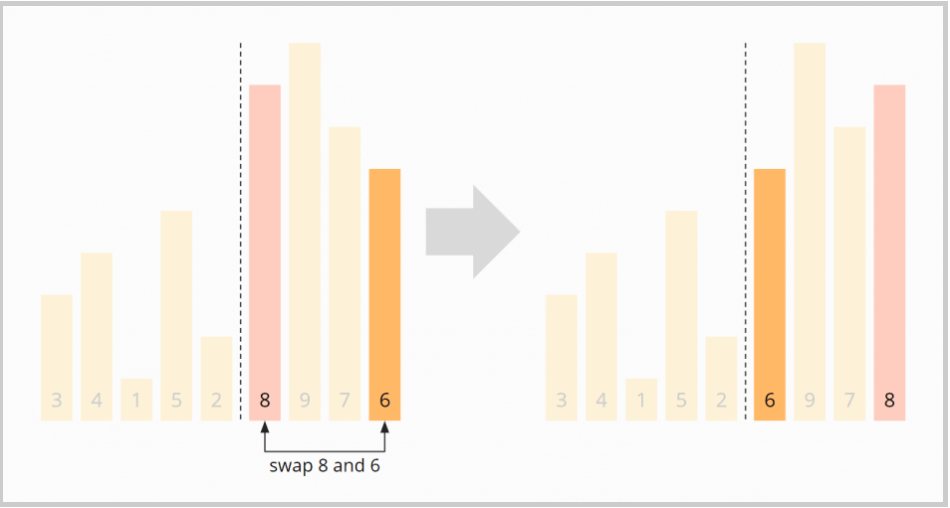
We continue searching and find the 8 from the left (the 1 is already on the correct side as it's less than 6) and the 5 from the right (the 9 is also already on the correct side as it's greater than 6). We swap the 8 and the 5:



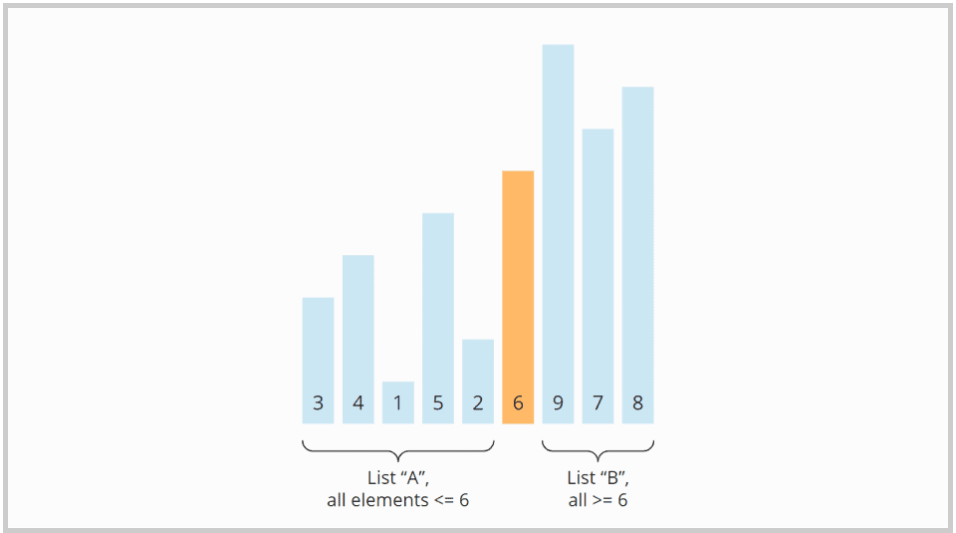
Now the left and right search positions meet at the 2. The swapping ends here. Since the 2 is smaller than the pivot element, we move the search pointer one more field to the right, to the 8, so that all elements from this position on are greater than or equal to the pivot element, and all elements before it are smaller:



To put the pivot element at the beginning of the right partition, we swap the 8 with the 6:



The partitioning is complete: The 6 is in the correct position, the numbers to the left of the 6 are smaller, and the numbers to the right are larger. So we have reached the state that was shown in the previous section after the first partitioning:



### The 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/" \l "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 – we face a classic chicken-and-egg problem.

## Quicksort Java Source Code

The following Java source code (class [QuicksortSimple](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/quicksort/QuicksortSimple.java" \t "https://www.happycoders.eu/algorithms/quicksort/_blank) in the GitHub repository) always uses – for simplicity – the right element of a (sub)array as the pivot element.

As explained above, this is not a wise choice if the input data may be already sorted. However, this variant makes the code easier to understand for now.

import org.apache.commons.lang3.ArrayUtils;

public class QuicksortSimple {

    public void sort(int[] elements) {

        quicksort(elements, 0, elements.length - 1);

    }

    private void quicksort(int[] elements, int left, int right) {

        // End of recursion reached?

        if (left >= right) {

            return;

        }

        int pivotPos = partition(elements, left, right);

        quicksort(elements, left, pivotPos - 1);

        quicksort(elements, pivotPos + 1, right);

    }

    public int partition(int[] elements, int left, int right) {

        int pivot = elements[right];

        int i = left;

        int j = right - 1;

        while (i < j) {

            // Find the first element >= pivot

            while (elements[i] < pivot) {

                i++;

            }

            // Find the last element < pivot

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

                j--;

            }

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

            if (i < j) {

                ArrayUtils.swap(elements, i, j);

                i++;

                j--;

            }

        }

        // 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.

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

            i++;

        }

        // Move pivot element to its final position

        if (elements[i] != pivot) {

            ArrayUtils.swap(elements, i, right);

        }

        return i;

    }

}

Explanation of the source code:

The method sort() calls quicksort() and passes the array and the start and end positions.

The quicksort() method first calls the partition() method to partition the array. It then calls itself recursively – once for the subarray to the left of the pivot element and once for the subarray to the pivot element's right. The recursion ends when quicksort() is called for a subarray of length 1 or 0.

The partition() method partitions the array and returns the position of the pivot element. The variable i represents the left search pointer, the variable j the right search pointer. The individual steps of the partition() method are documented in the code – they correspond to the steps in the example from the ["Quicksort Partitioning"](https://www.happycoders.eu/algorithms/quicksort/" \l "Quicksort_Partitioning) section.

### Source Code for Alternative Pivot Strategies

If we do not want to use the rightmost element but another one as the pivot element, the algorithm must be extended. There are three variants:

#### Algorithm Variant 1

The easiest way is to swap the selected pivot element with the element on the right in advance. In this case, the rest of the source code can remain unchanged.

You can find a corresponding implementation in the class [QuicksortVariant1](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/quicksort/QuicksortVariant1.java" \t "https://www.happycoders.eu/algorithms/quicksort/_blank) in the GitHub repository. In this variant, the method findPivotAndMoveRight() is called before each partitioning. It selects the pivot element according to the chosen strategy and swaps it with the far-right element.

The enum PivotStrategy defines the following strategies:

* RANDOM: a random element is selected.
* LEFT: the left element is selected.
* RIGHT: the right element is selected (corresponds to the "QuicksortSimple" variant printed above).
* MIDDLE: the middle element is selected.
* MEDIAN3: the median of three elements of the array is selected as the pivot element.

#### Algorithm Variant 2

In this variant, we include the pivot element in the swap process and swap elements that are greater than or equal to the pivot element with elements that are smaller than the pivot element.

If we swap the pivot element itself, we must remember this change in position.

Therefore, the pivot element is located in the right section before the last step of partitioning and can be swapped with the right section's first element without further check.

You can find the source code of this variant in [QuicksortVariant2](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/quicksort/QuicksortVariant1.java" \t "https://www.happycoders.eu/algorithms/quicksort/_blank).

#### Algorithm Variant 3

In this variant, we leave the pivot element in place during partitioning. We achieve this by swapping only elements that are larger than the pivot element with elements that are smaller than the pivot element.

In the last step of the partitioning process, we have to check if the pivot element is located in the left or right section. If it is in the left section, we have to swap it with the last element of the left section; if it is in the right section, we have to swap it with the right section's first element.

You will find the source code of this variant in [QuicksortVariant3](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/quicksort/QuicksortVariant1.java" \t "https://www.happycoders.eu/algorithms/quicksort/_blank).

## Quicksort Time Complexity

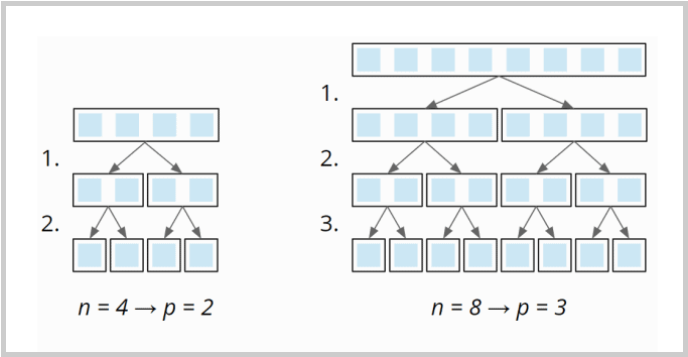
Click on the following link for an introduction to ["time complexity" and "O notation"](https://www.happycoders.eu/algorithms/big-o-notation-time-complexity/) (with examples and diagrams).

In the following sections, we refer to the number of elements to be sorted as n.

### Best-Case Time Complexity

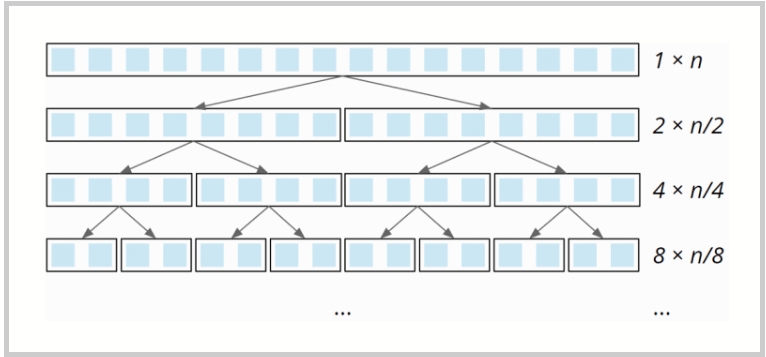
Quicksort achieves optimal performance if we always divide the arrays and subarrays into two partitions of equal size.

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 we have n elements times log2 n partitioning levels. Therefore:

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

### Average-Case Time Complexity

Unfortunately, the average time complexity cannot be derived without complicated mathematics, which would go beyond this article's scope. I refer to this [Wikipedia article](https://en.wikipedia.org/wiki/Quicksort" \l "Average-case_analysis" \t "https://www.happycoders.eu/algorithms/quicksort/_blank) instead.

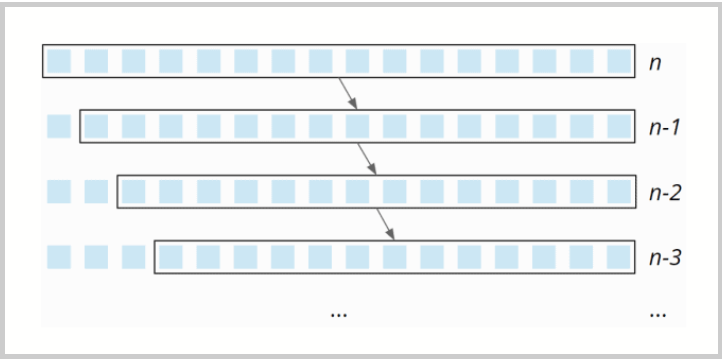
The article concludes that the average number of comparison operations is 1.39 n × log2 n – so we are still in a quasilinear time. Therefore:

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.:



The 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)

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.

## Java Quicksort Runtime

After all this theory, back to practice!

The [UltimateTest](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/demos/UltimateTest.java" \t "https://www.happycoders.eu/algorithms/quicksort/_blank) program allows us to measure the actual performance of Quicksort (and all other algorithms in this series of articles). The program operates as follows:

* It sorts arrays of sizes 1,024, 2,048, 4,096, etc. up to a maximum of 536,870,912 (= 229), but aborts if a single sorting process takes 20 seconds or longer.
* It applies the sorting algorithm to unsorted input data and input data sorted in ascending and descending order.
* It first runs two warmup phases to allow the HotSpot to optimize the code.
* The process is repeated until the process is killed.

### Runtime Measurement of the Quicksort Algorithm Variants

First of all, we have to decide which algorithm variant we want to put into the race to not let the test get out of hand. To do this, the [CompareQuicksorts](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/demos/comparisons/CompareQuicksorts.java) program combines all variants with all pivot strategies and sorts about 5.5 million elements with each combination 50 times.

Here is the result, sorted by runtime (file [Quicksort\_Pivot\_Strategies.log](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/results/2020-07-18/Quicksort_Pivot_Strategies.log))

| **Variant** | **Pivot Strategy** | **Median** |
| --- | --- | --- |
| QuicksortSimple | RIGHT | 458.5 ms |
| QuicksortVariant1 | RIGHT | 460.4 ms |
| QuicksortVariant1 | MIDDLE | 461.7 ms |
| QuicksortVariant3 | RIGHT | 472.4 ms |
| QuicksortVariant3 | MIDDLE | 473.5 ms |
| QuicksortVariant2 | RIGHT | 477.9 ms |
| QuicksortVariant2 | MIDDLE | 483.4 ms |
| QuicksortVariant1 | MEDIAN3 | 489.8 ms |
| QuicksortVariant3 | MEDIAN3 | 507.4 ms |
| QuicksortVariant2 | MEDIAN3 | 508.6 ms |
| QuicksortVariant1 | RANDOM | 516.1 ms |
| QuicksortVariant3 | RANDOM | 528.9 ms |
| QuicksortVariant2 | RANDOM | 534.2 ms |

You can read the following:

* The simple algorithm is the fastest.
* For all algorithm variants, the pivot strategy RIGHT is fastest, closely followed by MIDDLE, then MEDIAN3 with a slightly larger distance (the overhead is higher than the gain here). RANDOM is slowest (generating random numbers is expensive).
* For all pivot strategies, variant 1 is the fastest, variant 3 the second fastest, and variant 2 is the slowest.

### Runtime Measurements for Different Pivot Strategies and Array Sizes

Based on this result, I run the UltimateTest with algorithm variant 1 (pivot element is swapped with the right element in advance).

In the following sections, you will find the results for the various pivot strategies after 50 iterations (these are only excerpts; the complete test result can be found in [UltimateTest\_Quicksort.log](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/results/2020-07-18/UltimateTest_Quicksort.log))

#### Measurement Results for the "Right Element" Pivot Strategy

| **n** | **unsorted** | **ascending** | **descending** |
| --- | --- | --- | --- |
| 1,024 | 0.051 ms | 0.155 ms | 0.158 ms |
| 2,048 | 0.100 ms | 0.578 ms | 0.597 ms |
| 4,096 | 0.208 ms | 2.247 ms | 2.322 ms |
| 8,192 | 0.436 ms | 8.906 ms | 9.127 ms |
| 16,384 | 0.920 ms | StackOverflow | StackOverflow |
| 32,768 | 1.941 ms | StackOverflow | StackOverflow |
| ... | ... | ... | ... |
| 33,554,432 | 3,099.994 ms | StackOverflow | StackOverflow |
| 67,108,864 | 6,421.172 ms | StackOverflow | StackOverflow |
| 134,217,728 | 13,305.377 ms | StackOverflow | StackOverflow |
| 268,435,456 | 27,493.636 ms | StackOverflow | StackOverflow |

The data shows:

* For randomly distributed input data, the time required is slightly more than doubled if the array's size is doubled. This corresponds to the expected quasilinear runtime – O(n log n).
* For input data sorted in ascending or descending order, the time required quadruples when the input size is doubled, so we have quadratic time – O(n²).
* Sorting data in descending order takes only a little longer than sorting data in ascending order.
* With only 8,192 elements, sorting presorted input data takes 23 times as long as sorting unsorted data.
* With more than 8,192 elements, the dreaded StackOverflowException occurs with presorted input data.

#### Measurement Results for the "Middle Element" Pivot Strategy

| **n** | **unsorted** | **ascending** | **descending** |
| --- | --- | --- | --- |
| ... | ... | ... | ... |
| 16,777,216 | 1,508 ms | 191.3 ms | 227.0 ms |
| 33,554,432 | 3,127 ms | 409.5 ms | 464.7 ms |
| 67,108,864 | 6,486 ms | 806.4 ms | 942.9 ms |
| 134,217,728 | 13,409 ms | 1,727.2 ms | 1,945.8 ms |
| 268,435,456 | 27,740 ms | 3,405.2 ms | 3,959.2 ms |

The data shows:

* For both unsorted and sorted input data, doubling the array size requires slightly more than twice the time. This corresponds to the expected quasilinear runtime – O(n log n).
* The algorithm is significantly faster for presorted input data than for random data – both for ascending and descending sorted data.
* The performance loss due to the pilot element's initial swapping with the right element is less than 0.9% in all tests with unsorted input data.

#### Measurement Results for the "Median of Three Elements" Pivot Strategy

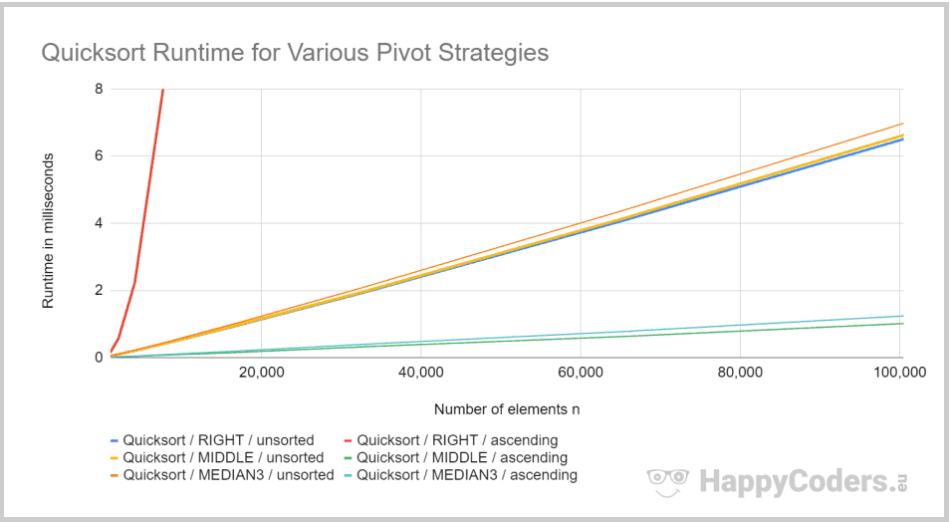
| **n** | **unsorted** | **ascending** | **descending** |
| --- | --- | --- | --- |
| ... | ... | ... | ... |
| 16,777,216 | 1,589 ms | 222.6 ms | 249.0 ms |
| 33,554,432 | 3,291 ms | 473.2 ms | 514.4 ms |
| 67,108,864 | 6,807 ms | 934.6 ms | 1,039.1 ms |
| 134,217,728 | 14,066 ms | 1,980.5 ms | 2,142.8 ms |
| 268,435,456 | 29,041 ms | 3,907.6 ms | 4,349.2 ms |

The data shows:

* Here too, we have quasilinear time in all cases – O(n log n).
* As in the algorithm variants comparison, the pivot strategy "median of three elements" is somewhat slower than the "middle element" strategy.

#### Overview of All Measurement Results

Here you can find the measurement results again as a diagram (I have omitted input data sorted in descending order for clarity):



Once again, you can see that the "right element" strategy leads to quadratic effort for ascending sorted data (red line) and is fastest for unsorted data (blue line). The second fastest (with a minimal gap) is the "middle element" pivot strategy (yellow line).

## Quicksort Optimized: 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.

### Quicksort/Insertion Sort Source Code

The source code changes compared to the standard quicksort are very straightforward and are limited to the quicksort() method. Here is the method from the standard algorithm once again:

private void quicksort(int[] elements, int left, int right) {

  // End of recursion reached?

  if (left >= right) {

    return;

  }

  int pivotPos = partition(elements, left, right);

  quicksort(elements, left, pivotPos - 1);

  quicksort(elements, pivotPos + 1, right);

}

And here is the optimized version. The variables insertionSort and partitioningAlgorithm are instances of an insertion sort and a quicksort algorithm. Only the code block commented with "Threshold for insertion sort reached?" has been added in the middle of the method:

private void quicksort(int[] elements, int left, int right) {

  // End of recursion reached?

  if (left >= right) {

    return;

  }

  // Threshold for insertion sort reached?

  if (right - left < threshold) {

    insertionSort.sort(elements, left, right + 1);

    return;

  }

  int pivotPos = partitioningAlgorithm.partition(elements, left, right);

  quicksort(elements, left, pivotPos - 1);

  quicksort(elements, pivotPos + 1, right);

}

You can find the complete source code in the [QuicksortImproved](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/quicksort/QuicksortImproved.java" \t "https://www.happycoders.eu/algorithms/quicksort/_blank) class in the GitHub repository. As constructor parameters, the threshold for switching to Insertion Sort, threshold, is passed and an instance of the Quicksort variant to be used.

### Quicksort/Insertion Sort Performance

The [CompareImprovedQuickSort](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/demos/comparisons/CompareImprovedQuicksort.java" \t "https://www.happycoders.eu/algorithms/quicksort/_blank) program measures the time needed to sort about 5.5 million elements at different thresholds for switching to Insertion Sort.

Since the optimized Quicksort only partitions arrays above a certain size, the influence of the pivot strategy and algorithm variant could play a different role than before. To take this into account, the program tests the limits for all three algorithm variants and the pivot strategies "middle" and "median of three elements".

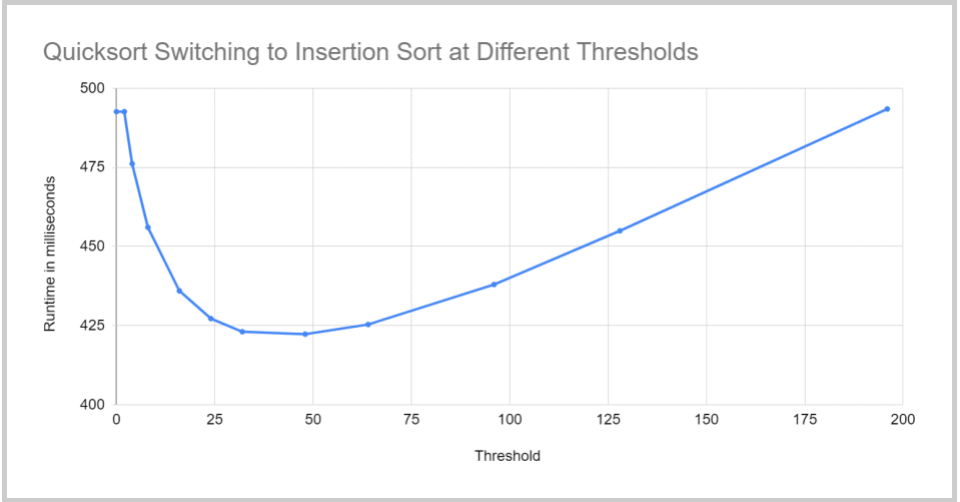
You will find the complete measurement results in [CompareImprovedQuicksort.log](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/results/2020-07-18/CompareImprovedQuicksort.log" \t "https://www.happycoders.eu/algorithms/quicksort/_blank).

As in the previous tests, algorithm variant 1 and pivot strategy "middle element" perform best.

Here are the measured runtimes for the chosen combination and various thresholds for switching to Insertion Sort:

| **Threshold** | **Runtime** |
| --- | --- |
| 0 (= regular Quicksort) | 492.6 ms |
| 2 | 492.6 ms |
| 4 | 476.1 ms |
| 8 | 456.1 ms |
| 16 | 436.0 ms |
| 24 | 427.2 ms |
| 32 | 423.1 ms |
| 48 | 422.3 ms |
| 64 | 425.3 ms |
| 96 | 438.0 ms |
| 128 | 454.9 ms |
| 196 | 493.4 ms |

Here are the measurements in graphical representation:



Result:

By switching to Insertion Sort for (sub)arrays containing 48 or fewer elements, we can reduce Quicksort's runtime for 5.5 million elements to about 85% of the original value.

You will see how the optimized Quicksort algorithm performs with other array sizes in the section ["Comparing all Quicksort optimizations"](https://www.happycoders.eu/algorithms/quicksort/" \l "Comparing_All_Quicksort_Optimizations).

## Dual-Pivot Quicksort

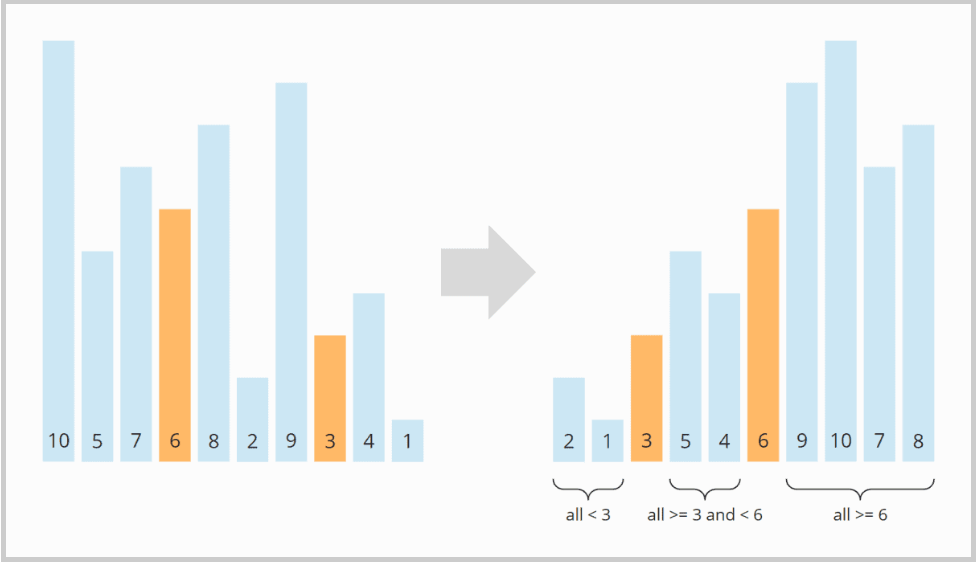
Quicksort can be further optimized by using two pivot elements instead of one. When partitioning, the elements are then divided into:

* 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.

The following diagram shows an example of partitioning with two pivot elements at the "thirds" positions:



Dual-Pivot Quicksort (with additional optimizations) is used in the JDK by the method Arrays.sort().

### Dual-Pivot Quicksort Source Code

Compared to the regular algorithm, the quicksort() method calls itself recursively not for two but three partitions:

private void quicksort(int[] elements, int left, int right) {

  // End of recursion reached?

  if (left >= right) {

    return;

  }

  int[] pivotPos = partition(elements, left, right);

  int p0 = pivotPos[0];

  int p1 = pivotPos[1];

  quicksort(elements, left, p0 - 1);

  quicksort(elements, p0 + 1, p1 - 1);

  quicksort(elements, p1 + 1, right);

}

The partition() method first calls findPivotsAndMoveToLeftRight(), which selects the pivot elements based on the chosen pivot strategy and swaps them with the left and right elements (similar to swapping the pivot element with the right element in the regular quicksort).

Then again, two search pointers run over the array from left and right and compare and swap the elements to be eventually divided into three partitions. How exactly they do this can be read reasonably well from the source code.

int[] partition(int[] elements, int left, int right) {

  findPivotsAndMoveToLeftRight(elements, left, right);

  int leftPivot = elements[left];

  int rightPivot = elements[right];

  int leftPartitionEnd = left + 1;

  int leftIndex = left + 1;

  int rightIndex = right - 1;

  while (leftIndex <= rightIndex) {

    // elements < left pivot element?

    if (elements[leftIndex] < leftPivot) {

      ArrayUtils.swap(elements, leftIndex, leftPartitionEnd);

      leftPartitionEnd++;

    }

    // elements >= right pivot element?

    else if (elements[leftIndex] >= rightPivot) {

      while (elements[rightIndex] > rightPivot && leftIndex < rightIndex) {

        rightIndex--;

      }

      ArrayUtils.swap(elements, leftIndex, rightIndex);

      rightIndex--;

      if (elements[leftIndex] < leftPivot) {

        ArrayUtils.swap(elements, leftIndex, leftPartitionEnd);

        leftPartitionEnd++;

      }

    }

    leftIndex++;

  }

  leftPartitionEnd--;

  rightIndex++;

  // move pivots to their final positions

  ArrayUtils.swap(elements, left, leftPartitionEnd);

  ArrayUtils.swap(elements, right, rightIndex);

  return new int[]{leftPartitionEnd, rightIndex};

}

The findPivotsAndMoveToLeftRight() method operates as follows:

With the LEFT\_RIGHT pivot strategy, it checks whether the leftmost element is smaller than the rightmost element. If not, both are swapped.

The THIRDS strategy first extracts the elements at the positions "one third" (variable first) and "two thirds" (variable second). This is followed by a series of if queries, which ultimately place the larger of the two elements to the far right and the smaller of the two elements to the far left.

(The code is so bloated because it has to handle two exceptional cases: In tiny partitions, the first pivot element could be the leftmost element, and the second pivot element could be the rightmost element.)

private void findPivotsAndMoveToLeftRight(int[] elements,

                                          int left, int right) {

  switch (pivotStrategy) {

    case LEFT\_RIGHT -> {

      if (elements[left] > elements[right]) {

        ArrayUtils.swap(elements, left, right);

      }

    }

    case THIRDS -> {

      int len = right - left + 1;

      int firstPos = left + (len - 1) / 3;

      int secondPos = right - (len - 2) / 3;

      int first = elements[firstPos];

      int second = elements[secondPos];

      if (first > second) {

        if (secondPos == right) {

          if (firstPos == left) {

            ArrayUtils.swap(elements, left, right);

          } else {

            // 3-way swap

            elements[right] = first;

            elements[firstPos] = elements[left];

            elements[left] = second;

          }

        } else if (firstPos == left) {

          // 3-way swap

          elements[left] = second;

          elements[secondPos] = elements[right];

          elements[right] = first;

        } else {

          ArrayUtils.swap(elements, firstPos, right);

          ArrayUtils.swap(elements, secondPos, left);

        }

      } else {

        if (secondPos != right) {

          ArrayUtils.swap(elements, secondPos, right);

        }

        if (firstPos != left) {

          ArrayUtils.swap(elements, firstPos, left);

        }

      }

    }

    default -> throw new IllegalStateException("Unexpected value: " + pivotStrategy);

  }

}

You can find the complete source code in the file [DualPivotQuicksort](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/quicksort/DualPivotQuicksort.java" \t "https://www.happycoders.eu/algorithms/quicksort/_blank).

### Dual-Pivot Quicksort Performance

If and to what extent Dual-Pivot Quicksort improves performance, you will find out in the section ["Comparing all Quicksort optimizations"](https://www.happycoders.eu/algorithms/quicksort/" \l "Comparing_All_Quicksort_Optimizations).

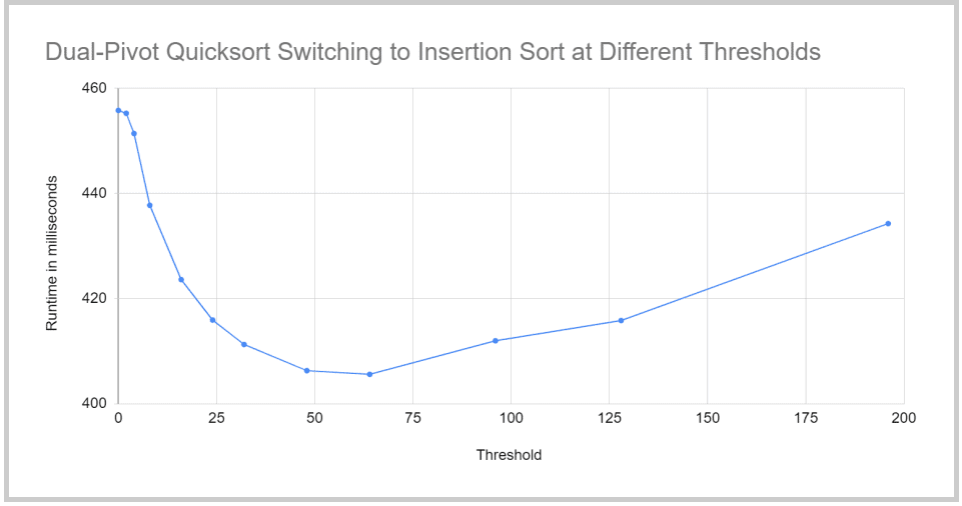
### Dual-Pivot Quicksort Combined With Insertion Sort

Just like the regular Quicksort, Dual-Pivot Quicksort can be combined with Insertion Sort. The source code changes are the same as for the regular quicksort (see section ["Quicksort/Insertion Sort Source Code"](https://www.happycoders.eu/algorithms/quicksort/" \l "QuicksortInsertion_Sort_Source_Code)). Therefore I will not go into the details here.

You can find the source code in [DualPivotQuicksortImproved](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/quicksort/DualPivotQuicksortImproved.java" \t "https://www.happycoders.eu/algorithms/quicksort/_blank).

The [CompareImprovedDualPivotQuicksort](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/demos/comparisons/CompareImprovedDualPivotQuicksort.java" \t "https://www.happycoders.eu/algorithms/quicksort/_blank) program tests the algorithm for different thresholds for switching to Insertion Sort.

You can find the results in [CompareImprovedDualPivotQuicksort.log](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/results/2020-07-18/CompareImprovedDualPivotQuicksort.log" \t "https://www.happycoders.eu/algorithms/quicksort/_blank). Here they are as a diagram:



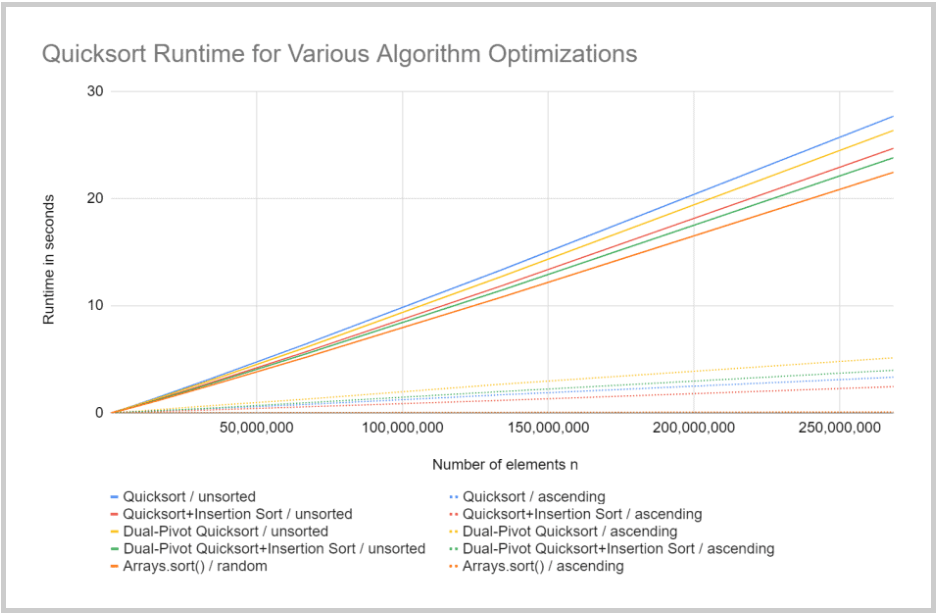
Therefore, for Dual-Pivot Quicksort, it is worthwhile to sort (sub)arrays with 64 elements or less with Insertion Sort.

## Comparing All Quicksort Optimizations

Finally, let's compare the performance Finally, I compare the following algorithms' performance with the UltimateTest mentioned in section ["Java Quicksort Runtime"](https://www.happycoders.eu/algorithms/quicksort/" \l "Java_Quicksort_Runtime):

* Regular quicksort with "middle element" pivot strategy,
* Quicksort combined with Insertion Sort and a threshold of 48,
* Dual-Pivot Quicksort with "elements in the positions one third and two thirds" pivot strategy,
* Dual-Pivot Quicksort combined with Insertion Sort and a threshold of 64,
* The JDK's Arrays.sort() (the JDK developers have optimized their Dual-Pivot Quicksort algorithm to such an extent that it is worth switching to Insertion Sort only with 44 elements).

You will find the result in [UltimateTest\_Quicksort\_Optimized.log](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/results/2020-07-18/UltimateTest_Quicksort_Optimized.log" \t "https://www.happycoders.eu/algorithms/quicksort/_blank) – and in the following diagram:



First of all, the quasilinear complexity of all variants can be seen very clearly.

Dual-Pivot Quicksort's performance is visibly better than that of regular Quicksort – about 5% for a quarter of a billion elements. The combinations with Insertion Sort bring at least 10% performance gain.

My Quicksort implementations do not quite come close to that of the JDK – about 6% are still missing. The JDK developers have highly optimized their code over the years. If you're interested in how exactly, you can check out the [source code on GitHub](https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/DualPivotQuicksort.java" \t "https://www.happycoders.eu/algorithms/quicksort/_blank).

It is also good to see that all variants sort presorted data much faster than unsorted data – and data sorted ascending a little quicker than data sorted descending. Arrays.sort() is also optimized for presorted data, so that the corresponding line in the diagram is only slightly above zero (172.7 ms for a quarter of a billion elements).

## Further Characteristics of Quicksort

This chapter discusses Quicksort's space complexity, its stability, and its parallelizability.

### 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) (see section ["Time complexity"](https://www.happycoders.eu/algorithms/quicksort/" \l "Quicksort_Time_Complexity)).

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

However, the algorithm can be optimized by [tail-end recursion](https://en.wikipedia.org/wiki/Tail_call" \t "https://www.happycoders.eu/algorithms/quicksort/_blank) so that only the smaller partition is processed by recursion, and the larger partition is processed by iteration.

Since the smaller subpartition is at most half the size of the original partition (otherwise it would not be the smaller but the larger subpartition), tail-end recursion results in a maximum recursion depth of log2 n even in the worst case.

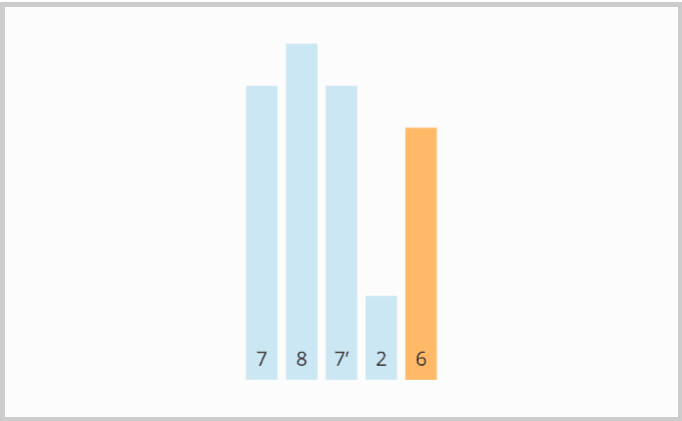
The additional memory requirement per recursion level is constant. Therefore:

Quicksort's space complexity is in the best and average case and – when using tail-end recursion also in the worst case – O(log n)

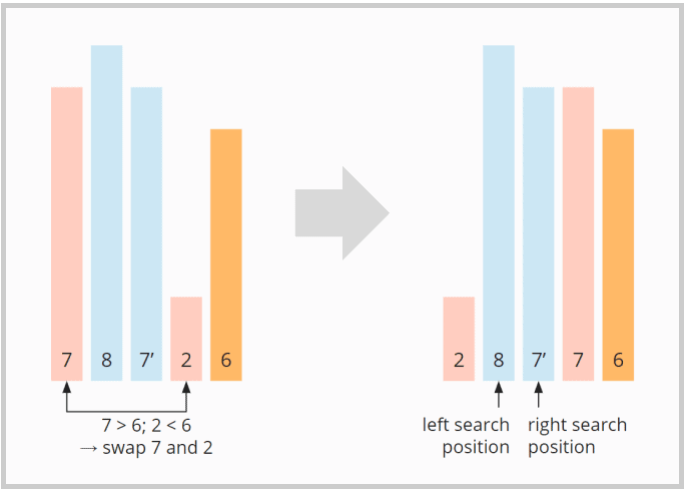
### Stability of Quicksort

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.

### Parallelizability of Quicksort

There are different ways to parallelize Quicksort.

Firstly, several partitions can be further partitioned in parallel. With this variant, however, the first partitioning level cannot be parallelized at all; in the second level, only two cores can be used; in the third, only four; and so on.

Several other – more sophisticated – variants exist; you can find a summary in this [article on parallel Quicksort](https://iq.opengenus.org/parallel-quicksort/" \t "https://www.happycoders.eu/algorithms/quicksort/_blank).

## Quicksort vs. Merge Sort

You can find a comparison of Quicksort and Merge Sort in the [article about Merge Sort](https://www.happycoders.eu/algorithms/merge-sort/" \l "Merge_Sort_vs_Quicksort).

## Conclusion

Quicksort is an efficient, unstable sorting algorithm with time complexity of O(n log n) in the best and average case and O(n²) in the worst case.

For small n, Quicksort is slower than Insertion Sort and is therefore usually combined with Insertion Sort in practice.

The Arrays.sort() method in the JDK uses a dual-pivot quicksort implementation that sorts (sub)arrays with less than 44 elements with Insertion Sort.

You can find more sorting algorithms in the [overview of all sorting algorithms and their characteristics](https://www.happycoders.eu/algorithms/sorting-algorithms/" \l "Comparison_of_the_Most_Important_Sorting_Algorithms" \t "https://www.happycoders.eu/algorithms/quicksort/_blank) in the first part of the article series.

If you liked the article, feel free to share it using one of the share buttons at the end. Would you like to be informed by e-mail when I publish a new article? Then [click here](https://www.happycoders.eu/algorithms/quicksort/) to subscribe to the HappyCoders newsletter.