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

# Heapsort – Algorithm, Source Code, Time Complexity

With Heapsort, every Java developer first thinks of the Java heap. This article will show you that Heapsort is something completely different – and how Heapsort works precisely.

You'll find out in detail:

* What is a Heap?
* How does the Heapsort algorithm work?
* What does the Heapsort source code look like?
* How to determine Heapsort's time complexity?
* What is Bottom-up Heapsort, and what are its advantages?
* How does Heapsort compare to Quicksort and Merge Sort?

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

**[1 What is a Heap?](https://www.happycoders.eu/algorithms/heapsort/" \l "What_is_a_Heap)**

**[2 Heapsort Algorithm](https://www.happycoders.eu/algorithms/heapsort/" \l "Heapsort_Algorithm)**

[2.1 Phase 1: Creating the Heap](https://www.happycoders.eu/algorithms/heapsort/" \l "Phase_1_Creating_the_Heap)

[2.2 Phase 2: Sorting the Array](https://www.happycoders.eu/algorithms/heapsort/" \l "Phase_2_Sorting_the_Array)

[2.3 Repeating the Steps](https://www.happycoders.eu/algorithms/heapsort/" \l "Repeating_the_Steps)

**[3 Heapsort Java Code Example](https://www.happycoders.eu/algorithms/heapsort/" \l "Heapsort_Java_Code_Example)**

**[4 Heapsort Time Complexity](https://www.happycoders.eu/algorithms/heapsort/" \l "Heapsort_Time_Complexity)**

[4.1 Time Complexity of the heapify() Method](https://www.happycoders.eu/algorithms/heapsort/" \l "Time_Complexity_of_the_heapify_Method)

[4.2 Time Complexity of the buildHeap() Method](https://www.happycoders.eu/algorithms/heapsort/" \l "Time_Complexity_of_the_buildHeap_Method)

[4.3 Total Time Complexity of Heapsort](https://www.happycoders.eu/algorithms/heapsort/" \l "Total_Time_Complexity_of_Heapsort)

[4.4 Time Complexity for Building the Heap – In-Depth Analysis](https://www.happycoders.eu/algorithms/heapsort/" \l "Time_Complexity_for_Building_the_Heap_-_In-Depth_Analysis)

[4.5 Runtime of the Java Heapsort Example](https://www.happycoders.eu/algorithms/heapsort/" \l "Runtime_of_the_Java_Heapsort_Example)

[4.6 Why Is Heapsort Faster for Presorted Input Data?](https://www.happycoders.eu/algorithms/heapsort/" \l "Why_Is_Heapsort_Faster_for_Presorted_Input_Data)

**[5 Bottom-Up Heapsort](https://www.happycoders.eu/algorithms/heapsort/" \l "Bottom-Up_Heapsort)**

[5.1 Bottom-Up Heapsort Algorithm](https://www.happycoders.eu/algorithms/heapsort/" \l "Bottom-Up_Heapsort_Algorithm)

[5.2 Bottom-Up Heapsort Example](https://www.happycoders.eu/algorithms/heapsort/" \l "Bottom-Up_Heapsort_Example)

[5.3 Bottom-Up Heapsort Source Code](https://www.happycoders.eu/algorithms/heapsort/" \l "Bottom-Up_Heapsort_Source_Code)

[5.4 Bottom-Up Heapsort Performance](https://www.happycoders.eu/algorithms/heapsort/" \l "Bottom-Up_Heapsort_Performance)

[5.5 Bottom-Up Heapsort With Expensive Comparison Operations](https://www.happycoders.eu/algorithms/heapsort/" \l "Bottom-Up_Heapsort_With_Expensive_Comparison_Operations)

**[6 Further Characteristics of Heapsort](https://www.happycoders.eu/algorithms/heapsort/" \l "Further_Characteristics_of_Heapsort)**

[6.1 Space Complexity of Heapsort](https://www.happycoders.eu/algorithms/heapsort/" \l "Space_Complexity_of_Heapsort)

[6.2 Stability of Heapsort](https://www.happycoders.eu/algorithms/heapsort/" \l "Stability_of_Heapsort)

[6.3 Heapsort Parallelizability](https://www.happycoders.eu/algorithms/heapsort/" \l "Heapsort_Parallelizability)

**[7 Comparing Heapsort With Other Efficient Sorting Algorithms](https://www.happycoders.eu/algorithms/heapsort/" \l "Comparing_Heapsort_With_Other_Efficient_Sorting_Algorithms)**

[7.1 Heapsort vs. Quicksort](https://www.happycoders.eu/algorithms/heapsort/" \l "Heapsort_vs_Quicksort)

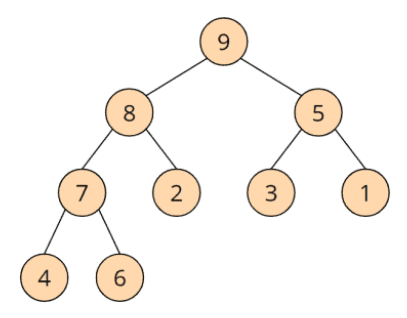
[7.2 Heapsort vs. Merge Sort](https://www.happycoders.eu/algorithms/heapsort/" \l "Heapsort_vs_Merge_Sort)

**[8 Summary](https://www.happycoders.eu/algorithms/heapsort/" \l "Summary)**

## What is a Heap?

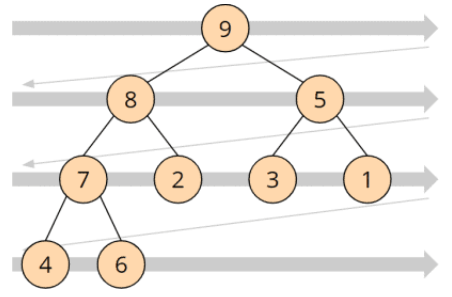
A "heap" is a [binary tree](https://www.happycoders.eu/algorithms/binary-tree-java/) in which each node is either greater than or equal to its children ("max heap") – or less than or equal to its children ("min heap").

Here is a simple example of a "max heap":



The 9 is greater than the 8 and the 5; the 8 is greater than the 7 and the 2; etc.

A heap is projected onto an array by transferring its elements line by line from top left to bottom right into the array:



The heap shown above looks like this as an array:



In a "max heap", the largest element is always at the top – in the array form, it is, therefore, on the far left. The following section explains how to use this characteristic for sorting.

## Heapsort Algorithm

The heapsort algorithm consists of two phases: In the first phase, the array to be sorted is converted into a max heap. And in the second phase, the largest element (i.e., the one at the tree root) is removed, and a new max heap is created from the remaining elements.

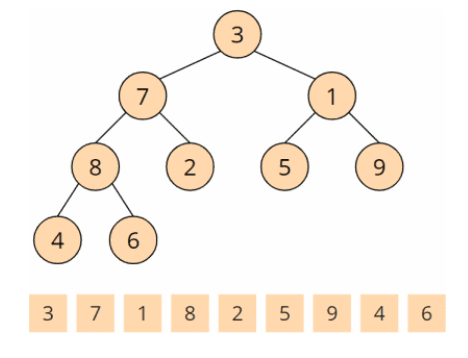
The following sections explain the two phases in detail using an example:

### Phase 1: Creating the Heap

The array to be sorted must first be converted into a heap. For this purpose, no new data structure is created, but the numbers are rearranged within the array so that the heap structure described above is created.

In the following example, I explain how exactly this is done using the number sequence known from the previous parts of the article series: [3, 7, 1, 8, 2, 5, 9, 4, 6].

We "project" these numbers onto a binary tree, as described above. The binary tree is not a separate data structure, but only a thought construct – in the computer's memory, the elements are located exclusively in the array.

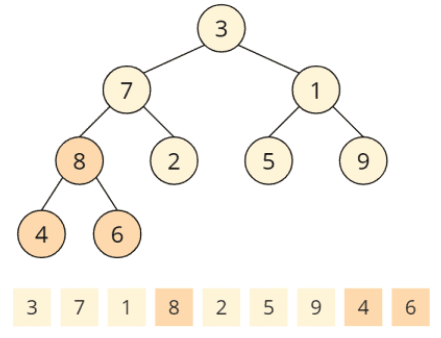


This tree does not yet represent a max heap. The definition of a max heap is that parents are always greater than or equal to their children.

To create a max heap, we now visit all parent nodes – backward from the last one to the first – and make sure that the heap condition for the respective node and the one below is fulfilled. We do this using the so-called heapify() method.

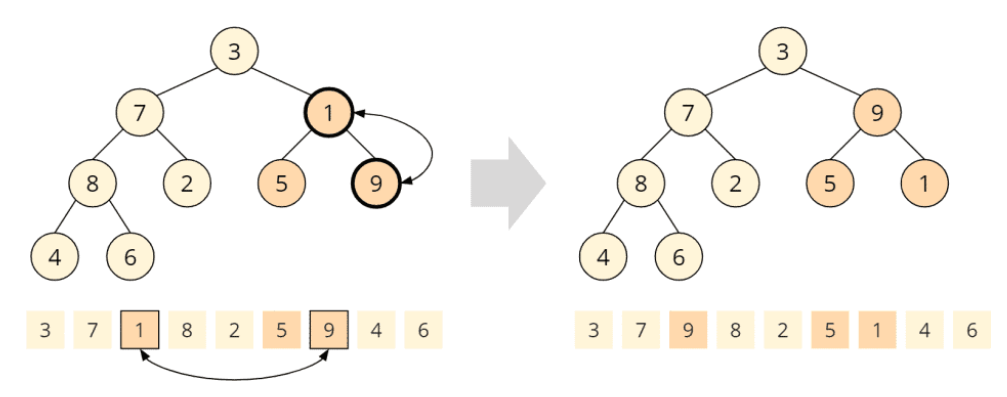
#### Invocation No. 1 of the Heapify Method

The heapify() method is called first for the last parent node. Parent nodes are 3, 7, 1, and 8. The last parent node is 8. The heapify() function checks if the children are smaller than the parent node. 4 and 6 are smaller than 8, so at this parent node, the heap condition is fulfilled, and the heapify() function is finished.



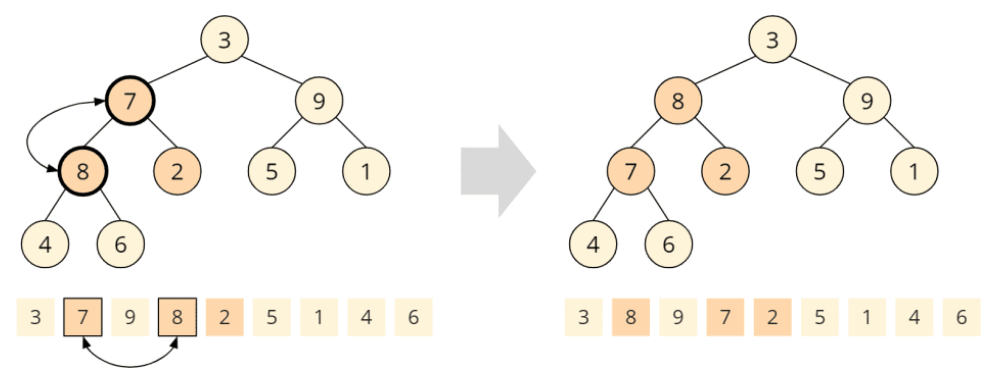
#### Invocation No. 2 of the Heapify Method

Second, heapify() is called for the penultimate node: the 1. Its children 5 and 9 are both greater than 1, so the heap condition is violated. To restore the heap condition, we now swap the larger child with the parent node, i.e., the 9 with the 1. The heapify() method is now finished again.

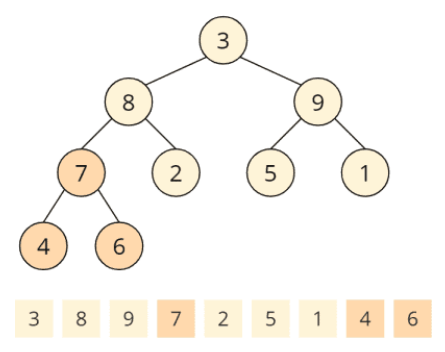


#### Invocation No. 3 of the Heapify Method

Now heapify() is called on the 7. Child nodes are 8 and 2; only the 8 is larger than the parent node. So we exchange the 7 with the 8:

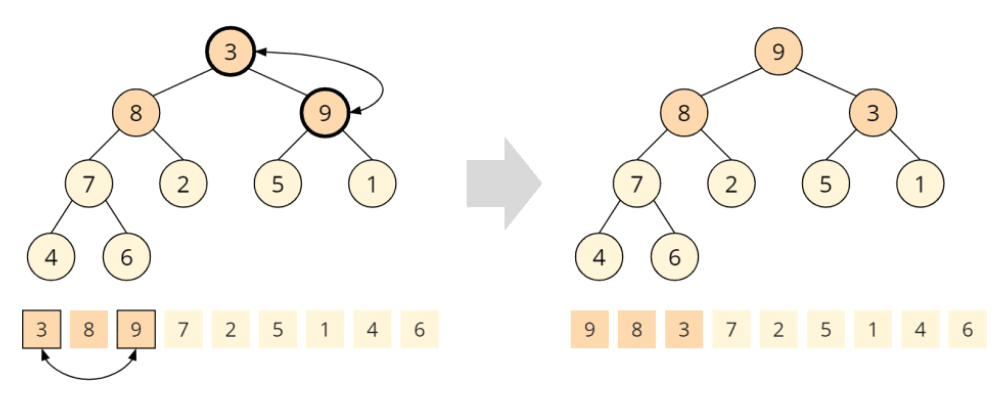


Since the child node we just swapped has two children itself, the heapify() method must now check if the heap condition for this child node is still valid. In this case, the 7 is greater than 4 and 6; the heap condition is fulfilled, and the heapify() function is finished.

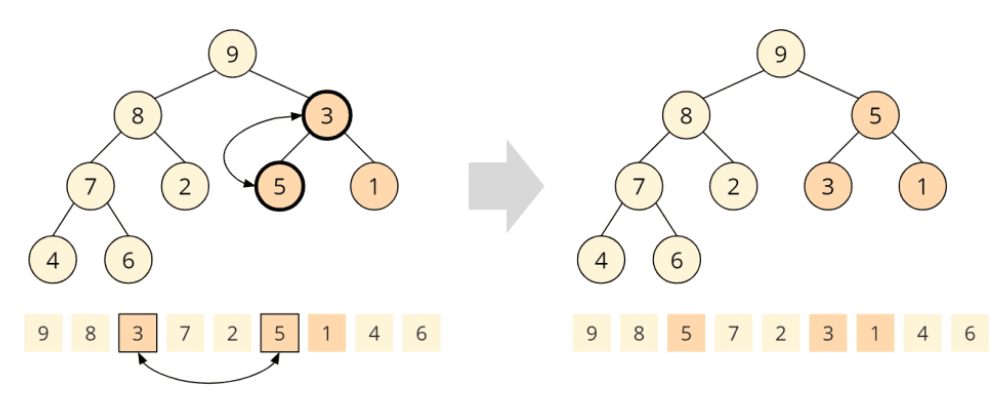


#### Invocation No. 4 of the Heapify Method

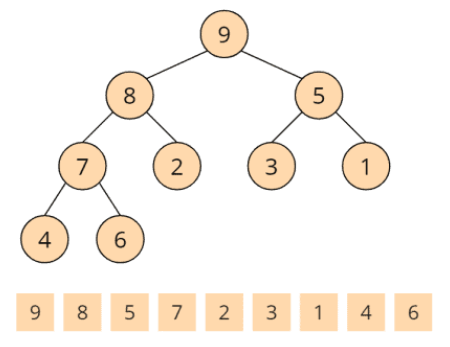
Now we have arrived at the root node with element 3. Both child nodes, 8 and 9 are larger, while 9 is the largest child and is, therefore, swapped with the parent node:



Again, the swapped child node has children itself, so we need to check the heap condition on this child node. The 5 is greater than the 3, i.e., the heap condition is not fulfilled. It must be restored by swapping the 5 and the 3:



The fourth and last call of the heapify() function has finished. A max heap has been created:



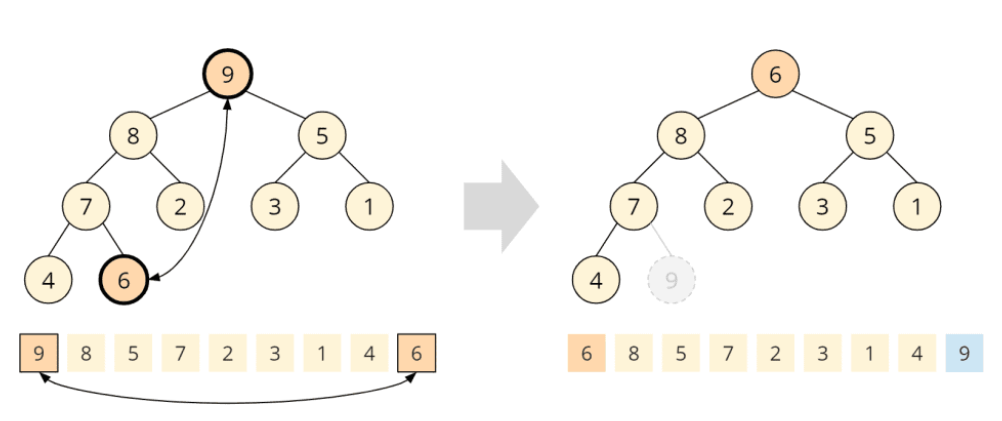
Which brings us to phase two of the heapsort algorithm.

### Phase 2: Sorting the Array

In phase 2, we take advantage of the fact that the largest element of the max heap is always at its root (in the array: on the far left).

#### Phase 2, Step 1: Swapping the Root and Last Elements

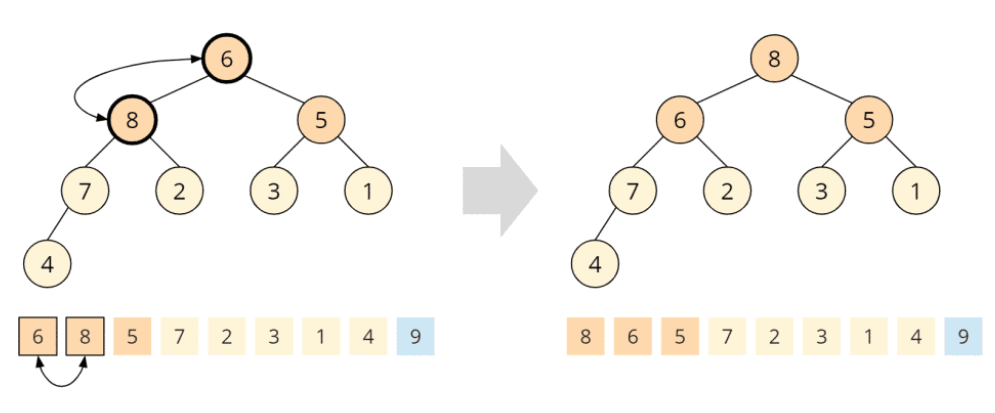
The root element (the 9) is now swapped with the last element (the 6) so that the 9 is at its final position at the end of the array (marked blue in the array). We also remove this element from the tree (displayed in grey):



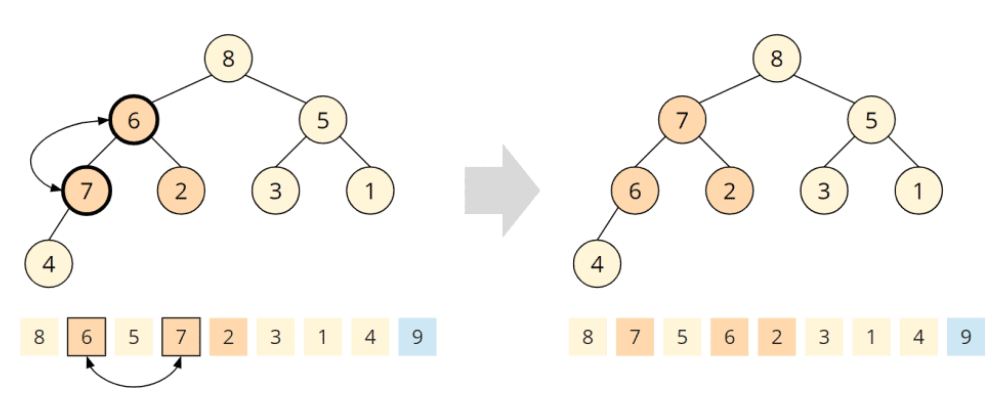
After we've placed the 6 at the root of the tree, it is no longer a max heap. Therefore, in the next step, we will "repair" the heap.

#### Phase 2, Step 2: Restoring the Heap Condition

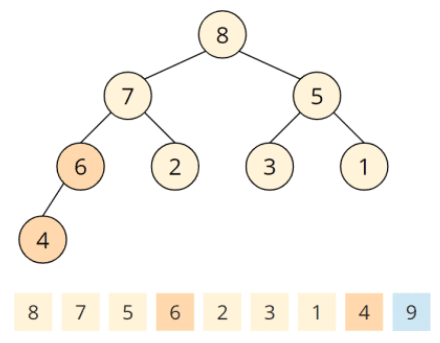
To restore the heap condition, we call the heapify() method known from phase 1 on the root node. This means we compare the 6 with its children, 8 and 5; the 8 is bigger, so we swap it with the 6:



The swapped child node has, in turn, two children, the 7 and the 2. The 7 is larger than the 6, and we swap these two elements as well:

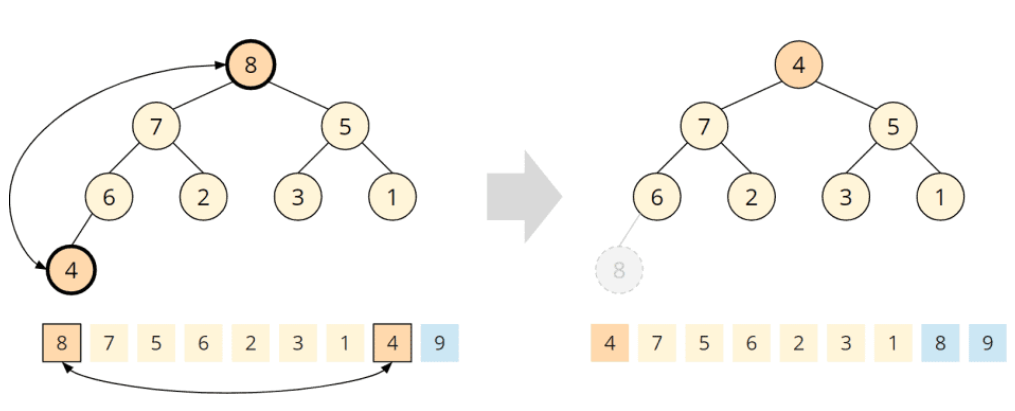


The exchanged child node also has a child, the 4. The 6 is greater than the 4, so the heap condition is fulfilled at this node. The heapify() function is finished, and we have a max heap again:



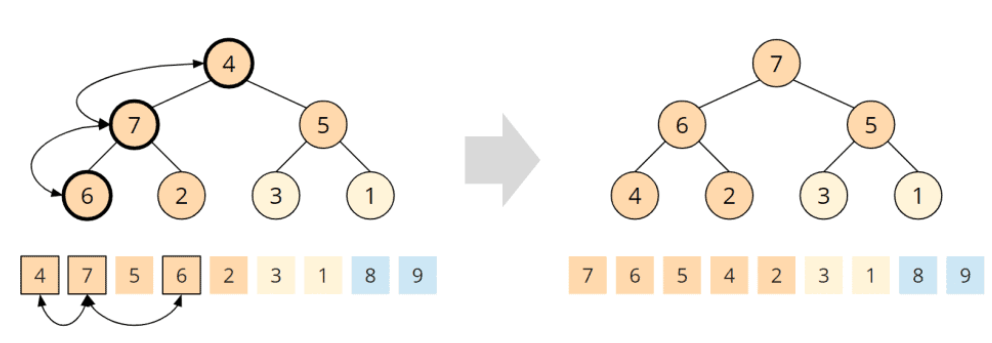
### Repeating the Steps

The largest number of the remaining array, 8, is now in the first position. We swap it with the last element of the tree. Since we have shortened the tree by one element, the last element of the tree is on the second last field of the array:

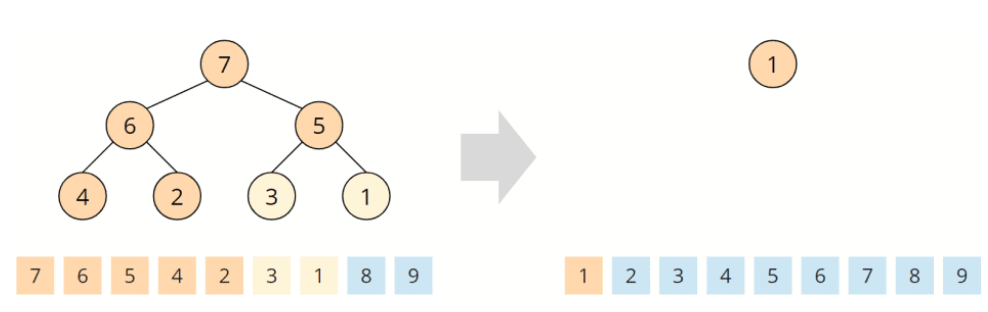


Now, the last two fields of the array are sorted.

At the root, the heap condition is violated again. We repair the tree by calling heapify() on the root element (the following picture shows all heapify steps at once).



We repeat the process until there is only one element left in the tree:



This element is the smallest and remains at the beginning of the array. The algorithm is finished, the array is sorted:



## Heapsort Java Code Example

In this section, you'll find the source code of Heapsort.

The sort() method first calls buildHeap() to initially build the heap.

In the following loop, the variable swapToPos iterates backward from the end of the array to its second field. In the loop body, the first element is swapped with the one at the swapToPos position, and then the heapify() method is called on the subarray up to (exclusive) the swapToPos position:

public class HeapSort {

  public void sort(int[] elements) {

    buildHeap(elements);

    for (int swapToPos = elements.length - 1; swapToPos > 0; swapToPos--) {

      // Move root to end

      ArrayUtils.swap(elements, 0, swapToPos);

      // Fix remaining heap

      heapify(elements, swapToPos, 0);

    }

  }

  [...]

The buildHeap() method calls heapify() for each parent node, starting with the last one, and passes to this method the array, the length of the subarray representing the heap, and the position of the parent node where heapify() should start:

void buildHeap(int[] elements) {

  // "Find" the last parent node

  int lastParentNode = elements.length / 2 - 1;

  // Now heapify it from here on backwards

  for (int i = lastParentNode; i >= 0; i--) {

    heapify(elements, elements.length, i);

  }

}

The heapify() method checks whether a child node is larger than the parent node. If this is the case, the parent element is swapped with the larger child element, and the process is repeated on the child node.

(You could also work with recursion here, but this would have a negative effect on the space complexity)

void heapify(int[] heap, int length, int parentPos) {

  while (true) {

    int leftChildPos = parentPos \* 2 + 1;

    int rightChildPos = parentPos \* 2 + 2;

    // Find the largest element

    int largestPos = parentPos;

    if (leftChildPos < length && heap[leftChildPos] > heap[largestPos]) {

      largestPos = leftChildPos;

    }

    if (rightChildPos < length && heap[rightChildPos] > heap[largestPos]) {

      largestPos = rightChildPos;

    }

    // largestPos is now either parentPos, leftChildPos or rightChildPos.

    // If it's the parent, we're done

    if (largestPos == parentPos) {

      break;

    }

    // If it's not the parent, then switch!

    ArrayUtils.swap(heap, parentPos, largestPos);

    // ... and fix again starting at the child we moved the parent to

    parentPos = largestPos;

  }

}

You can find the source code in the [HeapSort class in the GitHub repository](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/heapsort/Heapsort.java" \t "https://www.happycoders.eu/algorithms/heapsort/_blank). It is slightly different from the class printed here: The class in the repository implements the SortAlgorithm interface to be interchangeable within the test framework.

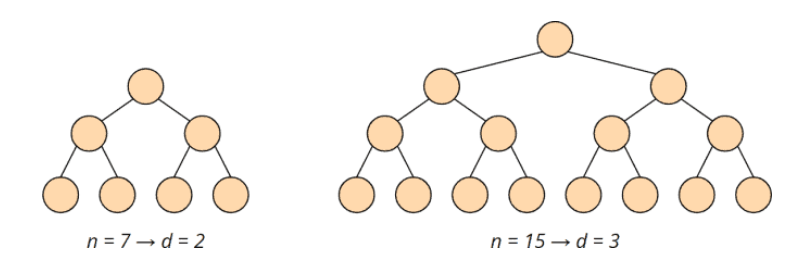
## Heapsort Time Complexity

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).

### Time Complexity of the heapify() Method

Let's start with the heapify() method since we also need it for the heap's initial build.

In the heapify() function, we walk through the tree from top to bottom. The height of a binary tree (the root not being counted) of size n is log2 n at most, i.e., if the number of elements doubles, the tree becomes only one level deeper:

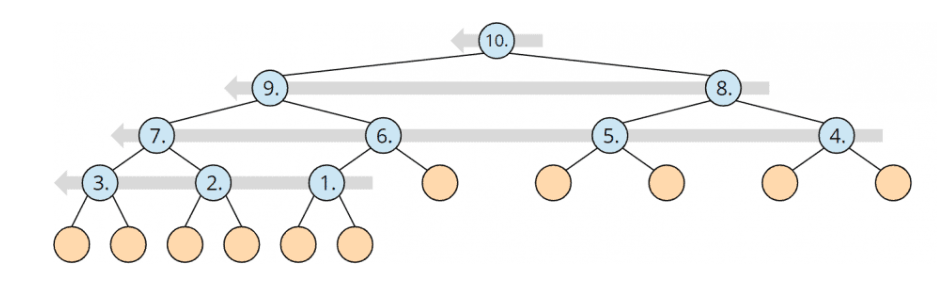


The complexity for the heapify() function is accordingly O(log n).

### Time Complexity of the buildHeap() Method

To initially build the heap, the heapify() method is called for each parent node – backward, starting with the last node and ending at the tree root.

A heap of size n has n/2 (rounded down) parent nodes:



Since the complexity of the heapify() method is O(log n) as shown above, the complexity for the buildHeap() method is, therefore, maximum\* O(n log n).

\* In the section after the next one, I will show that the time complexity of the buildHeap() method is actually O(n). Since this does not change the overall time complexity, it is not mandatory to perform this in-depth analysis.

### Total Time Complexity of Heapsort

The heapify() method is called n-1 times. So the total complexity for repairing the heap is also O(n log n).

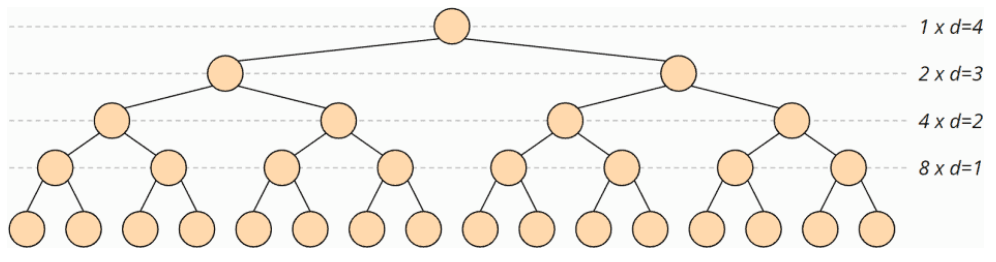
Both sub-algorithms, therefore, have the same time complexity. Hence:

The time complexity of Heapsort is:O(n log n)

### Time Complexity for Building the Heap – In-Depth Analysis

This section is very mathematical and not necessary for determining the time complexity of the overall algorithm (which we have already completed). You could, therefore, skip this section.

We have seen above that the buildHeap() method calls heapify() for each parent node. What we have not considered so far is that the depth of the subtrees, on which heapify() is called, varies. The following graphic illustrates this (d stands for the depth of the subtrees)



The heapify() method is called at most for n/4 trees of depth 1, for n/8 trees of depth 2, for n/16 trees of depth 3, etc.

The maximum number of swap operations in the heapify() method is equal to the depth of the subtree on which it is called.

The maximum number of swap operations Smax is therefore:

Smax = n/4 · 1 + n/8 · 2 + n/16 · 3 + n/32 · 4 + ...

If we multiply both sides of the term by 2, we get:

2 · Smax = n/2 · 1 + n/4 · 2 + n/8 · 3 + n/16 · 4 + ...

Let us place both terms on top of each other:

|  |  |  |
| --- | --- | --- |
| 2 · Smax = | n/2 · 1 + | n/4 · 2 + n/8 · 3 + n/16 · 4 + ... |
| Smax = |  | n/4 · 1 + n/8 · 2 + n/16 · 3 + n/32 · 4 + ... |

Both terms contain n/4, n/8, n/16 etc. with a factor differing by the constant 1. If we subtract the terms, we get:

2 · Smax - Smax = n/2 · 1 + n/4 · (2 - 1) + n/8 · (3 - 2) + n/16 · (4 - 3) + ...

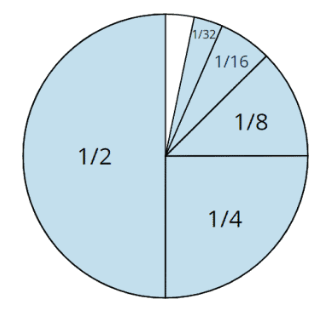
This can be simplified:

Smax = n/2 + n/4 + n/8 + n/16 + ...

Or:

Smax = n · (1/2 + 1/4 + 1/8 + 1/16 + ...)

The term 1/2 + 1/4 + 1/8 + 1/16 + ... approaches 1, as shown in the following diagram:



Thus the formula can finally be simplified to:

Smax ≤ n

We have thus shown that the effort required to build the heap is linear, i.e., the time complexity is O(n).

However, the total complexity of O(n log n) mentioned above does not change due to the lower complexity class of a partial algorithm.

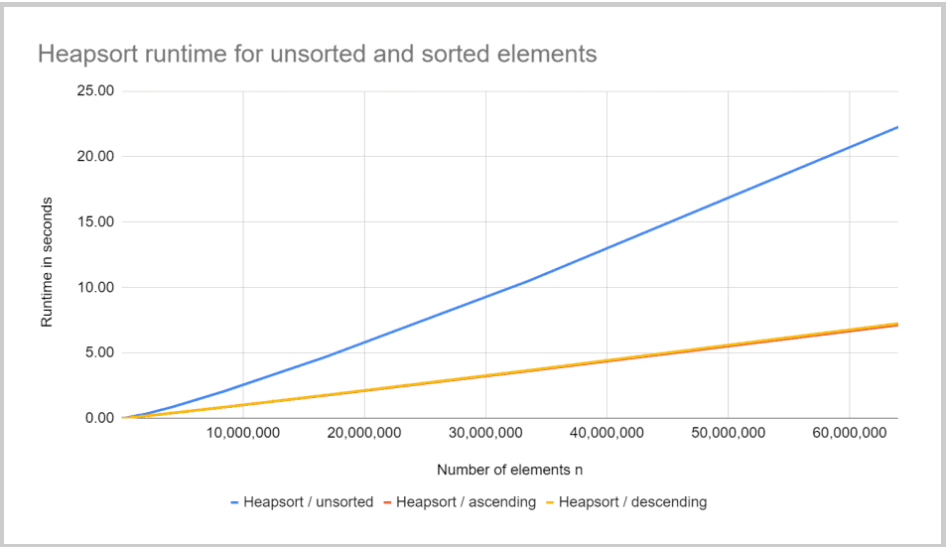
### Runtime of the Java Heapsort Example

The [UltimateSort](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/heapsort/_blank) class can be used to determine the runtime of different sorting algorithms for different input sizes.

The following table shows the medians of the runtimes for sorting randomly arranged, as well as ascending and descending presorted elements, after 50 repetitions (this is only an excerpt for the sake of clarity; the complete result [can be found here](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/results/2020-08-13/UltimateTest_Heapsort.log" \t "https://www.happycoders.eu/algorithms/heapsort/_blank)):

| **n** | **unsorted** | **ascending** | **descending** |
| --- | --- | --- | --- |
| ... | ... | ... | ... |
| 2,097,152 | 369.5 ms | 198.8 ms | 198.8 ms |
| 4,194,304 | 870.2 ms | 410.4 ms | 412.7 ms |
| 8,388,608 | 2,052.4 ms | 848.9 ms | 852.9 ms |
| 16,777,216 | 4,686.9 ms | 1,752.6 ms | 1,775.3 ms |
| 33,554,432 | 10,508.2 ms | 3,623.5 ms | 3,668.7 ms |
| 67,108,864 | 23,459.9 ms | 7,492,4 ms | 7,605.5 ms |

Here are the complete measurements as a diagram:



You can see clearly:

* When doubling the input quantity, sorting takes a little more than twice as long; this corresponds to the expected quasilinear runtime O(n log n).
* For presorted input data, Heapsort is about three times faster than for unsorted data.
* Input data sorted in ascending order will be sorted about as fast as input data sorted in descending order.

### Why Is Heapsort Faster for Presorted Input Data?

To address this question, I use the program [CountOperations](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/demos/CountOperations.java" \t "https://www.happycoders.eu/algorithms/heapsort/_blank) to measure the number of compare, read, and write operations of Heapsort for unsorted, ascending, and descending sorted data for the respective phases.

You can find the result in the file [CountOperations\_Heapsort.log](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/results/2020-08-13/CountOperations_Heapsort.log" \t "https://www.happycoders.eu/algorithms/heapsort/_blank). The results of the test are:

* If the input data is sorted in descending order, there are only about half as many comparisons in phase 1 as there are for unsorted or ascending data; there are also no swap operations. This is because a descending sorted array already corresponds to a max heap.
* Input data sorted in ascending order correspond to a min heap. The tree must be completely reversed in the buildHeap() phase, so in this case, we have about a third more swap operations than with randomly arranged data, in which the heap condition is already fulfilled on some subtrees.
* In phase 2, the number of operations differs only slightly.

Then how can we explain that heapsort is about three times faster for both ascending and descending presorted input data?

We find the answer in the so-called [branch prediction](https://en.wikipedia.org/wiki/Branch_predictor" \t "https://www.happycoders.eu/algorithms/heapsort/_blank).

With presorted input data, the comparison operations always lead to the same result. If the branch prediction now assumes that the comparisons will also lead to the same result in the future, the CPU's instruction pipelines can be fully utilized.

With unsorted input data, however, no reliable statement can be made about future comparison results. As a result, the instruction pipeline must often be deleted and refilled.

## Bottom-Up Heapsort

Bottom-up Heapsort is a variant in which the heapify() method makes do with fewer comparisons through smart optimization. This is advantageous if, for example, we don't compare int primitives, but objects with a time-consuming compareTo() function.

In the regular heapify(), we perform two comparisons on each node from top to bottom to find the largest of three elements:

1. Parent node with left child
2. The larger node from the first comparison with the second child

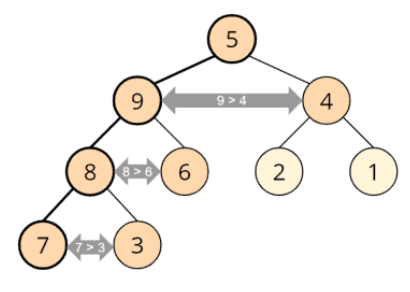
### Bottom-Up Heapsort Algorithm

Bottom-up Heapsort, on the other hand, only compares the two children and follows the larger child to the end of the tree ("top-down"). From there, the algorithm goes back towards the tree root (“bottom-up”) and searches for the first element larger than the root. From this position, all elements are moved one position towards the root, and the root element is placed in the field that has become free.

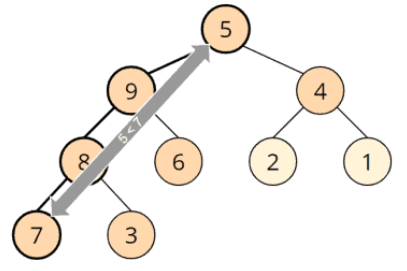
The following example should make it easier to understand.

### Bottom-Up Heapsort Example

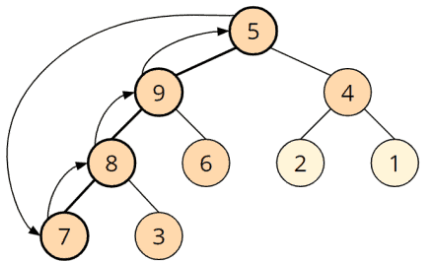
In the following example, we compare the 9 and 4, then the children of the 9 – the 8 and the 6, and finally the children of the 8 – the 7 and the 3:



In this way, we reach the 7 and compare it with the tree root, the 5:



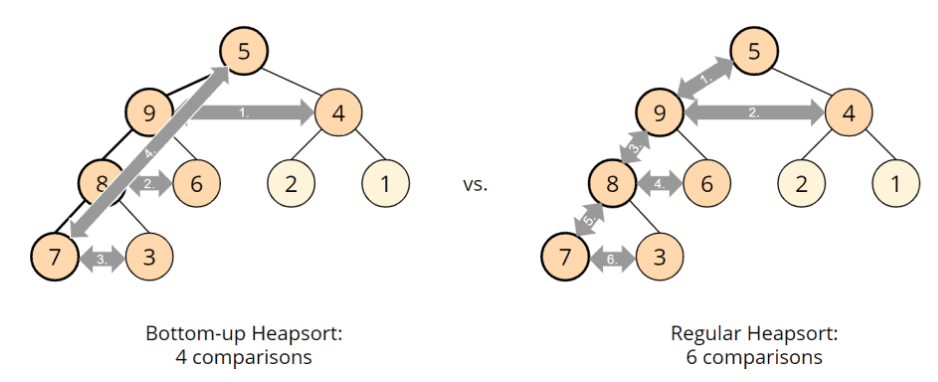
The 5 is smaller than the 7, which means that the root element must be passed all the way down:



In the end, this leads to the same result as the regular heapify().

Bottom-up Heapsort takes advantage of the fact that the root element is usually shifted very far down. The reason is that it comes from the end of the tree after each iteration and is therefore relatively small.

This means that fewer comparisons are necessary if one comparison per node is made all the way down and then a short distance up again – compared to two comparisons per node from top to bottom:



### Bottom-Up Heapsort Source Code

The class [BottomUpHeapsort](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/heapsort/BottomUpHeapsort.java" \t "https://www.happycoders.eu/algorithms/heapsort/_blank) inherits from Heapsort and overwrites its heapify() method with the following:

@Override

void heapify(int[] heap, int length, int rootPos) {

  int leafPos = findLeaf(heap, length, rootPos);

  int nodePos = findTargetNodeBottomUp(heap, rootPos, leafPos);

  if (rootPos == nodePos) return;

  // Move all elements starting at nodePos to parent, move root to nodePos

  int nodeValue = heap[nodePos];

  heap[nodePos] = heap[rootPos];

  while (nodePos > rootPos) {

    int parentPos = getParentPos(nodePos);

    int parentValue = heap[parentPos];

    heap[parentPos] = nodeValue;

    nodePos = getParentPos(nodePos);

    nodeValue = parentValue;

  }

}

The findLeaf() method compares two children and follows the larger one until the end of the tree is reached (or a node with only one child):

int findLeaf(int[] heap, int length, int rootPos) {

  int pos = rootPos;

  int leftChildPos = pos \* 2 + 1;

  int rightChildPos = pos \* 2 + 2;

  // Two child exist?

  while (rightChildPos < length) {

    if (heap[rightChildPos] > heap[leftChildPos]) {

      pos = rightChildPos;

    } else {

      pos = leftChildPos;

    }

    leftChildPos = pos \* 2 + 1;

    rightChildPos = pos \* 2 + 2;

  }

  // One child exist?

  if (leftChildPos < length) {

    pos = leftChildPos;

  }

  return pos;

}

The method findTargetNodeBottomUp() searches from bottom to top for the first element that is not smaller than the root node:

int findTargetNodeBottomUp(int[] heap, int rootPos, int leafPos) {

  int parent = heap[rootPos];

  while (leafPos != rootPos && heap[leafPos] < parent) {

    leafPos = getParentPos(leafPos);

  }

  return leafPos;

}

And finally the getParentPos() method:

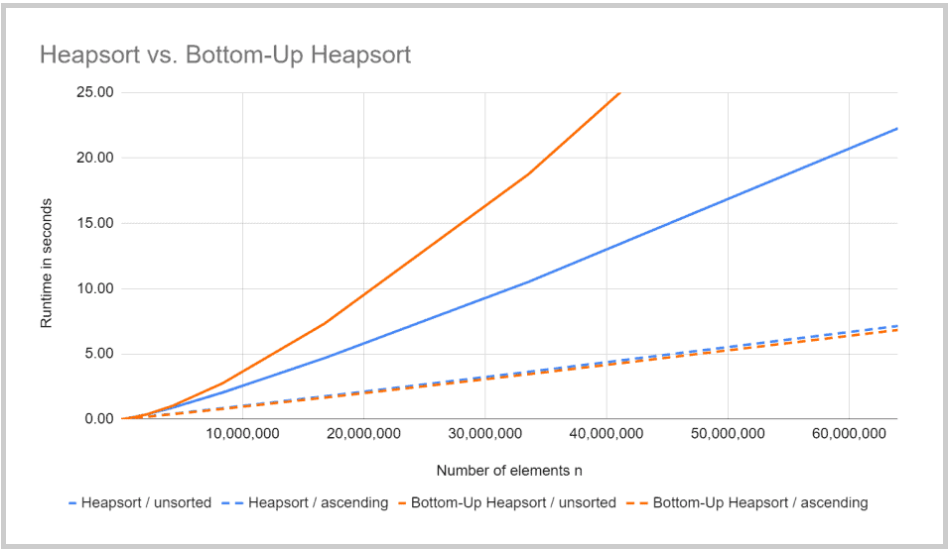
int getParentPos(int pos) {

  return (pos - 1) / 2;

}

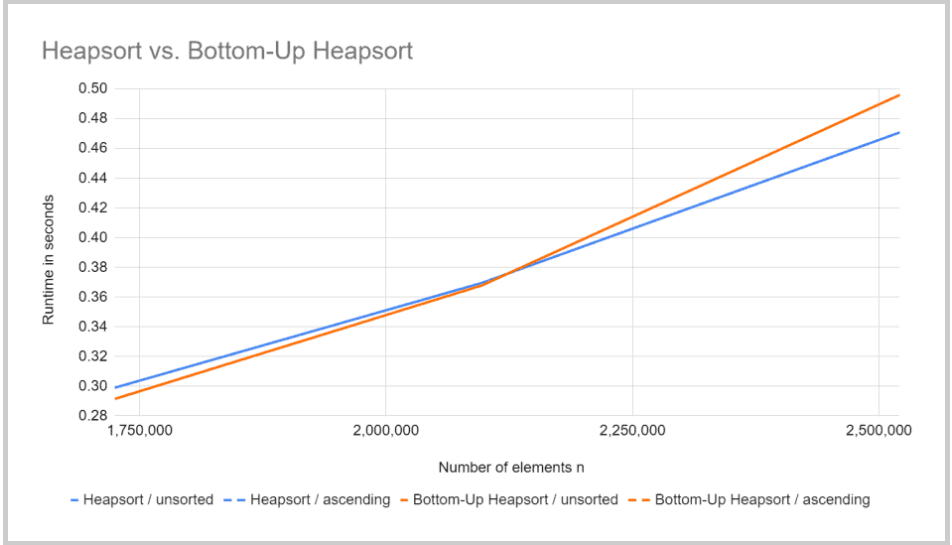
### Bottom-Up Heapsort Performance

We can also measure the performance of Bottom-Up Heapsort with [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/heapsort/_blank). You can find the results in [UltimateTest\_Heapsort.log](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/results/2020-08-13/UltimateTest_Heapsort.log" \t "https://www.happycoders.eu/algorithms/heapsort/_blank). The following diagram shows the runtimes of Bottom-Up Heapsort compared to regular Heapsort:



As you can see, for unsorted data, Bottom-Up Heapsort takes up to twice as long as the regular Heapsort, while it takes about the same time for sorted data.

Before we get to the bottom of the cause, let us first examine a smaller section of the diagram:



Bottom-Up Heapsort only becomes slower than the regular Heapsort, starting at about two million elements.

What is the cause of this?

The [result of the CountOperations program](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/results/2020-08-13/CountOperations_Heapsort.log" \t "https://www.happycoders.eu/algorithms/heapsort/_blank) mentioned above shows that Bottom-Up Heapsort requires fewer compare, read, and write operations than regular heapsort – regardless of the number of elements to be sorted.

Why is it still slower?

Bottom-Up Heapsort is based on the assumption that the root element is always moved down to the leaf level. The branch prediction of the CPU can also make use of this assumption and thus relativize this advantage.

Furthermore, in Bottom-Up Heapsort, we have to go through the tree twice: once from top to bottom and once back to the top. This does not increase the number of operations, but it does affect the access to main memory!

While memory pages only have to be loaded once from the main memory into the CPU cache when traversing the tree once, most memory pages are already removed from the cache and must be reread on the way back if the tree is large enough.

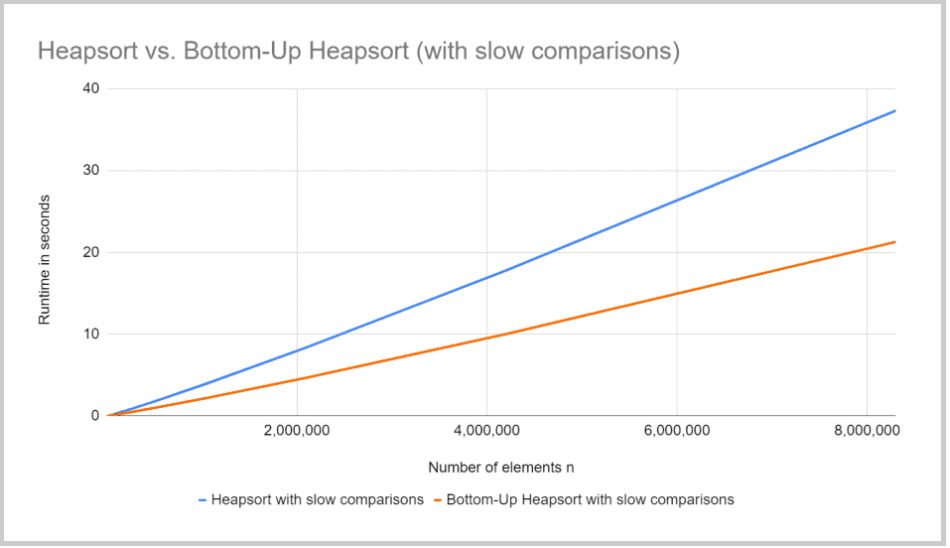
Therefore we approach the speed factor two for sufficiently large trees.

### Bottom-Up Heapsort With Expensive Comparison Operations

Bottom-Up Heapsort is optimized to reduce the number of comparisons required. With int primitives, comparisons are not significant, so Bottom-Up Heapsort cannot show its advantages here.

I have, therefore, carried out another test by artificially increasing the cost of the comparison operations. You can find the adapted algorithms in the classes [HeapsortSlowComparisons](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/heapsort/HeapsortSlowComparisons.java" \t "https://www.happycoders.eu/algorithms/heapsort/_blank) and [BottomUpHeapsortSlowComparisons](https://github.com/SvenWoltmann/sorting-algorithms-ultimate-guide/blob/main/src/main/java/eu/happycoders/sort/method/heapsort/BottomUpHeapsortSlowComparisons.java" \t "https://www.happycoders.eu/algorithms/heapsort/_blank) in the GitHub repository.

Bottom-Up Heapsort performs significantly better in this comparison:



## Further Characteristics of Heapsort

In the following sections, we look at the space complexity of heapsort, its stability, and parallelizability.

### Space Complexity of Heapsort

Heapsort is an in-place sorting method, i.e., no additional memory space is required except for loop and auxiliary variables. The number of these variables is always the same, whether we sort ten elements or ten million. Therefore:

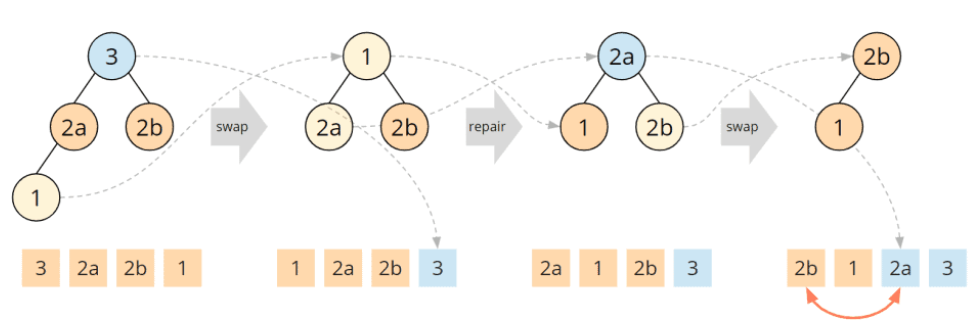
The space complexity of heapsort is: O(1)

### Stability of Heapsort

It is easy to construct examples that show that elements with the same key can change their position to each other:

#### Example 1

When we sort the array [3, 2a, 2b, 1] with Heapsort, we perform the following steps (2a and 2b represent two elements with the same key; highlighted in light yellow are the elements that will be swapped in the next step; highlighted in blue are finished elements):



At this point, we can abort because we can already see that the target array will end in [2a, 3], i.e., 2a will end up to the right of 2b in the target array.

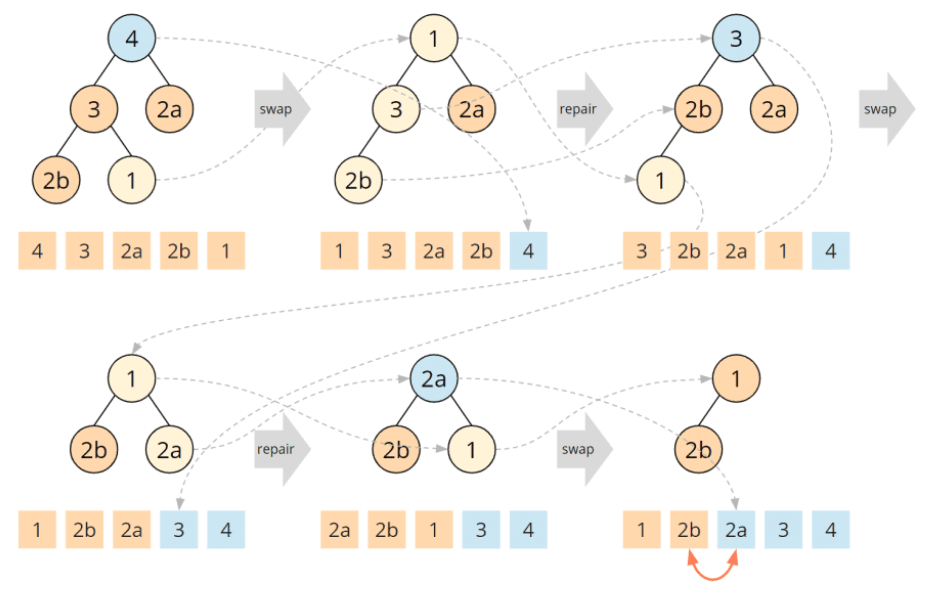
#### Adjust the algorithm?

In the second step, we swapped the 1 with the 2a according to the algorithm. Could we change the algorithm so that for child nodes with the same key, the parent is not swapped with the left child, but with the right one?

In that case, the array above would be sorted stable because the 1 would not be swapped with the 2a, but with the 2b. And then, the 2b would end up at the second last position of the array.

#### Example 2

Let's try this with another input array, with [4, 3, 2a, 2b, 1]:



After step 2, we have reached the state we had before as the initial array, with 2a and 2b having swapped their positions. If we now exchange 1 with the right child in the next step, the same thing happens as above: 2a arrives first in the target array and thus right of 2b.

We have shown counterexamples for both algorithm variants and can therefore state:

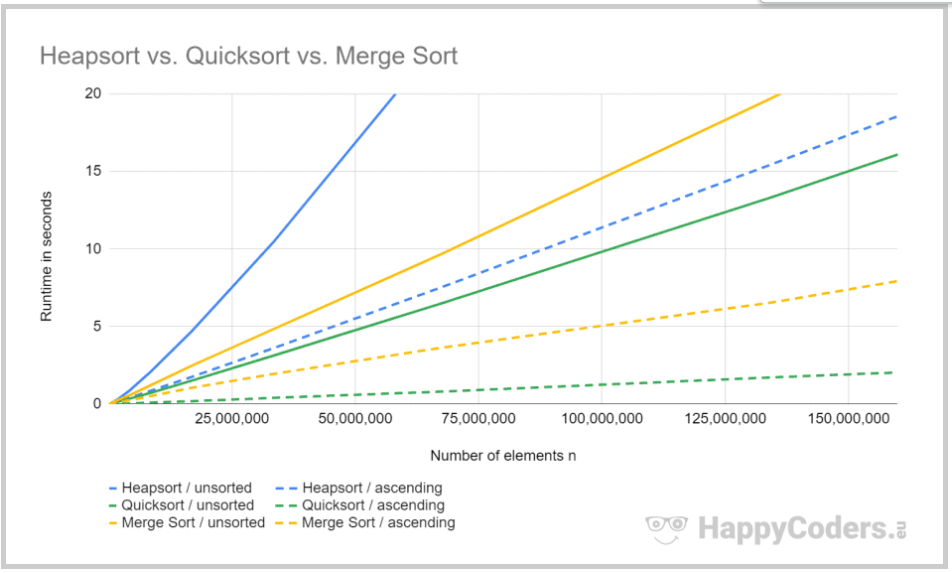
Heapsort is not a stable sorting algorithm.

### Heapsort Parallelizability

With Heapsort, the whole array is continuously changing, so there are no apparent solutions to parallelize the algorithm.

## Comparing Heapsort With Other Efficient Sorting Algorithms

The following diagram shows the UltimateTest results of Heapsort compared to the ones of [Quicksort](https://www.happycoders.eu/algorithms/quicksort/) and [Merge Sort](https://www.happycoders.eu/algorithms/merge-sort/) from the respective articles:



Heapsort is slower than Quicksort by factor 3.6 and slower than Merge Sort by factor 2.4 for randomly distributed input data. For sorted data, heapsort is eight to nine times slower than quicksort and two times slower than Merge Sort.

### Heapsort vs. Quicksort

As shown in the previous section, Quicksort is usually much faster than heapsort.

Due to the O(n²) worst-case time complexity of Quicksort, Heapsort is sometimes preferred to Quicksort in practice.

As shown in the [article about Quicksort](https://www.happycoders.eu/algorithms/quicksort/), if the pivot element is chosen appropriately, the worst case is unlikely to occur. Nevertheless, there is a certain risk that a potential attacker with sufficient knowledge of the Quicksort implementation used can exploit this knowledge to crash or freeze an application with appropriately prepared input data.

### Heapsort vs. Merge Sort

Merge Sort is also usually faster than Heapsort. Besides, unlike Heapsort, Merge Sort is stable.

Heapsort has an advantage over Merge Sort in that it does not require additional memory, while Merge Sort requires additional memory in the order of O(n).

## Summary

Heapsort is an efficient, unstable sorting algorithm with an average, best-case, and worst-case time complexity of O(n log n).

Heapsort is significantly slower than Quicksort and Merge Sort, so Heapsort is less commonly encountered in practice.

You will 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) 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 email when I publish a new article? Then [click here](https://www.happycoders.eu/algorithms/heapsort/) to subscribe to the HappyCoders newsletter.