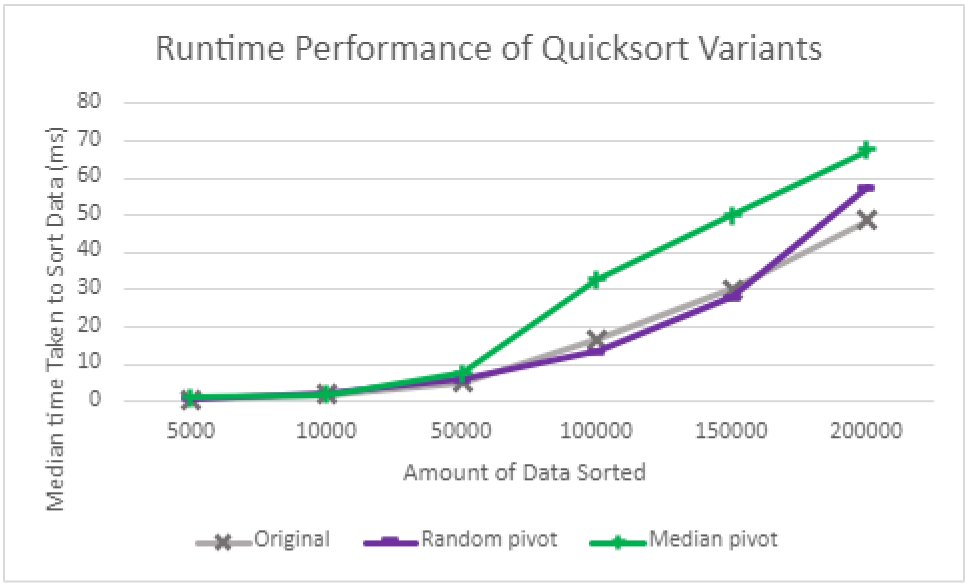
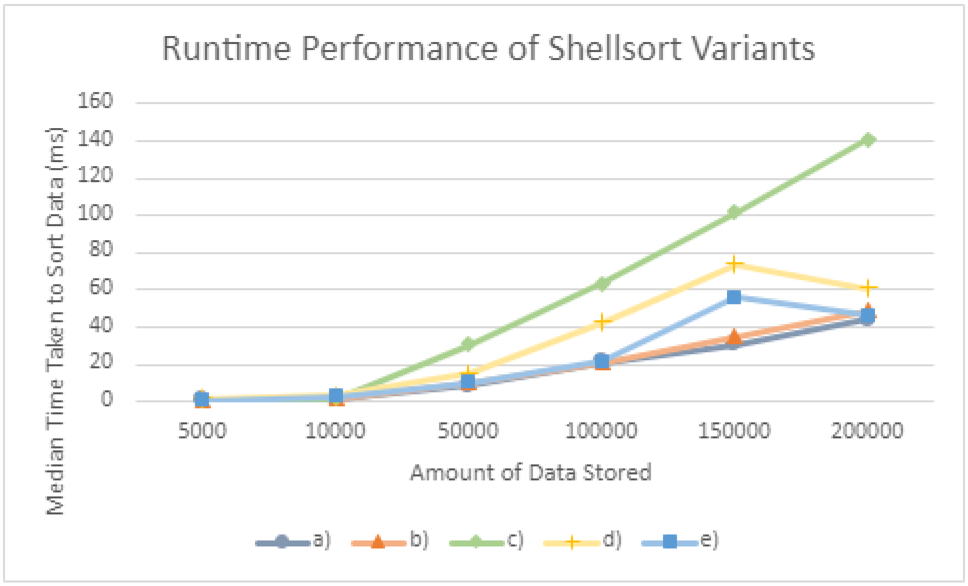


Based on the graph and the table, it can be seen that the original technique has the fastest runtime for most array sizes, that only changes when the sizes are 100,000 and 150,000. During which the random pivot technique performs slightly better than the original. But other than that, it remains quite stable. The median pivot technique, on the other hand, has the slowest runtime for all array sizes, with its runtime increasing significantly as the array size increases. It may be slower than the other two variants because it requires extra computation to find the median pivot element for each partition. This involves finding the median value of the first, middle, and last elements of the partition, which takes more time than simply choosing a random pivot or always using the first or last element as the pivot.

Overall, based on the runtime performance, I would recommend using the original pivot technique due to its pretty consistent fast runtime across the array sizes. Though the random pivot technique would not be a bad choice either. They are not very far from each other in the data so they can both be a reliable choice for datasets of varying sizes. The median pivot technique may be useful in certain cases where data is already partially sorted or close to being sorted, but its significantly slower runtime makes it less practical for most use cases.



a) This version of shellsort has the best runtime performance among the five variants for smaller arrays, with a runtime of less than 1 millisecond for array sizes up to 50000. However, it starts to become slower as the array size increases and takes around 44 milliseconds for an array of size 200000.

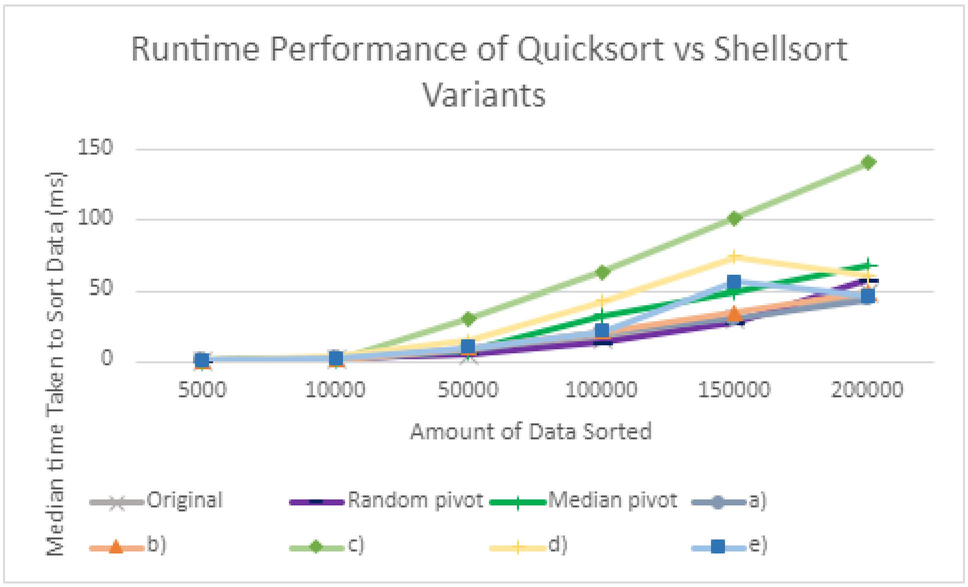
b) This variant has a relatively consistent runtime performance across different array sizes, taking around 2-4 milliseconds for array sizes up to 100000 and around 74 milliseconds for an array of size 200000. However, it is generally slower than variant (a) for smaller arrays.

c) This variant has a similar runtime performance to variant (b) for smaller arrays, but starts to become significantly slower for larger arrays. It takes around 140 milliseconds for an array of size 200000, which is the slowest among the five variants. Variant c) uses a sequence of increments that only includes odd numbers, which can result in more iterations compared to the other variants. This can make it slower, especially for larger input sizes, as it may take longer to fully sort the array. Additionally, the fact that the largest increment is not guaranteed to be 1 can also make this variant slower, as the final pass using a gap of 1 is where most of the sorting work is done.

d) This variant has a similar runtime performance to variant (a) for smaller arrays, but becomes significantly slower for larger arrays. It takes around 74 milliseconds for an array of size 200000, which is slower than variant (b) but faster than variant (c).

e) This variant has a similar runtime performance to variant (a) for smaller arrays, but becomes slower for larger arrays. It takes around 46 milliseconds for an array of size 200000, which is faster than variant (c) but slower than variants (a), (b), and (d).

Based on the above analysis, I would recommend using variant (a) for smaller arrays (up to 50000) and variant (b) for larger arrays (more than 50000). Variant (a) has the best runtime performance among the five variants for smaller arrays, while variant (b) has a relatively consistent runtime performance across different array sizes and is generally faster than the other variants for larger arrays.



In general, the performance of the quicksort variants tends to be better than that of the shellsort variants. Looking at the data in the graph, the quicksort lines tend to stay low and only go as high as about 67.596, during median sort. At larger array sizes(100,000+), it does a lot better than the shellsort variants, especially variant c. So for quicksort, the difference in performance between its techniques becomes more pronounced as the size of the array increases. While for shellsort, the performance of the variants can vary depending on the increment sequence used.

This is perhaps because quicksort has an average time complexity of O(n log n), while the shellsort variants have an average time complexity of O(n^1.25) or worse. These respective complexities may account for the observed difference in performance between them. Since quicksort has an overall better time complexity due to its recursive nature and ability to exploit locality of reference. Shellsort, on the other hand, has a simpler algorithmic structure but relies heavily on the choice of increment sequence to achieve good performance. If one had to choose between the algorithms then, quicksort would most likely be the better choice due to its faster runtime.

Bonus Question:

Shell sort works well with insertion sort but not with selection sort because of the nature of the two algorithms, since insertion sort is compatible with the techniques used by shell sort and selection sort is not. Insertion sort involves placing each element in its proper place within a partially sorted array.The efficiency of insertion sort reduces as the size of the array grows because more comparisons and shifts are needed. But, shell sort can lessen the amount of comparisons and shifts by utilizing the array's partially sorted nature. Initially sorting the elements at a distance, and then gradually reducing the distance until the entire array has been sorted. In doing so, effectively making insertion sort more efficient. Selection sort works by repeatedly finding the smallest element in the unsorted portion of the array and swapping it with the first element until the entire array is sorted. Unlike insertion sort, it has a fixed number of comparisons for each element. Consequently, it is unaffected by shell sort's use of the partially sorted nature of the array. As a result, shell sort's performance won't be enhanced by employing selection sort as its internal sorting algorithm.