## Q1.) Merge sort vs Insertion sort

Hypothesis:   
I believe Insertion sort will be faster than Merge sort for n = 30.

Methods:  
I used the standard Python compiler, and used the following implementations:

Merge Sort: <https://www.programiz.com/dsa/merge-sort>  
Insertion Sort: <https://www.geeksforgeeks.org/python-program-for-insertion-sort/>

This is the code I wrote used to test the two sorts:

<https://github.com/phil-yu39466/HW4/blob/master/Q1/main.py>

I used the copy, random, and timeit libraries in my code. For random, I set a constant seed (43) when generating lists of random integers. This is to ensure that when changing the length of the list, the integers will stay the same for both sorts. For copy, I create a deepcopy of the list prior to passing it into the sorts; this is due to how timeit works.

For timeit, I used timeit.repeat. For the parameters:   
setup = setupCode: includes imports for copy and random, implementations for insertion and merge sort, and produces 2 separate, identical lists to be sorted. This code is run once for each timeit.repeat call.  
stmt = insertioncode/mergecode: create a deepcopy of the list to be sorted, which is then passed into the respective sorts. The deepcopy is to ensure that it is not sorting an already-sorted list.  
number = 10000: number of times to run stmt   
repeat = 1: number of times to run timeit.repeat

I ran the code starting with lists of size n = 5. This runs both insertion sort and merge sort, and returns the time (in seconds) to takes to sort the list. I repeat this process, changing the size of the list when creating them, incrementing by 5, up to 100.

Results:

|  |  |  |
| --- | --- | --- |
| **n** | **merge\_sort** | **insertion\_sort** |
| 5 | 0.0924696 | 0.0616136 |
| 10 | 0.1976042 | 0.1271877 |
| 15 | 0.2747477 | 0.1644652 |
| 20 | 0.401317 | 0.2313453 |
| 25 | 0.5111979 | 0.3527113 |
| 30 | 0.5828661 | 0.4880493 |
| 35 | 0.7356029 | 0.607581 |
| 40 | 0.8136969 | 0.8031112 |
| 45 | 0.9797939 | 0.9514509 |
| 50 | 1.0892483 | 1.1357004 |
| 55 | 1.2296886 | 1.367523 |
| 60 | 1.3415446 | 1.5779918 |
| 65 | 1.4727317 | 1.7778509 |
| 70 | 1.5876642 | 2.0073098 |
| 75 | 1.6598089 | 2.2298561 |
| 80 | 1.8140275 | 2.390214 |
| 85 | 2.1998137 | 3.0377222 |
| 90 | 2.0110022 | 3.1239322 |
| 95 | 2.1840503 | 3.3991645 |
| 100 | 2.3569082 | 3.7669881 |

Listed above are the runtimes of merge sort and insertion sort when passed in a list of random integers at size 5-100. Next to this is a corresponding scatter plot, x-axis is the length of the unsorted array, and the y-axis is the time it took to sort the array (in seconds). The blue data represents merge sort, whereas the orange data represents insertion sort.

Discussion:  
The breakpoint for when merge sort sorts faster than insertion sort is somewhere around n = 50. I was not surprised, as I knew at some point, merge sort will out speed insertion sort. The main challenge when collecting this data was when initially setting it up, insertion sort was significantly faster than merge sort at all lengths. This was because after the first repetition, there were already-sorted lists that were being sorted, making insertion faster as it performs better with already sorted data.

Conclusion:   
Under the conditions tested, merge sort is faster for unsorted data with length of n < 50, and insertion is faster for unsorted data of length n >= 50.

## Q2.) Hybrid Sorting

Hypothesis:   
I believe Tim Sort will be optimized at partition size k = 50, similar to the previous part.

Methods:  
I used the standard Python compiler, and used the following implementations:

Tim Sort: https://www.pythonpool.com/python-timsort/

This is the code I wrote used to test the two sorts:

<https://github.com/phil-yu39466/HW4/blob/master/Q1/main.py>

I used the copy, random, and timeit libraries in my code. For random, I set a constant seed (43) when generating lists of random integers. This is to ensure that when changing the length of the list, the integers will stay the same for both sorts. For copy, I create a deepcopy of the list prior to passing it into the sorts; this is due to how timeit works.

For timeit, I used timeit.repeat. For the parameters:   
setup = setupCode: includes imports for copy and random, implementations for insertion and merge sort, and produces 2 separate, identical lists to be sorted. This code is run once for each timeit.repeat call.  
stmt = insertioncode/mergecode: create a deepcopy of the list to be sorted, which is then passed into the respective sorts. The deepcopy is to ensure that it is not sorting an already-sorted list.  
number = 10000: number of times to run stmt   
repeat = 1: number of times to run timeit.repeat

I ran the code starting with lists of size n = 5. This runs both insertion sort and merge sort, and returns the time (in seconds) to takes to sort the list. I repeat this process, changing the size of the list when creating them, incrementing by 5, up to 100.