```
function heapSort(Array arr) {
    copy the arr into a new array heap with the 0 index empty
    int firstIndex = 1   // assume the fisrt index is 1 for easier calculation      k0
    int lastIndex = arr.length - 1

    // do reheap() on every non-leaf nodes. The last leaf is at lastIndex, so the last non-leaf is at lastIndex / 2
    for(int rootIndex = lastIndex / 2; rootIndex > 0; rootIndex--)      O(n/2) = O(n)
     reheap(arr, rootIndex);           O(log(n)), see below
}
```

Conclusion:
Worst case complexity -> O(n * log(n)) = O(nlogn)

```
function reheap(Array heap, rootIndex) {
    boolean done = false;
 T orphan = heap[rootIndex];  // start from root         k2
 int leftChildIndex = rootIndex * 2;

 while(!done && (leftChildIndex <= lastIndex)) {
     int largerChildIndex = leftChildIndex;   // assume left is larger          k3
     int rightChildIndex = leftChildIndex + 1;

     if((rightChildIndex <= lastIndex) && heap[rightChildIndex].compareTo(heap[largerChildIndex]) > 0)       k4
      largerChildIndex = rightChildIndex;   // get the larger child

     if(orphan.compareTo(heap[largerChildIndex]) < 0)
     {
      heap[rootIndex] = heap[largerChildIndex];
      rootIndex = largerChildIndex;      // swap root and larger, and repeat the process      k5
      leftChildIndex = rootIndex * 2;
      }

     else
      done = true;    // end the loop         k6
     }

 heap[rootIndex] = orphan; // put this last piece onto the root       k7
    }
}
```

In the worst case:
It will have maximum number of swaps
The second if statment in the while loop " leftChildIndex = rootIndex * 2" will be called again and again until leftChildIndex >= lastIndex
Since every time the leftChildIndex is doubled, it will reach the lastIndex in log(n) times.

O(k2 + log(n) * (k3 + k4 + k5 + k6 + k7)) = O(log(n))