```
BEGIN
    BuildMaxHeap(A)
    for i = length[A] downto 2
       swap A[1] with A[i]
       heap-size[A] = heap-size[A] - 1
       MaxHeapify(A, 1)
    end for
END

MaxHeapify(A, i)
    l = left(i)
    r = right(i)
    if l <= heap-size[A] and A[l] > A[i]
       then largest = l
       else largest = i
    if r <= heap-size[A] and A[r] > A[largest]
        then largest = r
    if largest != I
        then swap A[i] with A[largest]
        MaxHeapify(A, largest)
END
```

```
BuildMaxHeap(A)
    heap-size[A] = length[A]
    for i = [length[A]/2] downto 1
        do MaxHeapify(A, i)
    end for
END
```

FIGURE 6: HEAP SORT

Now, let's find out the time complexity of a heap sort algorithm.

The performance of the overall heap sort is determined by analysing the time complexities of the two primary operations involved in algorithm – building the max heap and the swapping operations.

The height of a complete binary tree with 'n' elements is log(n). The worst-case scenario for making a max heap occurs when we need to move an element from the root to the leaf node. This involves performing multiple log(n) number of comparisons and swaps across $^{n}/_{2}$ elements.

So, the complexity of the first phase is $\frac{n}{2}$*log(n) ~ nlog(n)

During the second phase, the root element is swapped with the last element and the max heap is created again. Since this is performed over 'n' elements, the worst-case scenario complexity for this phase is also of the order of nlog(n).

Combining the two complexities using the rules of the Big-Oh notation discussed in the previous sections, we determine that the worst-case complexity of a Heap Sort Algorithm is *O* **(nlogn)**

‹ qsort() Library Function Demo (/?q=MULNCourseBook/listDetailedCBLearningContents/178099/cidb/full/view/153784/431550/431535/39731)