I think there are several questions buried in this topic:

- How do you implement buildHeap so it runs in *O(n)* time?
- How do you show that buildHeap runs in *O(n)*time when implemented correctly?
- Why doesn't that same logic work to make heap sort run in *O(n)* time rather than *O(n log n)*?

Often, answers to these questions focus on the difference between `siftUp` and `siftDown`. Making the correct choice between `siftUp` and `siftDown` is critical to get *O(n)* performance for `buildHeap`, but does nothing to help one understand the difference between `buildHeap` and `heapSort` in general. Indeed, proper implementations of both `buildHeap` and `heapSort` will **only** use `siftDown`. The `siftUp` operation is only needed to perform inserts into an existing heap, so it would be used to implement a priority queue using a binary heap, for example.

I've written this to describe how a max heap works. This is the type of heap typically used for heap sort or for a priority queue where higher values indicate higher priority. A min heap is also useful; for example, when retrieving items with integer keys in ascending order or strings in alphabetical order. The principles are exactly the same; simply switch the sort order.

The **heap property** specifies that each node in a binary heap must be at least as large as both of its children. In particular, this implies that the largest item in the heap is at the root. Sifting down and sifting up are essentially the same operation in opposite directions: move an offending node until it satisfies the heap property:

- `siftDown` swaps a node that is too small with its largest child (thereby moving it down) until it is at least as large as both nodes below it.
- `siftUp` swaps a node that is too large with its parent (thereby moving it up) until it is no larger than the node above it.

The number of operations required for `siftDown` and `siftUp` is proportional to the distance the node may have to move. For `siftDown`, it is the distance from the bottom of the tree, so `siftDown` is expensive for nodes at the top of the tree. With `siftUp`, the work is proportional to the distance from the top of the tree, so `siftUp` is expensive for nodes at the bottom of the tree. Although both operations are *O(log n)* in the worst case, in a heap, only one node is at the top whereas half the nodes lie in the bottom layer. So **it shouldn't be too surprising that if we have to apply an operation to every node, we would prefer `siftDown` over `siftUp`.**

The `buildHeap` function takes an array of unsorted items and moves them until they all satisfy the heap property, thereby producing a valid heap. There are two approaches one might take for `buildHeap` using the `siftUp` and `siftDown` operations we've described.

1. Start at the top of the heap (the beginning of the array) and call `siftUp` on each item. At each step, the previously sifted items (the items before the current item in the array) form a valid heap, and sifting the next item up places it into a valid position in the heap. After sifting up each node, all items satisfy the heap property.
2. Or, go in the opposite direction: start at the end of the array and move backwards towards the front. At each iteration, you sift an item down until it is in the correct location.

Both of these solutions will produce a valid heap. The question is: which implementation for `buildHeap` is more efficient? Unsurprisingly, it is the second operation that uses `siftDown`.

Let *h* = *log n* represent the height of the heap. The work required for the `siftDown` approach is given by the sum

```
(0 * n/2) + (1 * n/4) + (2 * n/8) + ... + (h * 1).
```

Each term in the sum has the maximum distance a node at the given height will have to move (zero for the bottom layer, h for the root) multiplied by the number of nodes at that height. In contrast, the sum for calling `siftUp` on each node is

```
(h * n/2) + ((h-1) * n/4) + ((h-2)*n/8) + ... + (0 * 1).
```

It should be clear that the second sum is larger. The first term alone is *hn/2 = 1/2 n log n*, so this approach has complexity at best *O(n log n)*. But how do we prove that the sum for the `siftDown` approach is indeed *O(n)*? One method (there are other analyses that also work) is to turn the finite sum into an infinite series and then use Taylor series. We may ignore the first term, which is zero:

$$
\begin{aligned}
1 \cdot \frac{n}{4} + 2 \cdot \frac{n}{8} + \ldots + (h \cdot 1) &= \sum_{k=1}^{h} \frac{kn}{2^{k+1}} = \frac{n}{4} \sum_{k=1}^{h} \frac{k}{2^{k-1}} \\
&< \frac{n}{4} \sum_{k=1}^{\infty} \frac{k}{2^{k-1}} = \frac{n}{4} \sum_{k=1}^{\infty} k x^{k-1}, \qquad x = \frac{1}{2} \\
&= \frac{n}{4} \frac{d}{dx} \left[\sum_{k=0}^{\infty} x^k \right] = \frac{n}{4} \frac{d}{dx} \left[\frac{1}{1-x} \right] \\
&= \frac{n}{4} \frac{1}{(1-x)^2} = \frac{n}{4} \frac{1}{(1-1/2)^2)} = n.
\end{aligned}
$$

If you aren't sure why each of those steps works, here is a justification for the process in words:

- The terms are all positive, so the finite sum must be smaller than the infinite sum.
- The series is equal to a power series evaluated at *x=1/2*.
- That power series is equal to (a constant times) the derivative of the Taylor series for *f(x)=1/(1-x)*.
- *x=1/2* is within the interval of convergence of that Taylor series.
- Therefore, we can replace the Taylor series with *1/(1-x)*, differentiate, and evaluate to find the value of the infinite series.

Since the infinite sum is exactly *n*, we conclude that the finite sum is no larger, and is therefore, *O(n)*.

The next question is: if it is possible to run `buildHeap` in linear time, why does heap sort require *O(n log n)* time? Well, heap sort consists of two stages. First, we call `buildHeap` on the array, which requires *O(n)* time if implemented optimally. The next stage is to repeatedly delete the largest item in the heap and put it at the end of the array. Because we delete an item from the heap, there is always an open spot just after the end of the heap where we can store the item. So heap sort achieves a sorted order by successively removing the next largest item and putting it into the array starting at the last position and moving towards the front. It is the complexity of this last part that dominates in heap sort. The loop looks likes this:

```
for (i = n - 1; i > 0; i--) {
    arr[i] = deleteMax();
}
```

Clearly, the loop runs O(n) times (*n - 1* to be precise, the last item is already in place). The complexity of `deleteMax` for a heap is *O(log n)*. It is typically implemented by removing the root (the largest item left in the heap) and replacing it with the last item in the heap, which is a leaf, and therefore one of the smallest items. This new root will almost certainly violate the heap property, so you have to call `siftDown` until you move it back into an acceptable position. This also has the effect of moving the next largest item up to the root. Notice that, in contrast to `buildHeap` where for most of the nodes we are calling `siftDown` from the

bottom of the tree, we are now calling `siftDown` from the top of the tree on each iteration! *Although the tree is shrinking, it doesn't shrink fast enough*: The height of the tree stays constant until you have removed the first half of the nodes (when you clear out the bottom layer completely). Then for the next quarter, the height is *h - 1*. So the total work for this second stage is

```
h*n/2 + (h-1)*n/4 + ... + 0 * 1.
```

Notice the switch: now the zero work case corresponds to a single node and the *h* work case corresponds to half the nodes. This sum is *O(n log n)* just like the inefficient version of `buildHeap` that is implemented using siftUp. But in this case, we have no choice since we are trying to sort and we require the next largest item be removed next.

In summary, the work for heap sort is the sum of the two stages: *O(n) time for buildHeap and* ***O(n log n) to remove each node in order****, so the complexity is O(n log n)*. You can prove (using some ideas from information theory) that for a comparison-based sort, *O(n log n)* is the best you could hope for anyway, so there's no reason to be disappointed by this or expect heap sort to achieve the O(n) time bound that `buildHeap` does.