**Question**:

To build a MAX heap tree, we can either siftDown or siftUp, by sifting down we start from the root and compare it to its two children, then we replace it with the larger element of the two children, if both children are smaller then we stop, otherwise we continue sifting that element down until we reach a leaf node (or of course again, until that element is larger that both of its children).

Now we will only need to do that n/2 times, because the number of leaves is n/2, and the leaves will satisfy the heap property when we finish heapifying the last element on the level before the last (before the leaves) - so we will be left with n/2 elements to heapify.

Now if we use siftUp, we will start with the leaves, and eventually we will need to heapify all n elements.

My question is: when we use siftDown, aren't we basically doing two comparisons (comparing the element to its both children), instead of only one comparison when using siftUp, since we only compare that element to its one parent? If yes, wouldn't that mean that we're doubling the complexity and really ending up with the same exact complexity as sifting down?

**Answer**:

Actually, building a heap with repeated calls of siftDown has a complexity of O(n) whereas building it with repeated calls of siftUp has a complexity of O(nlogn).

This is due to the fact that when you use siftDown, the time taken by each call **decreases** with the depth of the node because these nodes are closer to the leaves. When you use siftUp, the number of swaps **increases** with the depth of the node because if you are at full depth, you may have to swap all the way to the root. As the number of nodes grows exponentially with the depth of the tree, using siftUp gives a more expensive algorithm.

Moreover, if you are using a Max-heap to do some sort of sorting where you pop the max element of the heap and then reheapify it, it's easier to do so by using siftDown. You can reheapify in O(logn)time by popping the max element, putting the last element at the root node (which was empty because you popped it) and then sifting it down all the way back to its correct spot.