Heapsort Big-O Analysis:

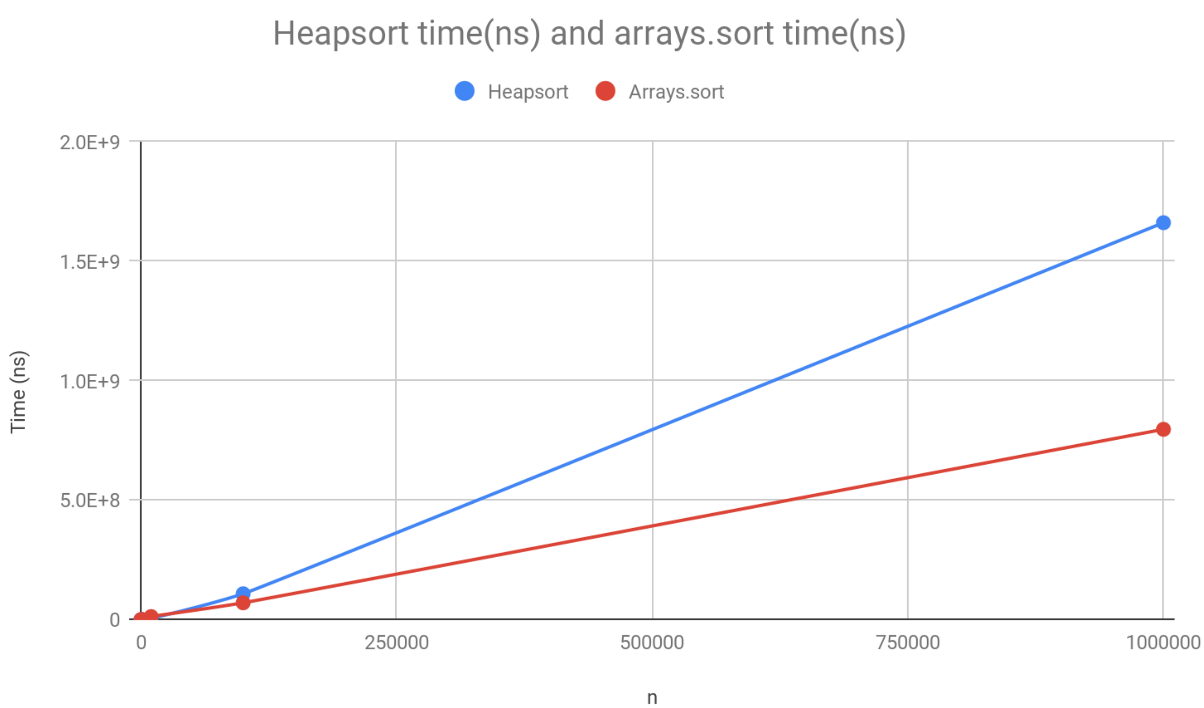
Methods:

- **Swap** - Time: O(1) - the swap method simple swaps two elements within the array of people, fairly self explanatory.
- **siftDown** - Time: O(logn) - the sift down method performs the sifting operation of the heap. Worst case, this method will visit log(n) nodes, one node from each level of the heap. Since the heap is a complete tree, we won't run into bad scenarios with severely unbalanced heaps. All other operations within this method are O(1).
- **Heapify** - Time: O(nlogn) - this method calls the siftDown method (n + 2) / 2 times (simplifies to n times), so multiplying the complexity of siftDown by n, we get nlogn.
- **heapSort** - Time: O(nlogn) - this method calls heapify once (nlogn) and then calls swap and sift down n times within a loop. The loop will have a runtime of n * (logn + 1) which simplifies to nlogn. The total runtime will be the complexity of heapify plus nlogn, thus O(nlogn) + O(nlogn) = O(2nlogn) = O(nlogn).

Overall Worst-case Runtime: O(nlogn)

Results:

n	Heapsort time(ns)	arrays.sort time(ns)
100	60053	124499
1000	467656	671154
10000	5343805	12855649
100000	108290634	70444068
1000000	1660660978	796239590



Comments:

Both sorts appear to be O(nlogn), although the Arrays.sort method seems to be significantly more efficient than my implementation of heapsort.