$\texttt{BUILDHEAP}(H)$ and $\texttt{SORTHEAP}(H)$ and $\texttt{SORTARRAY}(H)$

This routine begins with an array H of records in no particular order, with $n = \texttt{length}(H)$ the number of records. It ends with the same records in the same array but now the array is a heap. The algorithm is quite simple:
$\texttt{BUILDHEAP}(H)$
$n \leftarrow \texttt{length}(H)$ (* Just for convenience of understanding *)
For $i = n$ down to 1
$\quad\texttt{Heapify}-\texttt{down}(H, i)$
Endfor

Why does this work? $\texttt{Heapify}-\texttt{down}(H, i)$ looks only at the subtree consisting of i and all of its descendents. *If* that is a heap except maybe position i (the root of the subtree) is too big then it fixes it and the whole subtree has the heap property. From $i = n$ down to $\lfloor n/2 \rfloor + 1$, i has no descendents and so $\texttt{Heapify}-\texttt{down}(H, i)$ does nothing. (Indeed, we can have the for loop begin at $i = \lfloor n/2 \rfloor$. Now we show by induction that when we reach i and apply $\texttt{Heapify}-\texttt{down}(H, i)$ that the subtree with root i has the heap property. Suppose this is true until you reach i and i has children $x = 2i, y = 2i + 1$. By induction the subtrees with roots x, y became heaps when we applied $\texttt{Heapify}-\texttt{down}(H, x)$ and $\texttt{Heapify}-\texttt{down}(H, y)$ and those elements haven't been touched since. So the only problem with the subtree with root i is what is at its root, and so $\texttt{Heapify}-\texttt{down}(H, i)$ fixes it. When we finish $i = 1$ the subtree with root 1, that is, the whole tree, is a heap.

How long does this take? Let $T(n)$ be the time for $\texttt{BUILDHEAP}$, with an array of size n. Each application of $\texttt{Heapify}-\texttt{down}(H, i)$ takes time $O(\lg n)$ and so the total time is $O(n \lg n)$. While this is technically true (in the sense that you are less than 132 years old) we can say more. Actually, the total time is $O(n)$.

The time for $\texttt{Heapify}-\texttt{down}(H, i)$ is the distance from i to the leaves. For the $n/2$ values which are leaves the time is zero. For the $n/4$ values i which have children but not grandchilden the time is 1. (We are measuring time by the number of switches when we apply $\texttt{Heapify}-\texttt{down}(H, i)$. In general, the $n/2^j$ values i whose ditance to the leaves is j have time j. The key is that most of the time $\texttt{Heapify}-\texttt{down}(H, i)$ has a very small time. But lets be more precise. Let $g(n)$ be the number of switches. Then

$$g(n) \leq \frac{n}{2} \cdot 0 + \frac{n}{4} \cdot 1 + \frac{n}{8} \cdot 2 + \ldots$$

Taking out the common factor of n we have

$$g(n) \leq n \left[\frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \frac{4}{64} + \cdots \right]$$

We bound the sum by an infinite sum

$$g(n) \leq n \cdot \sum_{i=2}^{\infty} \frac{i-1}{2^i}$$

Intriguingly, this sum is precisely one! One way to see this is to make an infinite table:

1/4	1/8	1/16	1/32	1/64
-	1/8	1/16	1/32	1/64
-	-	1/16	1/32	1/64
-	-	-	1/32	1/64
-	-	-	-	1/64

The column sums are the terms we have. The runs are geometric series and sum to $1/2, 1/4, 1/8, \ldots$ respectively. The sum of the column sums is equal to the sum of the row sums which is a geometric series which sums to one. So $g(n) \leq n$ and hence $T(n) = O(n)$. We have the lower bound $T(n) = \Omega(n)$ since you can't make a heap without looking at all the data. Thus `BUILDHEAP` is a $\Theta(n)$ algorithm.

`SORTHEAP`(H) takes a heap of length n and turns it into a sorted array.

`SORTHEAP`(H)
$n \leftarrow$ `length`(H)
For $i = 1$ to $n-1$
    `TEMP` $\leftarrow$ `ExtractMin`(H) (* This places $H(1)$ in `TEMP` and rearranges rest of heap *)
    $H($`length`$(H)) \leftarrow$ `TEMP`
    `length`$(H) \leftarrow$ `length`$(H) - 1$
endfor
`length`$(H) \leftarrow n$ (* return length to original value *)

The FOR loop is being done $n-1$ times. On the first time the smallest element (which is $H(1)$) is moved to $H(n)$, the `length` is moved down to $n-1$ and now $H(1), \ldots, H(n-1)$ form a heap. Now the current $H(1)$ (so that will be the second smallest element) is moved to $H(n-1)$ and now $H(1), \ldots, H(n-2)$ form a heap. We continue removing the smallest elements and putting them at the end. (A subtlety here. The array H will always have n values in it. However, we will be lowering the value $\texttt{length}(H)$ to $s = n-1, n-2, \ldots$ down to 2. This is important because when $\texttt{length}(H) = s$ the subroutine $\texttt{ExtractMin}(H)$ is only looking at the terms $H(1), \ldots, H(s)$ and is leaving the other terms alone. The algorithm then puts $H(1)$ in the $H(s)$ position and shuffles around $H(2), \ldots, H(s)$ so that they are now the first $s-1$ positions and form a heap of length $s-1$.) At the end of the $n-1$ iterations the $H(1), \ldots, H(n)$ are in *reverse* order, the largest one first. (This can easily be reversed with a $O(n)$ algorithm if needed.)

As each application of `ExtractMin` takes time $O(\lg n)$ and there are $n-1$ applications the total time is $O(n \lg n)$. (It looks like there is something to be gained further here as `ExtractMin` is done on heaps of length $n, n-1, \ldots$ down to 2 and the later ones are quicker but it turns out that that gain is negligible.)

If we put these two algorithms together we get a sorting algorithm.

$\texttt{SORTARRAY}(H)$

  $\texttt{BUILDHEAP}(H)$

  $\texttt{SORTHEAP}(H)$

This takes a totally unsorted array H, first makes it a heap, and then makes it a sorted array. The time is $O(n)$ for the first part, $O(n \lg n)$ for the second part, so the total time is $O(n \lg n)$. (This matches the best other sorts we shall examine!)