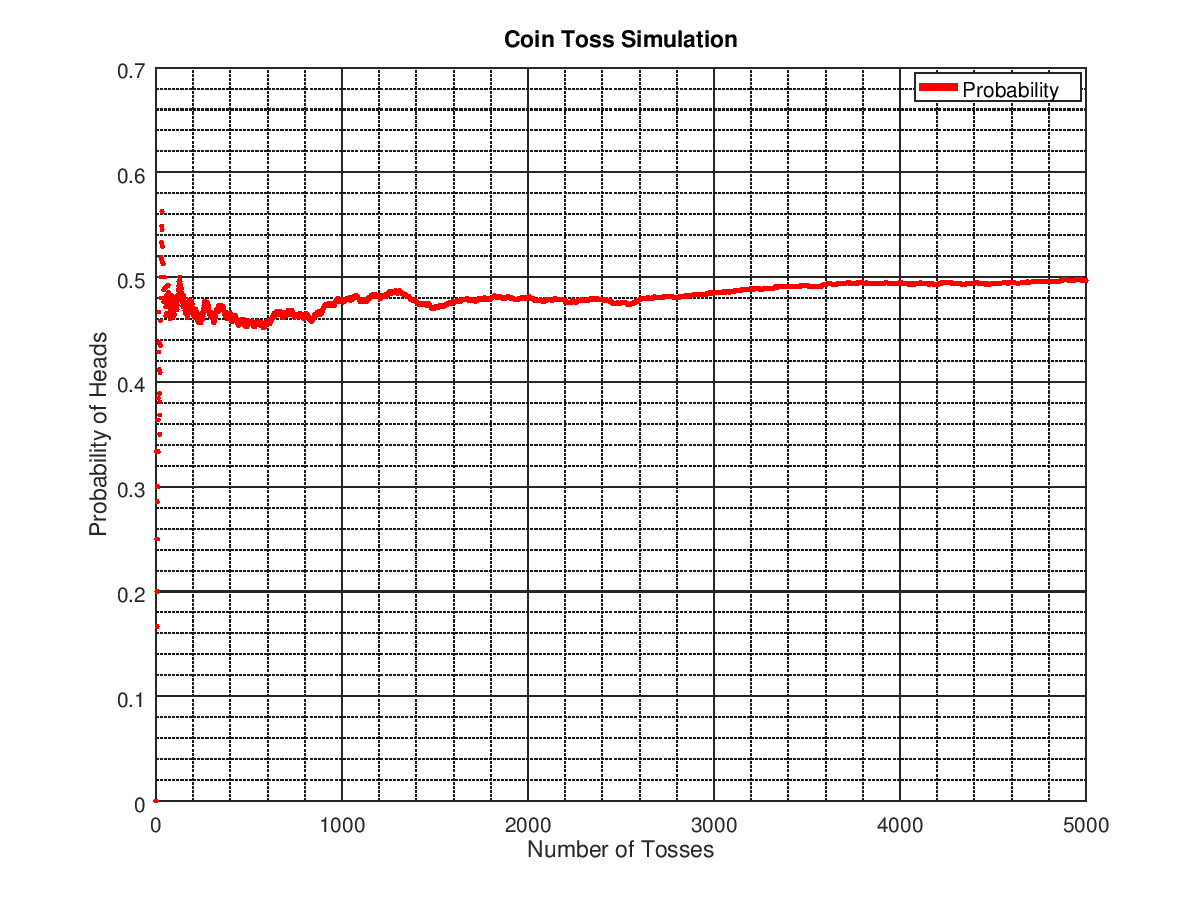
**Question 1**:Coin Toss Simulation

Through a coin toss simulation, show that probability of getting HEAD, by tossing a fair coin, is about 0.5. Write your observation from the simulation run.

**Solution**:

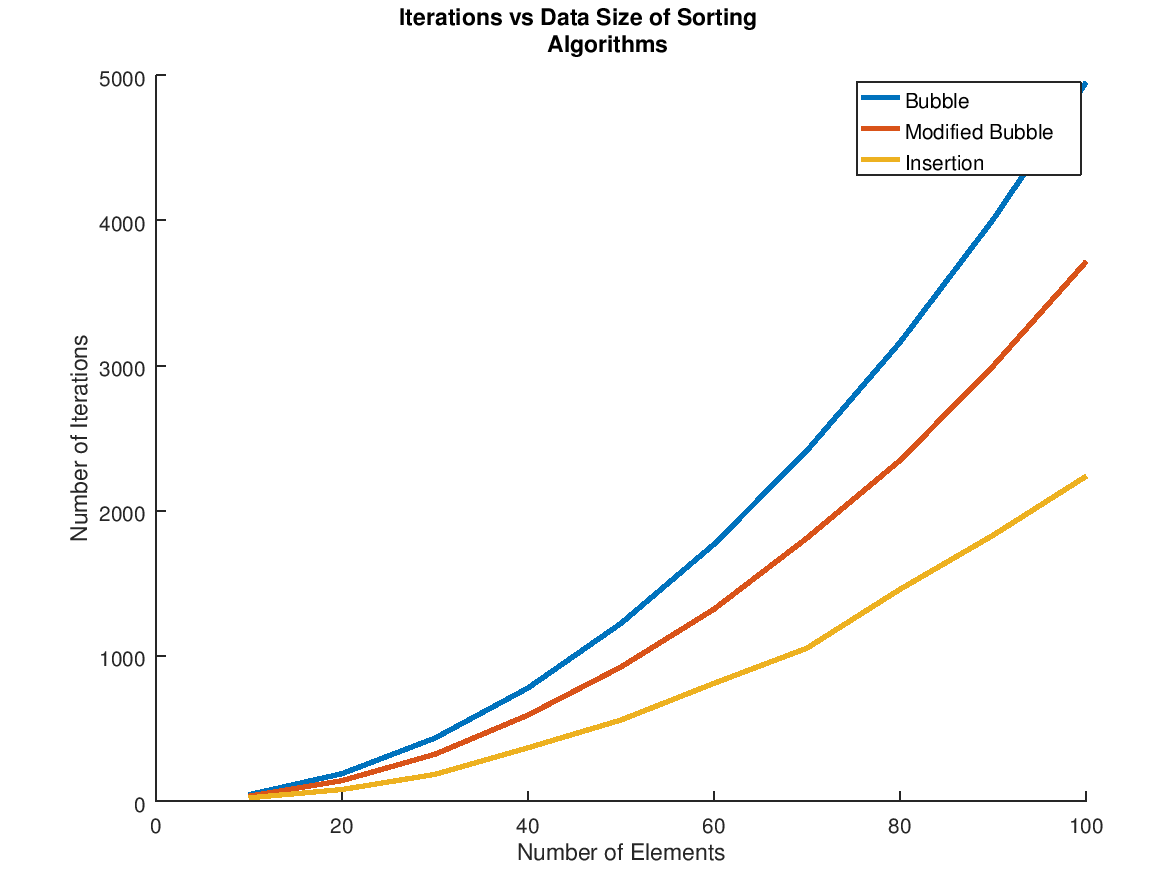
Figure 1: Probablity of heads in a coin toss simulation

Here a function was used to generate random numbers between 0 and 1 based on the normal distribution, and then was rounded to 0 for tails and 1 for heads. The number of heads vs the total number of tosses was plotted which gradually converged towards y=0.5 as the number of tosses increased. Eventually, at ∞, the probability converges to 0.5. At the end of the above simulation run, the probability was 0.49710.

**Question 2**: Bubble Sort Analysis

Implement two different versions of bubble sort for a randomized data sequence.

**Solution:** Alongside the classical bubble sort algorithm, a modified version was implemented where the loop terminates before the n-1th iteration. As the array is often sorted before the final pass, this proved to reduce the number of iterations required, significantly. However, it still remained an O(n2) algorithm as shown below in Figure 2. For sake of comparison, insertion sort had also been implemented.

Figure 2: Asymptotic analysis of bubble and modified bubble sort

**Question 3**: Internal Sorting Analysis

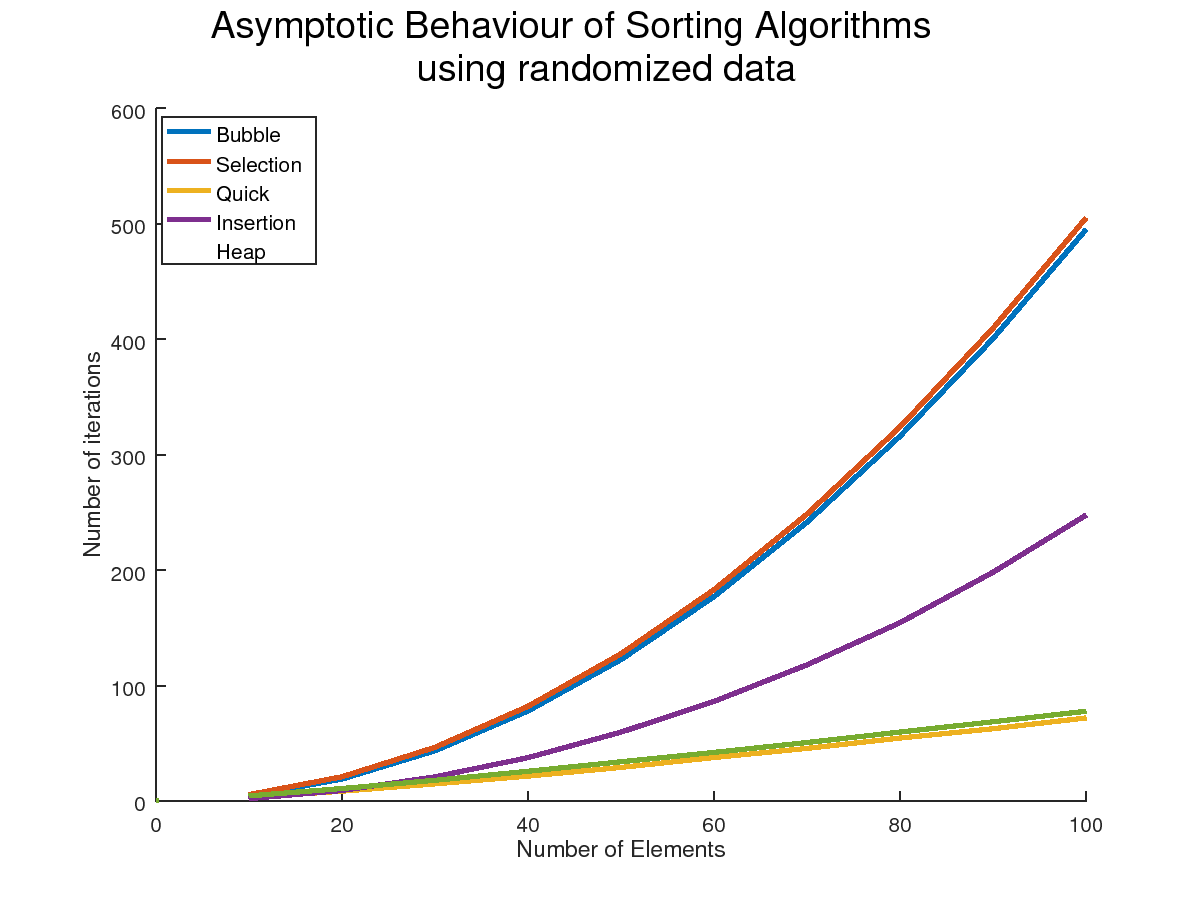
Determine the asymptotic behaviour of at least 5 internal sorting algorithms for each of the following input variations:

1. Random Data
2. Reverse Ordered Data
3. Almost Sorted Data
4. Highly Repetitive Data

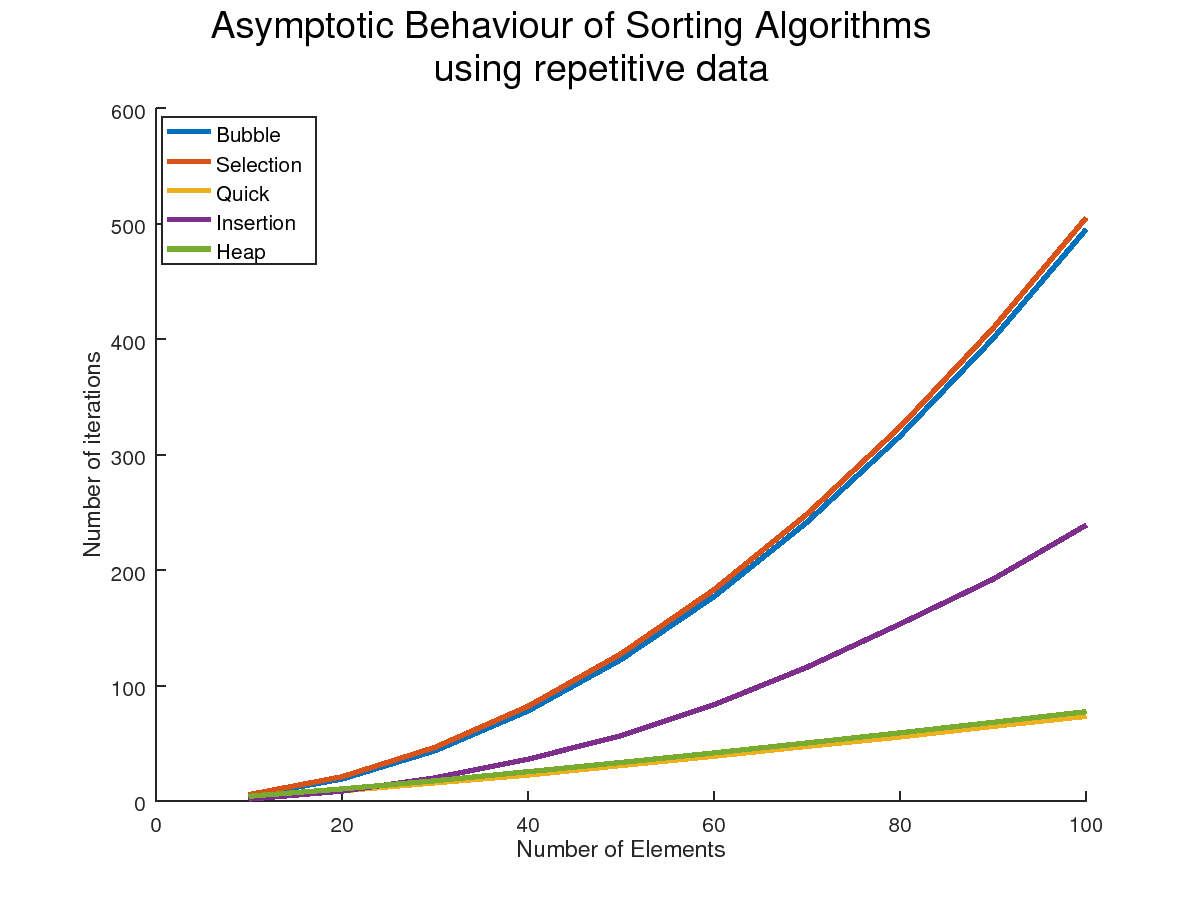
Select a suitable number of elements for the analysis that supports your program.

**Solution**: For analysis, bubble, selection, insertion, quick and heap sorting methods had been implemented.

1. Clearly, we observe that selection sort took the most iterations, followed by bubble, insertion, quick and heap sorting methods. (Figure 3)

Figure 3: Sorting analysis for randomized data

2. Repetitive data was generated by limiting the random number generation range to a reasonably small limit. The data was then fed through the sorting algorithms and they all produced nearly identical results to the previous question. (Figure 4)

Figure 4: Sorting analysis for repetitive data

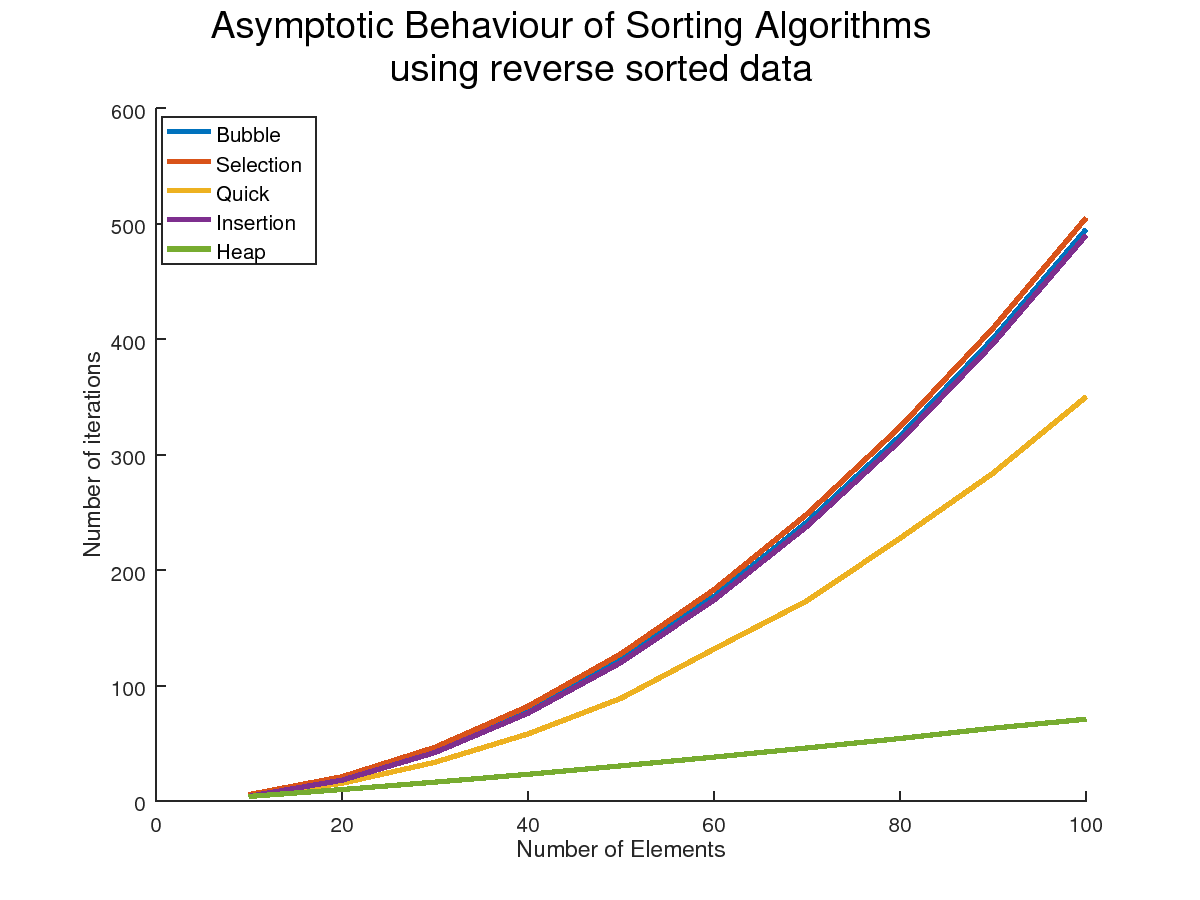
As an example for how this data input differs from Case 1, the following input has been included as a snippet.

3 9 2 4 3 1 4 6 6 3

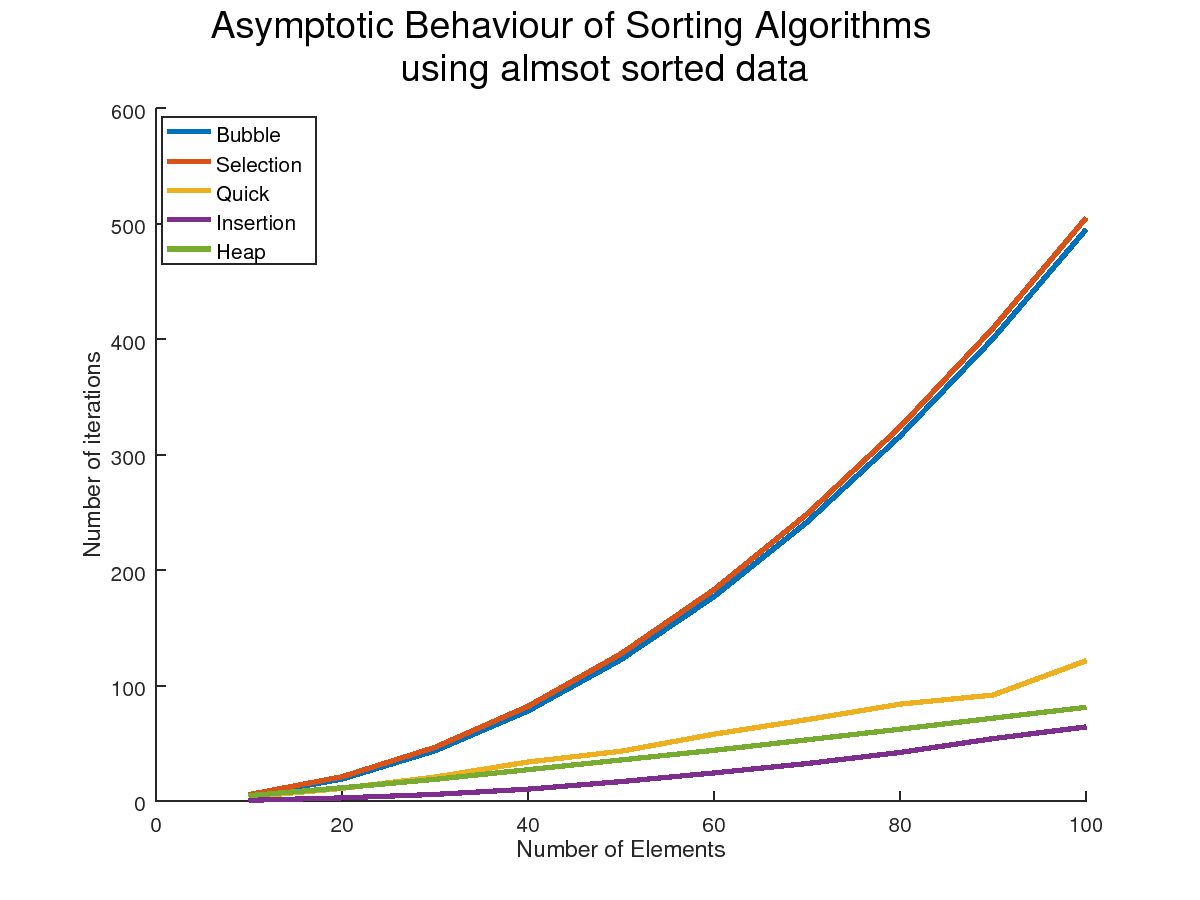
2 1 1 4 2 3 4 4 2 3

A simple modification has been made in the range of the rand function. As the sample size is small for this case, it is not a complete reflection of the entropy of the complete data input.

3. When reverse sorted data was used as input to the sorting algorithms, they all approached their worst-case complexity, which is evidenced by insertion sort being O(n2). Quick sort also increased significantly in its number of iterations, whereas heap sort retained its linear complexity. (Figure 5)

Figure 5: Sorting analysis for reverse sorted data

4. For the fourth case, nearly sorted data was generated by sorting random sub-arrays of the original randomized data array, whose length is at least 70% of the original. Selection and bubble sorting methods retained their O(n2) complexity whereas insertion sort, the most efficient algorithm in this case, dropped to a linear complexity. Quick sort also notably improved whereas heap sort remained the same. (Figure 6)

Figure 6: Sorting analysis for almost sorted data

Conclusion: Clearly, the nature of the data provided as input significantly impacts the chosen algorithm’s time complexity and performance. Therefore, we can conclude that there is no “best” algorithm as such, there is only a “best” for each use case.

**Question 4**: Quick Sort Analysis

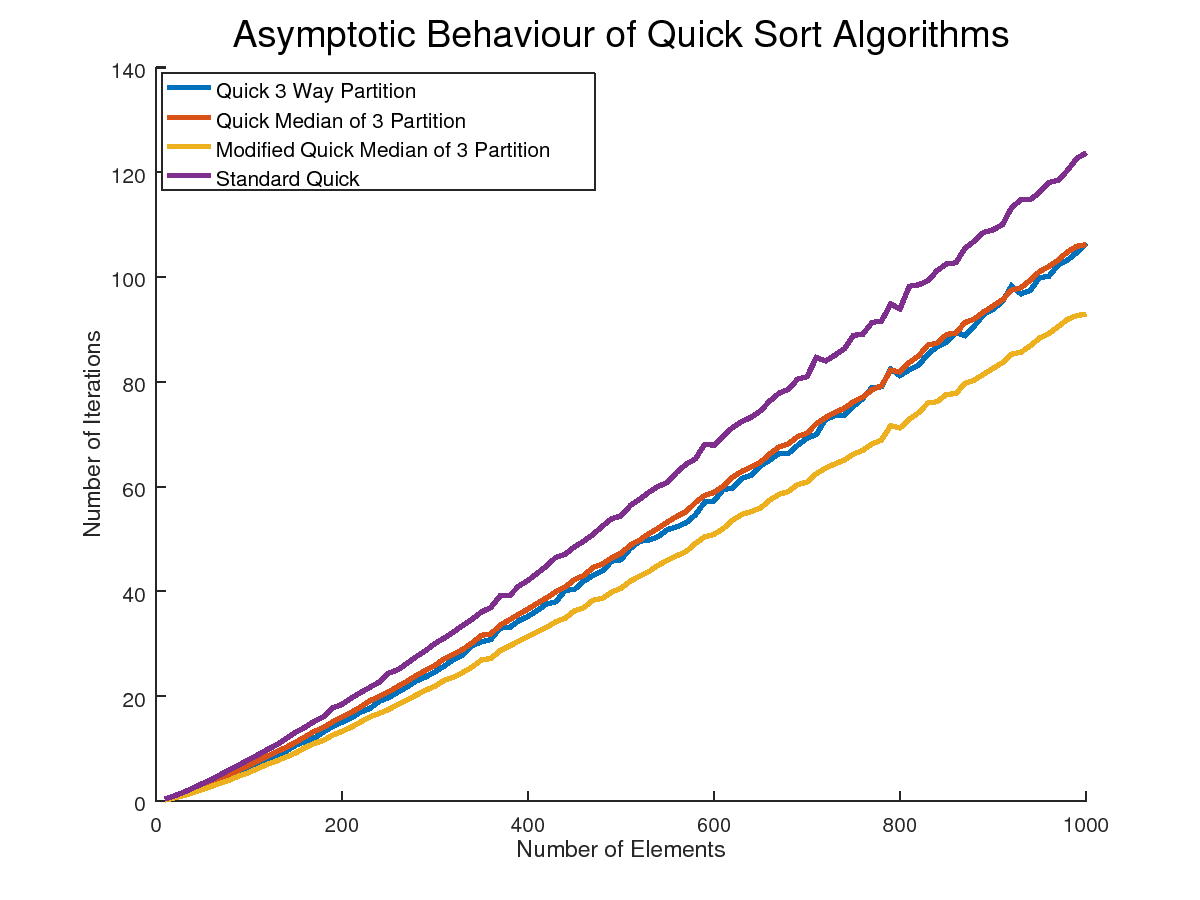
Compare the performance of variants of the quick sort algorithm for n=10 ... 1000. Use the results of Question 3 and accordingly modify your implementation of the quick sort algorithm. Repeat the experiment for 50 iterations and record the same set of statistics and compare the results for the two different sorting techniques.

**Solution**: Classical quick sort was implemented as a reference, where the pivot is the last element of the input array. The variants implemented were as follows[[1]](https://algs4.cs.princeton.edu/23quicksort/):

3-Way Partitioning: Sedgewick suggested using multiple pivots as a faster technique (multiquicksort). However this method does not yield any significant gains with more than 3 partitions. This is similar in logic to E.W. Djikstra’s Dutch National Flag problem.[[2]](https://en.wikipedia.org/wiki/Dutch_national_flag_problem)

Median of 3 Partitioning: Another reasonable increase can be obtained by using a better choice for pivot. This is done by taking the first, the middle and the last element of the input array and choosing the median of the three as input.

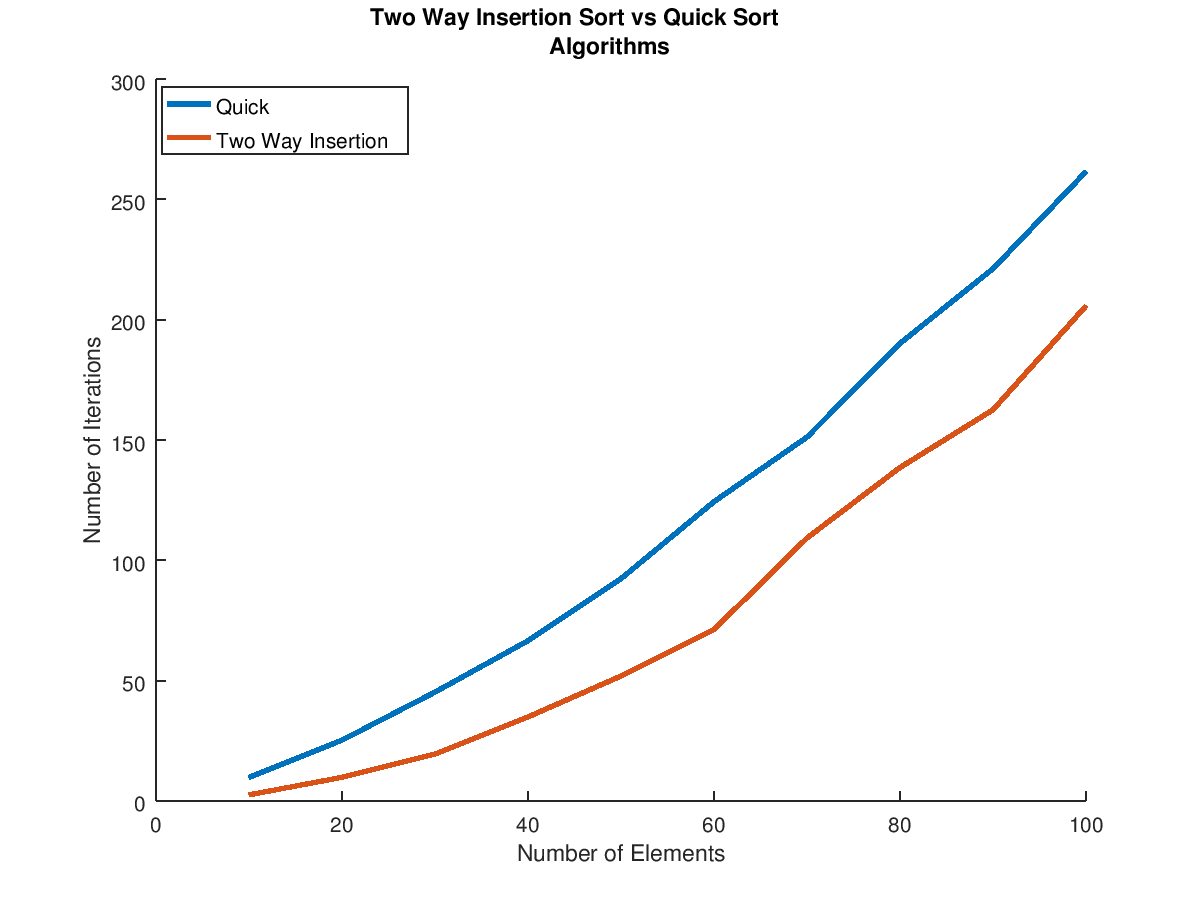
Modified Median of 3 Partitioning: This is a modification of the method stated above where insertion sort is used for sub-arrays of size less than 12.

Figure 7: Quick Sort Variant Analysis

From Figure 7 ,the modified Median of 3 Partitioning algorithm is noticeably more efficient than the other variants, of which classical quick sort is the worst, using randomized input.

**Question 5**: The two way insertion sort is a modification of the classical insertion sort. A separate output array of size 2n+1 is set aside. Initially the first element is placed into the middle of the “sorted” array. Insertion on either end is continued until an insertion in between a pair must be done. As before shifting must be performed, but unlike before, shifting can be done in any direction since there is additional room on both sides of the array. Compare the performance of two way insertion sort with quick sort.

**Solution**: It can be observed from Figure 8 that Two Way Insertion Sort significantly outperforms Quick Sort for randomized data inputs by about 30%.

Figure 8: Two Way Insertion Sort vs Quick Sort