



# Heapsort

From Wikipedia, the free encyclopedia

**Heapsort** is a [comparison-based sorting algorithm](#) to create a [sorted array](#) (or list), and is part of the [selection sort](#) family. Although somewhat slower in practice on most machines than a well-implemented [quicksort](#), it has the advantage of a more favorable worst-case  $O(n \log n)$  runtime. Heapsort is an [in-place algorithm](#), but is not a [stable sort](#).

## Contents

- [1 Overview](#)
- [2 Variations](#)
- [3 Comparison with other sorts](#)
- [4 Pseudocode](#)
- [5 Example](#)
- [6 Notes](#)
- [7 References](#)
- [8 External links](#)

## Overview

[\[edit\]](#)

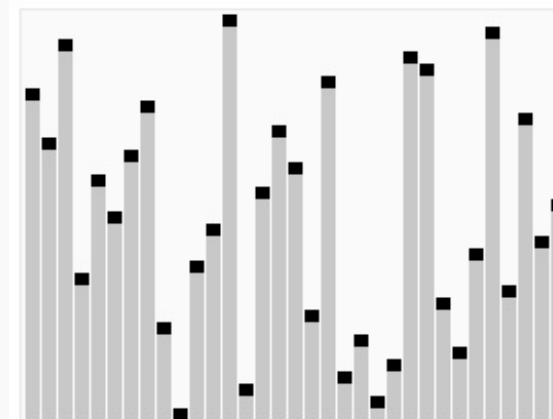
Heapsort is a two step algorithm.

The first step is to [build](#) a [heap](#) out of the data.

The second step begins with removing the largest element from the heap. We insert the removed element into the sorted array. For the first element, this would be position 0 of the array. Next we reconstruct the heap and remove the next largest item, and insert it into the array. After we have removed all the objects from the heap, we have a sorted array. We can vary the direction of the sorted elements by choosing a min-heap or max-heap in step one.

Heapsort can be performed in place. The array can be split into two parts, the sorted

## Heapsort



Arun of the heapsort algorithm sorting an array of randomly permuted values. In the first stage of the algorithm the array elements are reordered to satisfy the [heap property](#). Before the actual sorting takes place, the heap tree structure is shown briefly for illustration.

<b>Class</b>	<a href="#">Sorting algorithm</a>
<b>Data structure</b>	<a href="#">Array</a>
<b>Worst case performance</b>	$O(n \log n)$
<b>Best case performance</b>	$\Omega(n), O(n \log n)$ <sup>[1]</sup>
<b>Average case performance</b>	$O(n \log n)$
<b>Worst case space</b>	$O(n)$ <sub>total</sub> , $O(1)$

## Navigation

[Main page](#)  
[Contents](#)  
[Featured content](#)  
[Current events](#)  
[Random article](#)  
[Donate to Wikipedia](#)

## Interaction

[Help](#)  
[About Wikipedia](#)  
[Community portal](#)  
[Recent changes](#)  
[Contact Wikipedia](#)

## Toolbox

[What links here](#)  
[Related changes](#)  
[Upload file](#)  
[Special pages](#)  
[Permanent link](#)  
[Cite this page](#)

[Print/export](#)

[Create a book](#)

array and the heap. The storage of heaps as arrays is diagrammed at [Binary heap#Heap implementation](#). The heap's invariant is preserved after each extraction, so the only cost is that of extraction. Heapsort uses two heap operations: insertion and root deletion.

## Variations

[\[edit\]](#)

- The most important variation to the simple variant is an improvement by [R. W. Floyd](#) that, in practice, gives about a 25% speed improvement by using only one comparison in each [siftup](#) run, which must be followed by a [siftdown](#) for the original child. Moreover, it is more elegant to formulate. Heapsort's natural way of indexing works on indices from 1 up to the number of items. Therefore the start address of the data should be shifted such that this logic can be implemented avoiding unnecessary +/- 1 offsets in the coded algorithm.
- Ternary heapsort<sup>[2]</sup> uses a ternary heap instead of a binary heap; that is, each element in the heap has three children. It is more complicated to program, but does a constant number of times fewer swap and comparison operations. This is because each step in the shift operation of a ternary heap requires three comparisons and one swap, whereas in a binary heap two comparisons and one swap are required. The ternary heap does two steps in less time than the binary heap requires for three steps, which multiplies the index by a factor of 9 instead of the factor 8 of three binary steps. Ternary heapsort is about 12% faster than the simple variant of binary heapsort.<sup>[citation needed]</sup>
- The [smoothsort](#) algorithm<sup>[3][4]</sup> is a variation of heapsort developed by [Edsger Dijkstra](#) in 1981. Like heapsort, smoothsort's upper bound is  $O(n \log n)$ . The advantage of smoothsort is that it comes closer to  $O(n)$  time if the [input is already sorted to some degree](#), whereas heapsort averages  $O(n \log n)$  regardless of the initial sorted state. Due to its complexity, smoothsort is rarely used.
- [Levcopoulos and Petersson](#)<sup>[5]</sup> describe a variation of heapsort based on a [Cartesian tree](#) that does not add an element to the heap until smaller values on both sides of it have already been included in the sorted output. As they show, this modification can allow the algorithm to sort more quickly than  $O(n \log n)$  for inputs that are already nearly sorted.

## Comparison with other sorts

[\[edit\]](#)

Heapsort primarily competes with [quicksort](#), another very efficient general purpose nearly-in-place comparison-based sort algorithm.

Quicksort is typically somewhat faster, due to better cache behavior and other factors, but the worst-case running time for quicksort is  $O(n^2)$ , which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation, creating a security risk. See [quicksort](#) for a detailed discussion of this problem, and possible solutions.

Thus, because of the  $O(n \log n)$  upper bound on heapsort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heapsort.

Heapsort also competes with [merge sort](#), which has the same time bounds, but requires  $\Omega(n)$  auxiliary space, whereas heapsort requires only a constant amount. Heapsort also typically runs more quickly in practice on machines with small or slow [data caches](#). On the other hand, merge sort has several advantages over heapsort:

- Like quicksort, merge sort on arrays has considerably better data cache performance, often outperforming heapsort on a modern

desktop PC, because it accesses the elements in order.

- Merge sort is a [stable sort](#).
- Merge sort [parallelizes better](#); the most trivial way of parallelizing merge sort achieves close to [linear speedup](#), while there is no obvious way to parallelize heapsort at all.
- Merge sort can be easily adapted to operate on [linked lists](#) (with  $O(1)$  extra space<sup>[6]</sup>) and very large lists stored on slow-to-access media such as [disk storage](#) or [network attached storage](#). Heapsort relies strongly on [random access](#), and its poor [locality of reference](#) makes it very slow on media with long access times. (Note: Heapsort can also be applied to doubly linked lists with only  $O(1)$  extra space overhead)<sup>[citation needed]</sup>

[Introsort](#) is an interesting alternative to heapsort that combines quicksort and heapsort to retain advantages of both: worst case speed of heapsort and average speed of quicksort.

## Pseudocode

[\[edit\]](#)

The following is the "simple" way to implement the algorithm in [pseudocode](#). Arrays are [zero-based](#) and *swap* is used to exchange two elements of the array. Movement 'down' means from the root towards the leaves, or from lower indices to higher. Note that during the sort, the largest element is at the root of the heap at  $a[0]$ , while at the end of the sort, the largest element is in  $a[\text{end}]$ .

```
function heapSort(a, count) is
    input:  an unordered array a of length count

    (first place a in max-heap order)
    heapify(a, count)

    end := count-1 //in languages with zero-based arrays the children are 2*i+1 and 2*i+2
    while end > 0 do
        (swap the root(maximum value) of the heap with the last element of the heap)
        swap(a[end], a[0])
        (decrease the size of the heap by one so that the previous max value will
        stay in its proper placement)
        end := end - 1
        (put the heap back in max-heap order)
        siftDown(a, 0, end)

function heapify(a, count) is
    (start is assigned the index in a of the last parent node)
    start := (count - 2) / 2

    while start ≥ 0 do
        (sift down the node at index start to the proper place such that all nodes below
```

```

        the start index are in heap order)
    siftDown(a, start, count-1)
    start := start - 1
    (after sifting down the root all nodes/elements are in heap order)

function siftDown(a, start, end) is
    input:   end represents the limit of how far down the heap
              to sift.
    root := start

    while root * 2 + 1 ≤ end do                (While the root has at least one child)
        child := root * 2 + 1                (root*2 + 1 points to the left child)
        swap := root                        (keeps track of child to swap with)
        (check if root is smaller than left child)
        if a[swap] < a[child]
            swap := child
            (check if right child exists, and if it's bigger than what we're currently swapping with)
            if child+1 ≤ end and a[swap] < a[child+1]
                swap := child + 1
            (check if we need to swap at all)
            if swap ≠ root
                swap(a[root], a[swap])
                root := swap                (repeat to continue sifting down the child now)
        else
            return

```

The heapify function can be thought of as building a heap from the bottom up, successively shifting downward to establish the [heap property](#). An alternative version (shown below) that builds the heap top-down and sifts upward may be conceptually simpler to grasp. This "siftUp" version can be visualized as starting with an empty heap and successively inserting elements, whereas the "siftDown" version given above treats the entire input array as a full, "broken" heap and "repairs" it starting from the last non-trivial sub-heap (that is, the last parent node).

Also, the "siftDown" version of heapify [has  \$O\(n\)\$  time complexity](#), while the "siftUp" version given below has  $O(n \log n)$  time complexity due to its equivalence with inserting each element, one at a time, into an empty heap.<sup>[7]</sup> This may seem counter-intuitive since, at a glance, it is apparent that the former only makes half as many calls to its logarithmic-time sifting function as the latter; i.e., they seem to differ only by a constant factor, which never has an impact on asymptotic analysis.

To grasp the intuition behind this difference in complexity, note that the number of swaps that may occur during any one siftUp call *increases* with the depth of the node on which the call is made. The crux is that there are many (exponentially many) more "deep" nodes than there are "shallow" nodes in a heap, so that siftUp may have its full logarithmic running-time on the approximately linear number of calls made on the nodes at or near the "bottom" of the heap. On the other hand, the number of swaps that may occur during any one siftDown call *decreases* as the depth of the node on which the call is made increases. Thus, when the "siftDown" heapify begins and is

calling siftDown on the bottom and most numerous node-layers, each sifting call will incur, at most, a number of swaps equal to the "height" (from the bottom of the heap) of the node on which the sifting call is made. In other words, about half the calls to siftDown will have at most only one swap, then about a quarter of the calls will have at most two swaps, etc.

The heapsort algorithm itself has  $O(n \log n)$  time complexity using either version of heapify.

```
function heapify(a, count) is
    (end is assigned the index of the first (left) child of the root)
    end := 1

    while end < count
        (sift up the node at index end to the proper place such that all nodes above
         the end index are in heap order)
        siftUp(a, 0, end)
        end := end + 1
    (after sifting up the last node all nodes are in heap order)

function siftUp(a, start, end) is
    input: start represents the limit of how far up the heap to sift.
           end is the node to sift up.
    child := end
    while child > start
        parent := floor((child - 1) ÷ 2)
        if a[parent] < a[child] then (out of max-heap order)
            swap(a[parent], a[child])
            child := parent (repeat to continue sifting up the parent now)
        else
            return
```

## Example

[\[edit\]](#)

Let { 6, 5, 3, 1, 8, 7, 2, 4 } be the list that we want to sort from the smallest to the largest. (NOTE, for 'Building the Heap' step: Larger nodes don't stay below smaller node parents. They are swapped with parents, and then recursively checked if another swap is needed, to keep larger numbers above smaller numbers on the heap binary tree.)

### 1. Build the heap

## 1. Build the heap

Heap	newly added element	swap elements
nul	6	
6	5	
6, 5	3	
6, 5, 3	1	
6, 5, 3, 1	8	
6, <b>5</b> , 3, 1, <b>8</b>		5, 8
<b>6</b> , <b>8</b> , 3, 1, 5		6, 8
8, 6, 3, 1, 5	7	
8, 6, <b>3</b> , 1, 5, <b>7</b>		3, 7
8, 6, 7, 1, 5, 3	2	
8, 6, 7, 1, 5, 3, 2	4	
8, 6, 7, <b>1</b> , 5, 3, 2, <b>4</b>		1, 4
8, 6, 7, 4, 5, 3, 2, 1		

6 5 3 1 8 7 2 4

An example on heapsort.



## 2. Sorting.

Heap	swap elements	delete element	sorted array	details
<b>8</b> , 6, 7, 4, 5, 3, 2, <b>1</b>	8, 1			swap 8 and 1 in order to delete 8 from heap
1, 6, 7, 4, 5, 3, 2, <b>8</b>		8		delete 8 from heap and add to sorted array
<b>1</b> , 6, 7, 4, 5, 3, 2	1, 7		8	swap 1 and 7 as they are not in order in the heap
7, 6, <b>1</b> , 4, 5, <b>3</b> , 2	1, 3		8	swap 1 and 3 as they are not in order in the heap
<b>7</b> , 6, 3, 4, 5, 1, <b>2</b>	7, 2		8	swap 7 and 2 in order to delete 7 from heap
2, 6, 3, 4, 5, 1, <b>7</b>		7	8	delete 7 from heap and add to sorted array
<b>2</b> , <b>6</b> , 3, 4, 5, 1	2, 6		7, 8	swap 2 and 6 as they are not in order in the heap
6, <b>2</b> , 3, 4, <b>5</b> , 1	2, 5		7, 8	swap 2 and 5 as they are not in order in the heap
<b>6</b> , 5, 3, 4, 2, <b>1</b>	6, 1		7, 8	swap 6 and 1 in order to delete 6 from heap
1, 5, 3, 4, 2, <b>6</b>		6	7, 8	delete 6 from heap and add to sorted array
<b>1</b> , <b>5</b> , 3, 4, 2	1, 5		6, 7, 8	swap 1 and 5 as they are not in order in the heap

5, 1, 3, 4, 2	1, 4		6, 7, 8	swap 1 and 4 as they are not in order in the heap
5, 4, 3, 1, 2	5, 2		6, 7, 8	swap 5 and 2 in order to delete 5 from heap
2, 4, 3, 1, 5		5	6, 7, 8	delete 5 from heap and add to sorted array
2, 4, 3, 1	2, 4		5, 6, 7, 8	swap 2 and 4 as they are not in order in the heap
4, 2, 3, 1	4, 1		5, 6, 7, 8	swap 4 and 1 in order to delete 4 from heap
1, 2, 3, 4		4	5, 6, 7, 8	delete 4 from heap and add to sorted array
1, 2, 3	1, 3		4, 5, 6, 7, 8	swap 1 and 3 as they are not in order in the heap
3, 2, 1	3, 1		4, 5, 6, 7, 8	swap 3 and 1 in order to delete 3 from heap
1, 2, 3		3	4, 5, 6, 7, 8	delete 3 from heap and add to sorted array
1, 2	1, 2		3, 4, 5, 6, 7, 8	swap 1 and 2 as they are not in order in the heap
2, 1	2, 1		3, 4, 5, 6, 7, 8	swap 2 and 1 in order to delete 2 from heap
1, 2		2	3, 4, 5, 6, 7, 8	delete 2 from heap and add to sorted array
1		1	2, 3, 4, 5, 6, 7, 8	delete 1 from heap and add to sorted array
			1, 2, 3, 4, 5, 6, 7, 8	completed

## Notes

[\[edit\]](#)




1. <sup>^</sup> <http://dx.doi.org/10.1006/jagm.1993.1031>
2. <sup>^</sup> "Data Structures Using Pascal", 1991, page 405, gives a ternary heapsort as a student exercise. "Write a sorting routine similar to the heapsort except that it uses a ternary heap."
3. <sup>^</sup> <http://www.cs.utexas.edu/users/EWD/ewd07xx/EWD796a.PDF>
4. <sup>^</sup> <http://www.cs.utexas.edu/~EWD/transcriptions/EWD07xx/EWD796a.html>
5. <sup>^</sup> Levcopoulos, Christos; Petersson, Ola (1989), "Heapsort - Adapted for Presorted Files", *WADS '89: Proceedings of the Workshop on Algorithms and Data Structures*, Lecture Notes in Computer Science, **382**, London, UK: Springer-Verlag, pp. 499–509, DOI:10.1007/3-540-51542-9\_41.
6. <sup>^</sup> [Merge sort Wikipedia page](#)
7. <sup>^</sup> "Priority Queues". Retrieved 24 May 2011.

## References

[\[edit\]](#)









- **J. W. J. Williams**. *Algorithm 232 - Heapsort* , 1964, Communications of the ACM 7(6): 347–348.
- **Robert W. Floyd**. *Algorithm 245 - Treesort 3* , 1964, Communications of the ACM 7(12): 701.
- **Svante Carlsson**, *Average-case results on heapsort* , 1987, BIT 27(1): 2-17.
- **Donald Knuth**. *The Art of Computer Programming* , Volume 3: *Sorting and Searching* , Third Edition. Addison-Wesley, 1997. ISBN 0-201-

89685-0. Pages 144–155 of section 5.2.3: Sorting by Selection.

- [Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. \*Introduction to Algorithms\*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapters 6 and 7 Respectively: Heapsort and Priority Queues](#)
- [A PDF of Dijkstra's original paper on Smoothsort](#) 
- [Heaps and Heapsort Tutorial](#)  by David Carlson, St. Vincent College
- [Heaps of Knowledge](#) 

## External links

[\[edit\]](#)

- [Animated Sorting Algorithms: Heap Sort](#)  – graphical demonstration and discussion of heap sort
- [Courseware on Heapsort from Univ. Oldenburg](#)  - With text, animations and interactive exercises
- [NIST's Dictionary of Algorithms and Data Structures: Heapsort](#) 
- [Heapsort implemented in 12 languages](#) 
- [Sorting revisited](#)  by Paul Hsieh
- [A color graphical Java applet](#)  that allows experimentation with initial state and shows statistics
- [A PowerPoint presentation demonstrating how Heap sort works](#)  that is for educators.
- [Open Data Structures - Section 11.1.3 - Heap-Sort](#) 



The Wikibook *Algorithm implementation* has a page on the topic of **Heapsort**

V · T · E · <span>Sorting algorithms</span>	
<b>Theory</b>	Computational complexity theory · Big O notation · Total order · Lists · Stability · Comparison sort · Adaptive sort · Sorting network · Integer sorting ·
<b>Exchange sorts</b>	Bubble sort · Cocktail sort · Odd–even sort · Comb sort · Gnome sort · Quicksort · Stooge sort · Bogosort ·
<b>Selection sorts</b>	Selection sort · <b>Heapsort</b> · Smoothsort · Cartesian tree sort · Tournament sort · Cycle sort ·
<b>Insertion sorts</b>	Insertion sort · Shellsort · Tree sort · Library sort · Patience sorting ·
<b>Merge sorts</b>	Merge sort · Polyphase merge sort · Strand sort ·
<b>Distribution sorts</b>	American flag sort · Bead sort · Bucket sort · Burtsort · Counting sort · Pigeonhole sort · Proxmap sort · Radix sort · Flashsort ·
<b>Concurrent sorts</b>	Bitonic sorter · Batcher odd–even mergesort · Pairwise sorting network ·
<b>Hybrid sorts</b>	Timsort · Introsort · Spreadsort · UnShuffle sort · JSort ·
<b>Other</b>	Topological sorting · Pancake sorting · Spaghetti sort ·

Categories: [Sorting algorithms](#) | [Comparison sorts](#) | [Heaps \(data structures\)](#)



Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. See [Terms of use](#) for details.  
Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Contact us](#)

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Mobile view](#)

