Heap sort:

Let H = min-ordered binary heap, initially empty
A[1…n] = input array

```
for (k=1; k<=n; k++)
      H.insert (A[k]);
for (k=1; k<=n; k++)
      A[k] = H.removeMin( );
```

Analysis of heap sort:

- Each operation insert, removeMin takes θ(lg n) time.
- So θ(n lg n) total time for n inserts, n removeMins.

In-place Heap sort:

Use a max-ordered binary heap.
Store this heap in the same array A[1..n].
Values in A[1…k] are currently in the heap, so heap size is k.

```
for (k=1; k<=n; k++)
      A.insert (A[k]);
for (k=n; k>=1; k--)
      A[k] = A.removeMax( );
```

1	2	3	4	5	6	7	8
26	48	17	31	50	9	21	16
26							
48	26						
48	26	17					
48	31	17	26				
50	48	17	26	31			
50	48	17	26	31	9		
50	48	21	26	31	9	17	
50	48	21	26	31	9	17	16
48	31	21	26	16	9	17	50
31	26	21	17	16	9	48	
26	17	21	9	16	31		
21	17	16	9	26			
17	9	16	21				
16	9	17					
9	16						
9							

Heap elements highlighted in yellow

Analysis of in-place heap sort:

- Each operation insert, removeMax takes $\theta(\lg n)$ time.
- So $\theta(n \lg n)$ total time for n inserts, n removeMaxes.

Note: there is a faster way to build the heap in only $\theta(n)$ time, but it would still take $\theta(n \lg n)$ time to do the n removeMaxes.

Build-Heap in $\theta(n)$ time