Implement the following sorting algorithms.

- Insertion sort
- Merge sort
- Heapsort [vector based, and insert one item at a time]
- In-place quicksort (any random item or the first or the last item of your input can be pivot).
- Modified quicksort
    - Use median-of-three as pivot.
    - For small sub-problem of size <= 15 , you must use insertion sort.

Execution instructions:

- Run these algorithms for different input sizes (e.g. n = 1000, 2000, 3000, 5000, 10,000, 40,000, 50000, 60000). You will randomly generate input numbers for your input array. Record the execution time (need to take the average of several run) and plot them all in a single graph against various input sizes. Note that you will compare these sorting algorithms for the same data set.
- Also observe and present performance of the following two special cases:
    - Input array is already sorted.
    - Input array is reversely sorted.

Report:

About this code: I wrote and ran this code using Python in Visual Studio code with no errors. I used a library, matplotlib, in order to help finish this project (pip install matplotlib). Text file of this code can be found in the README.txt file in the attached folder, or, you can run the .py file that I also included. Running the code will provide three graphs that show the performance of each algorithm used. Please let me know if you encounter any issues. Note: when I ran my code using the input sizes given, I got an error: “RecursionError: maximum recursion depth exceeded”. I lowered the input sizes and had no issues.

<u>Insertion Sort:</u>

The algorithm is implemented in-place in the array, and doesn't require any additional data structures. The worst-case and average case time complexity is O(n^2), where n is the number of elements in the array. This makes it inefficient for large data sets. However, the best-case time complexity is O(n), which occurs when the input array is already sorted.

<u>Merge Sort:</u>

The Merge sort algorithm uses a divide-and-conquer strategy, and requires additional space to store the divided arrays. The worst-case, best-case and average time complexity is O(n log n), where n is the number of elements. This makes it a preferable choice for larger data sets, but the additional space requirement is its primary disadvantage.

<u>Heap Sort:</u>

The Heap sort algorithm requires a binary heap data structure. In the provided code, this is implemented in-place in the array. The worst-case, best-case and average-case time complexity is O(n log n), similar to Merge Sort. However, Heap Sort doesn't require any additional space which gives it an advantage over Merge Sort in space-limited scenarios.

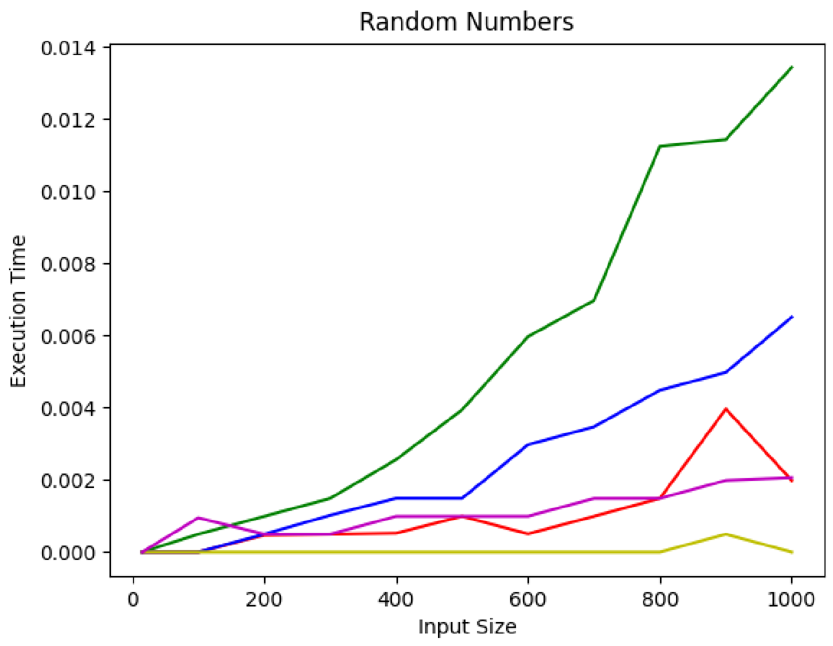
In-place Quick Sort:

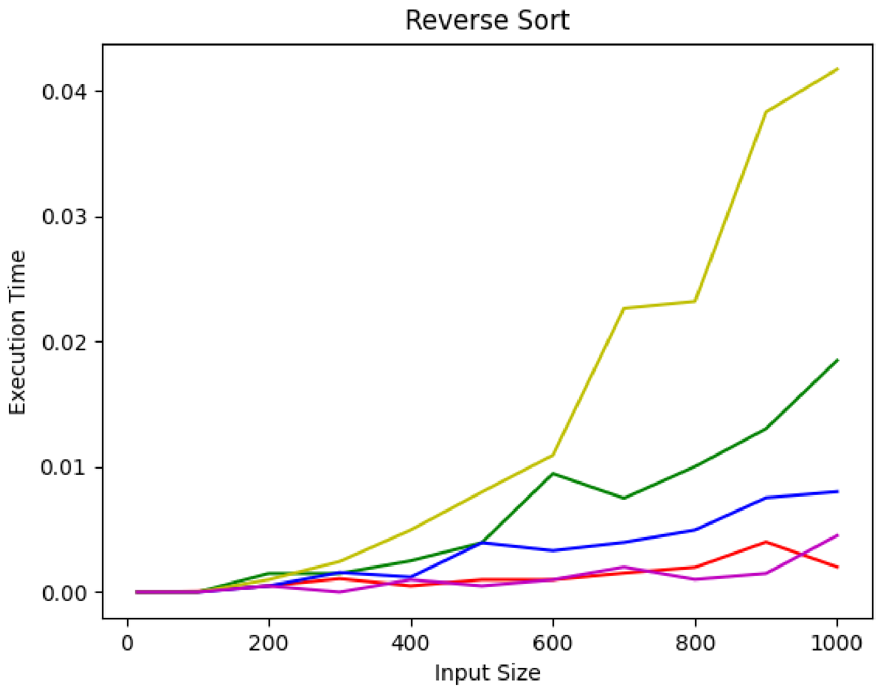
This quick sort is implemented in-place in the array, and doesn't require any additional data structures. The pivot element can be any element of the array. The average-case time complexity is O(n log n). However, the worst-case scenario, which happens when the smallest or largest element is always selected as pivot, is O(n^2). The best-case scenario is O(n log n), which occurs when the median is always selected as pivot.

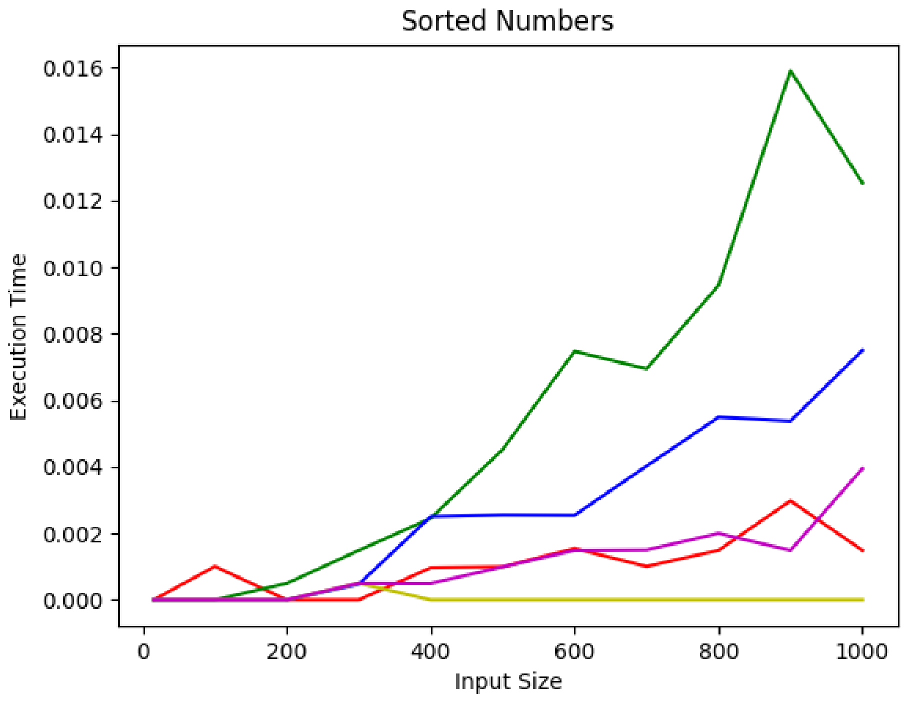
Modified Quick Sort:

This is a variant of Quick sort where insertion sort is used for small sub-arrays (of size 15 or less). The pivot selection strategy is median-of-three, which aims to limit the depth of the recursion and reduce the time complexity. The average-case and best-case time complexity is O(n log n). The worst-case scenario is still O(n^2), but due to the median-of-three pivot selection, this situation is very unlikely. The threshold for switching to insertion sort can be fine-tuned to optimize performance.

In the analysis, we see different sorting algorithms perform differently depending on the size of the array and the nature of the data. Insertion Sort is the simplest and performs well on small or nearly sorted arrays, but poorly on large, random arrays. Merge Sort and Heap Sort have good time complexity but Merge Sort uses significant additional space. Quick Sort is usually very fast, but can degrade to O(n^2) in unlucky scenarios. The Modified Quick Sort tries to get the best of both worlds, using Insertion Sort for small subarrays and a median-of-three pivot selection to reduce worst-case scenarios. Below are some of the results from the graphs that were plot using my program:




Reverse Sort
Execution Time
Input Size
0.00
0.01
0.02
0.03
0.04
0
200
400
600
800
1000


Sorted Numbers
Execution Time
Input Size
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014
0.016
0
200
400
600
800
1000