Overview

The heapsort algorithm can be divided into two parts.

In the first step, a heap is built out of the data. The heap is often placed in an array with the layout of a complete binary tree. The complete binary tree maps the binary tree structure into the array indices; each array index represents a node; the index of the node's parent, left child branch, or right child branch are simple expressions. For a zero-based array, the root node is stored at index 0; if `i` is the index of the current node, then

```
iParent(i)     = floor((i-1) / 2)
iLeftChild(i)  = 2*i + 1
iRightChild(i) = 2*i + 2
```

In the second step, a sorted array is created by repeatedly removing the largest element from the heap (the root of the heap), and inserting it into the array. The heap is updated after each removal to maintain the heap. Once all objects have been removed from the heap, the result is a sorted array.

Heapsort can be performed in place. The array can be split into two parts, the sorted array and the heap. The storage of heaps as arrays is diagrammed here. The heap's invariant is preserved after each extraction, so the only cost is that of extraction.

Algorithm

The heapsort algorithm involves preparing the list by first turning it into a max heap. The algorithm then repeatedly swaps the first value of the list with the last value, decreasing the range of values considered in the heap operation by one, and sifting the new first value into its position in the heap. This repeats until the range of considered values is one value in length.

The steps are:

1. Call the buildMaxHeap() function on the list. Also referred to as heapify(), this builds a heap from a list in O(n) operations.
2. Swap the first element of the list with the final element. Decrease the considered range of the list by one.
3. Call the siftDown() function on the list to sift the new first element to its appropriate index in the heap.
4. Go to step (2) unless the considered range of the list is one element.

The buildMaxHeap() operation is run once, and is O(n) in performance. The siftDown() function is O(log(n)), and is called n times. Therefore, the performance of this algorithm is O(n + n * log(n)) which evaluates to O(n log n).

Pseudocode

The following is a 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[end]`.

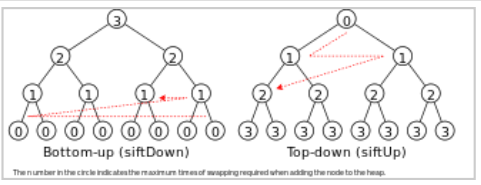
```
procedure heapsort(a, count) is
    input: an unordered array a of length count

    (Build the heap in array a so that largest value is at the root)
    heapify(a, count)

    (The following loop maintains the invariants that a[0:end] is a heap and every element
     beyond end is greater than everything before it (so a[end:count] is in sorted order))
    end ← count - 1
    while end > 0 do
        (a[0] is the root and largest value. The swap moves it in front of the sorted elements.)
        swap(a[end], a[0])
        (the heap size is reduced by one)
        end ← end - 1
        (the swap ruined the heap property, so restore it)
        siftDown(a, 0, end)
```

The sorting routine uses two subroutines, `heapify` and `siftDown`. The former is the common in-place heap construction routine, while the latter is a common subroutine for implementing `heapify`.

```
(Put elements of 'a' in heap order, in-place)
procedure heapify(a, count) is
    (start is assigned the index in 'a' of the last parent node)
    (the last element in a 0-based array is at index count-1; find the parent of that element)
    start ← iParent(count-1)

    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)
        (go to the next parent node)
        start ← start - 1
    (after sifting down the root all nodes/elements are in heap order)

(Repair the heap whose root element is at index 'start', assuming the heaps rooted at its children are valid)
procedure siftDown(a, start, end) is
    root ← start

    while iLeftChild(root) ≤ end do    (While the root has at least one child)
        child ← iLeftChild(root)   (Left child of root)
        swap ← root                (Keeps track of child to swap with)

        if a[swap] < a[child]
            swap ← child
        (If there is a right child and that child is greater)
        if child+1 ≤ end and a[swap] < a[child+1]
            swap ← child + 1
        if swap = root
            (The root holds the largest element. Since we assume the heaps rooted at the
             children are valid, this means that we are done.)
            return
        else
            swap(a[root], a[swap])
            root ← swap            (repeat to continue sifting down the child now)
```

The `heapify` procedure can be thought of as building a heap from the bottom up by successively sifting downward to establish the heap property. An alternative version (shown below) that builds the heap top-down and sifts upward may be simpler to understand. 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 but "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.[5] 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.



Difference in time complexity between the "siftDown" version and the "siftUp" version.

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.

```
procedure 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)

procedure 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 := iParent(child)
        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
```

Variations

- The most important variation to the simple variant is an improvement by Floyd that uses only one comparison in each siftup run, which must be followed by a siftdown for the original child. The worst-case number of comparisons during the Floyd's heap-construction phase of Heapsort is known to be equal to $2N - 2s_2(N) - e_2(N)$, where $s_2(N)$ is the sum of all digits of the binary representation of N and $e_2(N)$ is the exponent of 2 in the prime factorization of N.[6]
- Ternary heapsort[7] 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.

- The **smoothsort** algorithm[8][9] 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[10] 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.

Bottom-up heapsort

Bottom-up heapsort was announced as beating quicksort (with median-of-three pivot selection) on arrays of size ≥16000.[11] This version of heapsort keeps the linear-time heap-building phase, but changes the second phase, as follows. Ordinary heapsort extracts the top of the heap, $a[0]$, and fills the gap it leaves with $a[end]$, then sifts this latter element down the heap; but this element comes from the lowest level of the heap, meaning it is one of the smallest elements in the heap, so the sift-down will likely take many steps to move it back down. Bottom-up heapsort instead finds the element to fill the gap, by tracing a path of maximum children down the heap as before, but then sifts that element *up* the heap, which is likely to take fewer steps.[12]

> **Note:** This code uses 1-based arrays.

```
function leafSearch(a, end, i) is
    j ← i
    while 2×j ≤ end do
        (Determine which of j's children is the greater)
        if 2×j+1 < end and a[2×j+1] > a[2×j] then
            j ← 2×j+1
        else
            j ← 2×j
    return j
```

The return value of the `leafSearch` is used in a replacement for the `siftDown` routine:[12]

```
function siftDown(a, end, i) is
    j ← leafSearch(a, end, i)
    while a[i] > a[j] do
        j ← parent(j)
    x ← a[j]
    a[j] ← a[i]
    while j > i do
        swap x, a[parent(j)]
        j ← parent(j)
```

Bottom-up heapsort requires only $1.5\ n \log n + O(n)$ comparisons in the worst case and $n \log n + O(1)$ on average. A 2008 re-evaluation of this algorithm showed it to be no faster than ordinary heapsort, though, presumably because modern branch prediction nullifies the cost of the comparisons that bottom-up heapsort manages to avoid.[13]

Comparison with other sorts