The Heapsort Algorithm

```
void max-heapify( T[] A, Integer n, Integer i )
    p = i;
    while ( 2p ≤ n )
        if ( 2p+1 ≤ n  and  A[2p+1] > A[2p] )
            m = 2p+1;
        else
            m = 2p;
        if ( A[p] < A[m] )
            swap( A[p], A[m]);
            p = m;
        else
            return;
```

Initially: A is an array of size at least n, and $1 \le i \le n$. The *max-heap property* holds everywhere in the subtree of $A[1..n]$ rooted at $A[i]$, except possibly at $A[i]$ itself.

Upon return: The subtree of $A[1..n]$ rooted at $A[i]$ is a max-heap. The rest of A is unchanged.

Comparisons: **at most 2*h***, where h is the height of the subtree. This height is

1 if $\lfloor n/2^2 \rfloor+1 \le i \le \lfloor n/2 \rfloor$,
2 if $\lfloor n/2^3 \rfloor+1 \le i \le \lfloor n/2^2 \rfloor$,
etc.

```
void build-max-heap( T[] A)
    n = A.length;
    for ( i = ⌊n/2⌋, ⌊n/2⌋–1, ..., 1 )
        max-heapify( A, n, i)
```

Initially: A is an arbitrary array.

Upon return: A is a max-heap.

Note: Pass i through the loop makes the subtree of A rooted at $A[i]$ into a max-heap.

Comparisons: **at most 2*n***.

```
void sort-max-heap( T[] A)
    n = A.length;
    for ( i = n, n–1, ..., 2 )
        swap( A[1], A[i]);
        max-heapify( A, i–1, 1);
```

Initially: A is a max-heap.

Upon return: A is a sorted array.

Note: Pass i through the loop moves the i^{th} smallest element to position i, and then rebuilds $A[1..i-1]$ into a max-heap.

Comparisons: **at most 2*n* lg(*n*)**

```
void heapsort( T[] A )
    build-max-heap( A);
    sort-max-heap( A);
```

Initially: A is an arbitrary array.

Upon return: A is sorted.

Comparisons: **at most 2*n* lg(*n*) + O(*n*)**

Recall that our array A implicitly represents a nearly complete binary tree. The *max-heap property* holds at $A[p]$ provided $A[p] \ge A[2p]$ and $A[p] \ge A[2p+1]$ whenever $2p$ and $2p+1$ are within bounds.