Complexity of Heapsort

Let $T(n)$ be the time to run HEAPSORT on an array of size $n$. Examination of the algorithms leads to the following formulation for runtime:

$$T(n) = T_{\text{buildheap}}(n) + \sum_{k=1}^{n-1} T_{\text{heapify}}(k) + \Theta(n - 1) \quad (1)$$

**Heapify.**

Since Heapify is also used in Buildheap, we will attack it first:

$$T_{\text{heapify}}(n) = \Theta(1) + T_{\text{heapify}}(\text{size of subtree}) \quad (2)$$

So we need to know how big the subtrees of a heap with $n$ elements can be. A first guess might be that a subtree can only be half as big as the tree, since a heap is a binary tree. However, a little reflection shows that a better guess is $2/3$:

```
    0
   /|\  \\
  /  |  \ \\
 /   |   \ \\
/     |   \ \\
---------
/         III | IV \\
---------
```

In a complete binary tree, each of the regions shown has approximately the same number of nodes. In a heap, region IV may be empty while region III is full. Since the left subtree consists of regions I and III, it has approximately $2/3$ of the nodes in the heap.

This argument can be formalized. Let $\ell$ be the depth of heap $A$, and let $n$ be the number of nodes in $A$.

$$2^\ell \leq n < 2^{\ell+1}$$

The right subtree of $A$ cannot be larger than the left subtree. A complete tree of height $h$ has $2^{h+1} - 1$ nodes.

Regions I and II in the diagram each have $2^{\ell-1} - 1$ nodes. Region III has $2^{\ell-1}$ nodes, and the size of the left subtree is $2^\ell - 1$. 
Let \( n = 2^\ell + k \), where \( 0 \leq k < 2^\ell \). The left subtree contains \( 2^{\ell-1} + j \) nodes and \( 0 \leq j < 2^{\ell-1} \). If \( i = k - j, 0 \leq i \leq 2^{\ell-1} \), then \( n = 2^\ell + j + i \) and
\[
\frac{\text{size of left subtree}}{n} = \frac{2^{\ell-1} + j}{2^{\ell} + j + i} \leq \frac{2^{\ell-1} + j}{2 \cdot 2^{\ell-1} + j} = f(j).
\]

Let \( \alpha = 2^{\ell-1} \). Then
\[
f(j) = \frac{\alpha + j}{2\alpha + j}.
\]

Extend \( f \) to \([0, \alpha]\) and analyze:
\[
\begin{align*}
f(0) &= \frac{\alpha + 0}{2\alpha + 0} = \frac{1}{2} \\
f(\alpha) &= \frac{\alpha + \alpha}{2\alpha + \alpha} = \frac{2}{3} \\
f'(j) &= \frac{\alpha}{(2\alpha + j)^2} \geq 0
\end{align*}
\]

So \( f \) is an increasing function that takes on a maximum at the left endpoint, and the max is \( 2/3 \).

We state our result as a theorem:

If a heap \( A \) has size \( n \), its subtrees have size less than or equal to \( 2n/3 \).

Applying this theorem to Equation (2), we have
\[
T_{\text{heapify}}(n) = T_{\text{heapify}}(\frac{2n}{3}) + \Theta(1).
\]

Using Case 2 of the Master Theorem, we have
\[
T_{\text{heapify}}(n) = \Theta(\lg n).
\]

**Buildheap.**

The complexity of \textsc{Buildheap} appears to be \( \Theta(n \lg n) - n \) calls to \textsc{Heapify} at a cost of \( \Theta(\lg n) \) per call, but this result can be improved to \( \Theta(n) \). The analysis is in the book. The intuition is that most of the calls to heapify are on very short heaps.

Putting our results together with Equation (1) gives us the following run-time complexity for \textsc{Heap-sort}:
\[
\begin{align*}
T(n) &= T_{\text{buildheap}}(n) + \sum_{k=1}^{n-1} T_{\text{heapify}}(k) + \Theta(n - 1) \\
&= \Theta(n) + \sum_{k=1}^{n-1} \lg k + \Theta(n - 1) \\
&= \Theta(n \lg n)
\end{align*}
\]