Heapsort Algorithm

• We build a max heap out of the given array of numbers $A[1..n]$.

• We repeatedly extract the maximum item from the heap.

• Once the max item is removed, we are left with a hole at the root.

• To fix this, we will replace it with the last leaf in the tree.

• But now the heap order will very likely be destroyed.

• We will apply a heapify procedure to the root to restore the heap.
Heap Sort

HEAPSORT (array A, int n)
1 BUILD-HEAP(A, n)
2 m ← n
3 while (m ≥ 2)
4 do SWAP(A[1], A[m])
5 m ← m − 1
6 HEAPIFY(A, 1, m)
Heapsort Trace
Heapsort Trace
Heapsort Trace
Heapsort Trace

Algorithms – p. 198
## Heapsort Trace

![Heapsort Tree Diagram](image-url)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19</td>
<td>23</td>
<td>12</td>
<td>15</td>
<td>44</td>
<td>57</td>
<td>87</td>
</tr>
<tr>
<td>1</td>
<td>23</td>
<td>19</td>
<td>23</td>
<td>15</td>
<td>44</td>
<td>57</td>
<td>87</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>15</td>
<td>12</td>
<td>15</td>
<td>44</td>
<td>57</td>
<td>87</td>
</tr>
<tr>
<td>3</td>
<td>19</td>
<td>23</td>
<td>12</td>
<td>15</td>
<td>44</td>
<td>57</td>
<td>87</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>23</td>
<td>12</td>
<td>15</td>
<td>44</td>
<td>57</td>
<td>87</td>
</tr>
<tr>
<td>5</td>
<td>19</td>
<td>23</td>
<td>12</td>
<td>15</td>
<td>44</td>
<td>57</td>
<td>87</td>
</tr>
<tr>
<td>6</td>
<td>19</td>
<td>23</td>
<td>12</td>
<td>15</td>
<td>44</td>
<td>57</td>
<td>87</td>
</tr>
<tr>
<td>7</td>
<td>19</td>
<td>23</td>
<td>12</td>
<td>15</td>
<td>44</td>
<td>57</td>
<td>87</td>
</tr>
</tbody>
</table>

Sorted: 19, 23, 44, 12, 15, 57, 87
Heapsort Trace

Sorted
Heapsort Trace

44
23
12
15
19

0 1 2 3 4 5 6 7
44 23 19 12 15 57 87

sorted
Heapsort Trace

15 23 19 12 44

0 1 2 3 4 5 6 7

15 23 19 12 44 57 87

sorted
Heapsort Trace

Algorithms – p. 198
Heapsort Trace

23 → 15 → 19 → 12

sorted
Heapsort Trace

0 1 2 3 4 5 6 7
12 15 19 23 44 57 87

sorted
Heapsort Trace

19
15
12

0 1 2 3 4 5 6 7
19 15 12 23 44 57 87

sorted
Heapsort Trace
Heapsort Trace

0  1  2  3  4  5  6  7

sorted

12 15 19 23 44 57 87
Heapify

HEAPIFY(array A, int i, int n)
  1 l ← LEFT(i)
  2 r ← RIGHT(i)
  3 max ← i
  4 if (l ≤ m) and (A[l] > A[max])
     then max ← l
  5 if (r ≤ m) and (A[r] > A[max])
     then max ← r
  6 if (max ≠ i)
     then SWAP(A[i], A[max])
  7 HEAPIFY(A, max, m)
Analysis of Heapify

- We call heapify on the root of the tree.
- The maximum levels an element could move up is $\Theta(\log n)$ levels.
- At each level, we do simple comparison which $O(1)$.
- The total time for heapify is thus $O(\log n)$.
- Notice that it is not $\Theta(\log n)$ since, for example, if we call heapify on a leaf, it will terminate in $\Theta(1)$ time.
BuildHeap

BUILDHEAP( array A, int n)

1 for i ← n/2 downto 1
2 do
3     HEAPIFY(A, i, n)
Analysis of BuildHeap

- For convenience, we will assume $n = 2^{h+1} - 1$ where $h$ is the height of the tree.
- The heap is a left-complete binary tree.
- Thus at each level $j$, $j < h$, there are $2^j$ nodes in the tree.
- At level $h$, there will be $2^h$ or less nodes.
- How much work does buildHeap carry out?
BuildHeap

Total work performed

3 x 1
2 x 2
1 x 4
0 x 8
Analysis of BuildHeap

- At the bottom most level, there are $2^h$ nodes but we do not heapify these.
- At the next level up, there are $2^{h-1}$ nodes and each might shift down 1.
- In general, at level $j$, there are $2^{h-j}$ nodes and each may shift down $j$ levels.
Analysis of BuildHeap

- So, if count from bottom to top, level-by-level, the total time is

\[ T(n) = \sum_{j=0}^{h} j2^{h-j} = \sum_{j=0}^{h} j \frac{2^h}{2^j} \]

- We can factor out the \(2^h\) term:

\[ T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j} \]
Analysis of BuildHeap

• How do we solve this sum? Recall the geometric series, for any constant $x < 1$

$$\sum_{j=0}^{\infty} x^j = \frac{1}{1 - x}$$

• Take the derivative with respect to $x$ and multiply by $x$

$$\sum_{j=0}^{\infty} jx^{j-1} = \frac{1}{(1 - x)^2}$$
$$\sum_{j=0}^{\infty} jx^j = \frac{x}{(1 - x)^2}$$
Analysis of BuildHeap

- We plug $x = 1/2$ and we have the desired formula:

$$\sum_{j=0}^{\infty} \frac{j}{2^j} = \frac{1/2}{(1 - (1/2))^2} = \frac{1/2}{1/4} = 2$$
Analysis of BuildHeap

- In our case, we have a bounded sum, but since the infinite series is bounded, we can use it as an easy approximation:

\[
T(n) = 2^h \sum_{j=0}^{h} \frac{j}{2^j} \\
\leq 2^h \sum_{j=0}^{\infty} \frac{j}{2^j} \\
\leq 2^h \cdot 2 = 2^{h+1}
\]
Analysis of BuildHeap

• Recall that $n = 2^{h+1} - 1$.
• Therefore

$$T(n) \leq n + 1 \in O(n)$$

• The algorithm takes at least $\Omega(n)$ time since it must access every element at once.
• So the total time for BuildHeap is $\Theta(n)$.
Analysis of BuildHeap

- BuildHeap is a relatively complex algorithm.
- Yet, the analysis yield that it takes $\Theta(n)$ time.
- An intuitive way to describe why it is so is to observe an important fact about binary trees.
- The fact is that the vast majority of the nodes are at the lowest level of the tree.
- For example, in a complete binary tree of height $h$, there is a total of $n \approx 2^{h+1}$ nodes.
Analysis of BuildHeap

- The number of nodes at the bottom three levels alone is

\[ 2^h + 2^{h-1} + 2^{h-2} = \frac{n}{2} + \frac{n}{4} + \frac{n}{8} = \frac{7n}{8} = 0.875n \]

- Almost 90% of the nodes of a complete binary tree reside in the 3 lowest levels.

- Thus, algorithms that operate on trees should be efficient (as BuildHeap is) on the bottom-most levels since that is where most of the weight of the tree resides.
Analysis of Heapsort

- Heapsort calls BuildHeap once. This takes $\Theta(n)$.
- Heapsort then extracts roughly $n$ maximum elements from the heap.
- Each extract requires a constant amount of work (swap) and $O(\log n)$ heapify.
- Heapsort is thus $O(n \log n)$. 
Analysis of Heapsort

- Is HeapSort $\Theta(n \log n)$?
- The answer is yes.
- In fact, later we will show that comparison based sorting algorithms cannot run faster than $\Omega(n \log n)$.
- Heapsort is such an algorithm and so is Mergesort that we saw earlier.