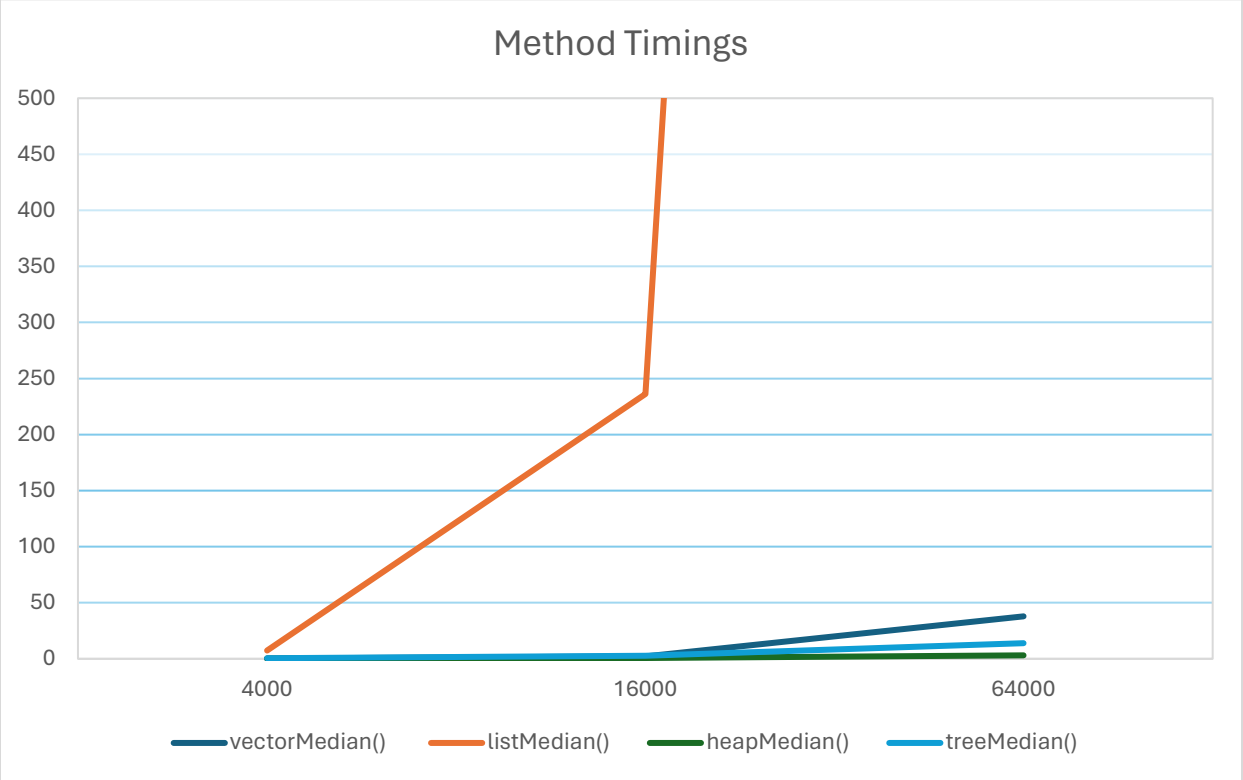
Rio Simpson
Report

Timings of vectorMedian, listMedian, heapMedian, and treeMedian main loop for input size of 4000, 16000, and 64000 instructions. *Results are in milliseconds*

| Input Size | Methods and Their Time in Milliseconds | | | |
|---|---|---|---|---|
| | vectorMedian() | listMedian() | heapMedian() | treeMedian() |
| 4000 Instructions | 0.29 | 7.37 | 0.17 | 0.52 |
| 16000 Instructions | 2.16 | 236.3 | 0.76 | 2.62 |
| 64000 Instructions | 37.86 | 5646.16 | 3.08 | 13.92 |




| | vectorMedian() | listMedian() | heapMedian() | treeMedian() |
|---|---|---|---|---|
| O-Notation | $O(n^2)$ | $O(n^2)$ | O(n log n) | O(n log n) |

Since in vectors, we are given random access to the container, searching through the data structure and manipulating it (either by inserting or removing), we average out to about a time complexity of $O(n^2)$. The results of the function times shows that as the input size grows, so does the time it takes for the function to execute, to a proportional extent. Specifically, since our two main operations are insert and pop, insert will at worst take O(n) or the size of the vector, and pop will take O(n) since we can index to its location but will have to shift the latter half of the vector back. The time complexity is $O(n^2)$ since pop take O(n) for n instructions. As for lists, the cost of each operation averages out to be $O(n^2)$. This is because for each insert operation, we must go through the list to find its appropriate location, meaning a time complexity of at most O(n); and to pop we must also go through the list to find its location (although limited since we know the median is somewhere in the middle) – this gives us an average time complexity of O(n/2) -- which translates to O(n). As with vectors, list methods take O(n) for n instructions, meaning the time complexity is $O(n^2)$

For trees and heaps, the time complexity difference is a bit more blurred. For insert operations, the time complexity is fairly small, although each insert requires some amount of rebalancing, swapping, and other methods to maintain the structure of the container as we specified. For the pop and insert methods, operations to find a specific value, rebalance the data structures, and remove said value has a time complexity of O(log n). since heaps and trees take O(log n) to perform n instructions, the time complexity is O(n log n).

Observations:

- Despite logical analysis of the methods of vectors, lists, heaps, and trees, the data shows stoic differences between vectors and lists, and nuanced differences between heaps and trees. This is certainly because there is not enough data to draw out a more precise picture; though, there is some logical reasoning to suggest why vectors are slightly better than lists, and why heaps (at least in our implementation of a max heap and min heap) are slightly better than trees. I believe that because of vectors ability to randomly access memory, it reduces the total speed required for the pop method – not to mention that the nature of our program, focusing on the median, will in the end contribute to a O(n/2) time complexity. As for Heaps, because of the way that it is structured, it appears that finding data contributes to a faster insert and pop method.