Heapsort

1 Heapsort

1.1 Algorithm

Assuming a max-heap is used.

```
def HeapSort(A):
    heapify(A)
    return [h.extract_max(0 while len(h) > 0].reverse()
```

1.2 Analysis

In the worst case, when every element is sifted down to the bottom of the heap, per sift-down operation after extracting the top of the heap, there are $2\log n \pm 1$ comparisons done because sift-down requires finding the greater child node. Therefore, a good bound for the number of comparisons done in the entire sort is $2n \log n \pm n$.

$$\sum_{i=1}^{n} 2\lg i \pm 1 = 2\sum i = 1^n \pm n \tag{1}$$

$$= 2\lg \prod_{i=1}^{n} i \pm n \tag{2}$$

$$= 2\lg n! \pm n \tag{3}$$

Apply **Stirling's Formula** to approximate $n!$:

$$n! \approx \left(\frac{n}{e}\right)^m \cdot \sqrt{2\pi n} \tag{4}$$

$$2\lg n! \pm n \approx 2\lg \left[\left(\frac{n}{e}\right)^n \cdot \sqrt{2\pi n}\right] \pm n \tag{5}$$

$$= 2(\lg n^n - \lg e^n + \lg \sqrt{2\pi n}) \pm n \tag{6}$$

$$= 2n\lg n + \Theta(n) \tag{7}$$

The original estimate was very close to the actual computation for two reasons:

1. The size of heap levels grows exponentially, so half of all elements are in the bottom level. This means that there's a 50% chance sift-down after rotating the last element to the head puts it back on the bottom layer.

2. All elements on the bottom level of the heap share the property that they are all relatively small, so it's very likely that the swapped-up element belongs in the bottom after sift-down anyways.

Therefore, given that heapify is $\Theta(n)$, the final comparison-based runtime of heapsort is $2n \lg n + \Theta(n)$, or $O(n \lg n)$.

2 Floyd Siftdown

R.W. Floyd proposed an improved version of siftdown that turns heapsort into an $n \lg n$ algorithm instead of $2n \lg n$. Because sifting an element down after an extraction will likely put it back in the bottom level of the heap or close to it, Floyd's siftdown bubbles the element all the way to the bottom, using 1 comparison per level, and then bubbles the element back up to its proper level, which again uses 1 comparison per level.

Given that a random element from the bottom level is swapped up to the head after each extraction, the chance that it belongs in level i is $(1/2)^i$ due to the way heap levels grow exponentially. So the expected number of siftup operations that will be done after putting the element on the bottom is

$$\sum_{i=1}^{n} i \frac{1}{2^i} \tag{8}$$

$$= \frac{1}{2}(1) + \frac{1}{4}(2) + \frac{1}{8}(3) + \dots \tag{9}$$

$$\approx 2 \tag{10}$$

So there are $\lg(n-1)+2$ comparisons done per siftdown operation, which brings the runtime for heapsort to $n \lg n + \Theta(n)$.

Improved Siftup In Floyd's version of siftdown, the path that the top element takes to the bottom is itself ordered from greatest to least. If this path is memo-ized, a binary search through the path will reveal where to sift the element up to in $\lg \lg n$ comparisons.