

Worst case for building a heap

The algorithm I presented last lecture for building a heap is relatively inefficient in the worst case. If we index the nodes from $i = 1$ to $i = n$, then if node i is at level l (the root is at level $l = 0$), we can see by inspection (see lecture slides) that

$$2^l \leq i \leq 2^{l+1} - 1$$

and so $\lfloor \log i \rfloor = l$. Thus, when we add node i to the heap, we may need to do $\lfloor \log i \rfloor$ swaps up the tree to bring the new element i to a position where it is greater than its parent, namely we may need to swap it all the way up to the root. Since we are adding n nodes in total, the worst case number of swaps is:

$$t(n) = \sum_{i=1}^n \lfloor \log i \rfloor$$

To visualize this sum, consider the plot below which show the $\log i$ (thick) and $\lfloor \log i \rfloor$ (dashed) curves as a function of i up to $i = 500$ (left) and $i = 5000$ (right). The area under the dashed curve is the above summation. It should be obvious from the figures that

$$\frac{1}{2}n \log n < t(n) < n \log n$$

where the left side of the inequality is the area under the diagonal line from $(0,0)$ to $(n, \log n)$ and $n \log n$ is the area under the rectangle of height $\log n$. It follows that the algorithm for building a heap is $O(n \log n)$.

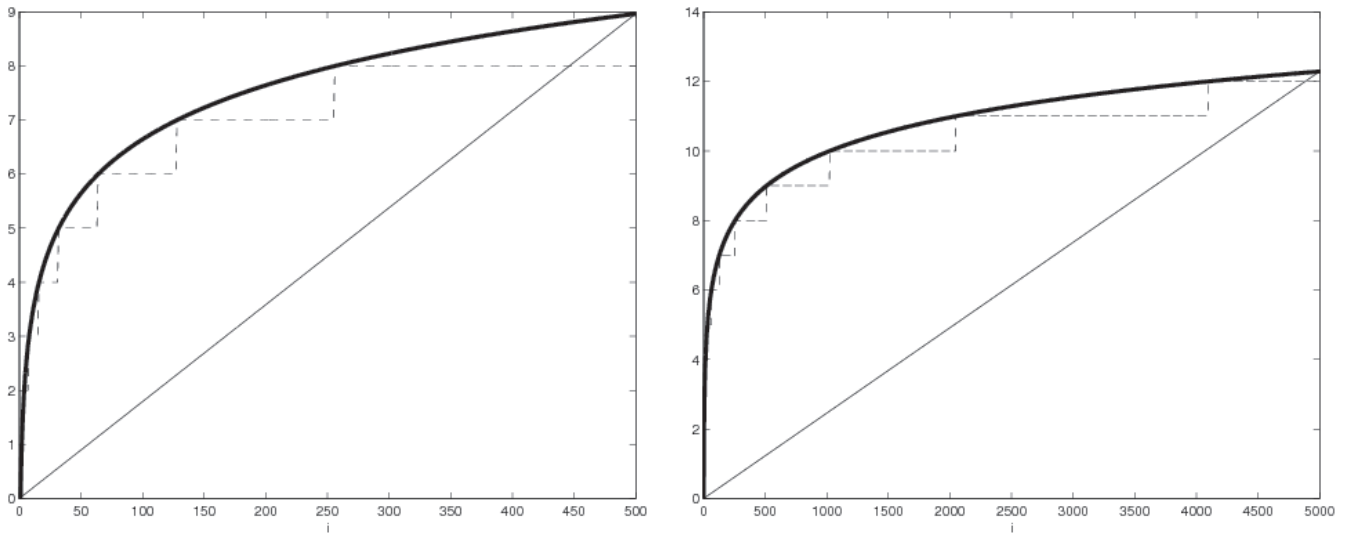
A few notes: first, the figure is not a mathematical proof of the lower bound, but it probably is as convincing to you as any proper proof would be. Second, the steps of the dashed line correspond to the nodes at different levels of the tree. Note that the width of each successive step is double that of the previous one. Moreover, it is not difficult to show that the area under the last full step (corresponding to the last full level of the tree) satisfies $\frac{n}{4}(\log n - 2) < \text{area} < \frac{n}{2}(\log n - 1)$ which is $\Omega(n \log n)$ and $O(n \log n)$.

removeMin and downHeap

Recall the `removeMin()` algorithm from last lecture. We can write this algorithm using the array representation of heaps as follows.

```
removeMin(){
    n = a.size
    element = a[1]      // to be returned
    a[1] = a[n]
    downHeap(1, n-1)
    a.size = a.size - 1
    return element
}
```

Here we make use of a method `downHeap` which swaps a node with the smaller of its children (recursively). Let's define this method.



Suppose this binary tree has the property that the two trees rooted at node i 's two children (if they exist) are each the root of a heap. Then the subtree rooted at i itself would only be a heap if i is less than both its children. The operation `downHeap` addresses this case. It turns the tree rooted at i into a heap, by swapping i with the smaller of its children and then proceeding recursively.

The algorithm below mentions a “pre-condition”. Think of this as a condition that is assumed to be true before the algorithm begins. It also mentions a “post-condition”. This is a condition that is supposed to be true after the algorithm finishes. (In later courses, you will learn more about how to work with pre- and post-conditions.)

Algorithm: `downHeap(i, n)`

"pre-condition": index i such that two subtrees rooted at the children of i (if they exist) are heaps

"post-condition": the subtree rooted at node i is a heap

```

if (2*i <= n){                                // There is a left child
    child = 2*i
    if child < n {                             // There is also a right child
        if a[child] > a[child+1]             // Which is smaller?
            child++
    }
    if a[child] < a[i]{                        // then swap i and child
        swap(i, child)
        downheap(child, n)
    }
}

```

Heapsort

A heap can be used to sort a set of elements. First, we build a heap. Then, we are sure that the minimum element in our array is the root (index $i = 1$ in the array). The idea of heapsort is to remove the minimum element from the heap, replace it by the last element and then downheap that element from the root. (Note the size of the heap is reduced by 1.) We use the freed slot in the array (the last element) to store the removed (smallest) element. After doing this i times, the heap is of size $n - i$ only. Thus we `downHeap` the root to at most index $n - i$. The last i elements in the array stored the smallest i elements in the set.

INPUT: heap a[1 .. n] i.e. minimum is a[1]
 OUTPUT: a sorted array a[1 .. n] (sorted from maximum to minimum)

```
for i = 1 to n{
    swap( a[1], a[n+1 - i])
    downHeap( a, 1, n-i)      // overloaded downHeap (see below)
}
```

Note that the array will be sorted from largest to smallest. Is this a problem? Not at all, since we can reverse the elements in an array in $O(n)$ time, just by swapping i and $n + 1 - i$ for $i = 1$ to $n/2$.

The `heapSort` algorithm is $O(n \log n)$ since we loop n times and at the $i - th$ pass, the i^{th} `downHeap` may require up to $\log(n - i)$ swaps. Thus, in the worst case, the number of swaps is $\sum_{i=0}^{n-1} \log(n - i)$ which is identical to $\sum_{i=0}^{n-1} \log i$ which we saw last class is bounded below by $\frac{n}{2} \log n$.

Example

The example below shows the state of the array after each pass through the for loop. The vertical line marks the boundary between the remaining heap (on the left) and the sorted elements (on the right).

1	2	3	4	5	6	7	8	9		

a	d	b	e	l	u	k	f	w		
b	d	k	e	l	u	w	f	a		(removed a, put w at root, ...)
d	e	k	f	l	u	w	b	a		(removed b, put f at root, ...)
e	f	k	w	l	u	d	b	a		(removed d, put w at root, ...)
f	l	k	w	u	e	d	b	a		(removed e, put u at root, ...)
k	l	u	w	f	e	d	b	a		(removed f, put u at root, ...)
l	w	u	k	f	e	d	b	a		(removed k, put w at root, ...)
u	w	l	k	f	e	d	b	a		(removed l, put u at root, ...)
w	u	l	k	f	e	d	b	a		(removed u, put w at root, ...)
w	u	l	k	f	e	d	b	a		(removed w, and done)