## 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/Q2/main.py>

I used the copy, random, and timeit libraries in my code, like the last problem. The only difference is my setup and stmt code I passed into timeit.repeat. For setup, instead of having insertion and merge sort, I used an implementation of tim sort. I then created 4 lists of sizes 100, 125, 150, 200. Next, I have 4 sets of code that run tim sort with a deep copy of the respective lists. I then run timeit.repeat of each list to generate my data.

The independent variable in this testing is the k value, or minrun in the code. This is when tim sort changes which sort it uses.

### Result:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **k/minrun** | **n = 100** | **n = 125** | **n = 150** | **n = 200** |
| 5 | 2.1773103 | 2.7315222 | 3.2663559 | 4.7974299 |
| 10 | 1.999324 | 2.4357918 | 2.8759316 | 4.207157 |
| 15 | 1.8495525 | 2.9683213 | 3.1028426 | 3.9547963 |
| 20 | 2.1545404 | 2.678178 | 3.2023872 | 4.7403486 |
| 25 | 1.9706087 | 2.8495727 | 3.1573538 | 4.5612227 |
| 30 | 2.0884445 | 2.8362996 | 3.6484159 | 4.5968448 |
| 35 | 2.6706319 | 3.1890355 | 4.2962726 | 5.6919299 |
| 40 | 2.6513389 | 3.2459311 | 4.2888905 | 5.7428344 |
| 45 | 2.8778329 | 3.5927898 | 4.5681761 | 6.3641719 |
| 50 | 2.9446835 | 3.7580359 | 4.7892188 | 6.1735827 |
| 55 | 3.0378892 | 4.2083846 | 4.9996197 | 6.7806789 |
| 60 | 2.9464914 | 4.3059676 | 5.5101138 | 6.7494722 |
| 65 | 2.9826626 | 4.1295527 | 5.5222614 | 7.3708787 |
| 70 | 3.1300121 | 4.2561005 | 5.7534434 | 7.3015468 |
| 75 | 3.2430552 | 4.0267291 | 5.6665282 | 8.1050371 |
| 80 | 3.5093573 | 4.1163599 | 5.9030342 | 8.1945001 |
| 85 | 4.0983843 | 4.2994884 | 5.9872808 | 8.1594239 |
| 90 | 4.8069795 | 5.179465 | 5.7818737 | 8.5115396 |
| 95 | 4.213129 | 4.704678 | 6.01839 | 8.8404072 |
| 100 | 4.1484296 | 5.0946155 | 5.9200112 | 8.8935484 |

Shown is the table and graph corresponding to the tim sort testing. In the table, the first column is the k/minrun value, which is changed on each run of the code. The next four columns represent sort on arrays of sizes 100, 125, 150, 200. The data is the runtime in seconds of tim sort. The graph is a visual representation of the data collected. The x-axis is the k value used, and the y-axis is the time it took for the program to sort the array (in seconds). We can see there are 4 separate sets of data, blue is list size = 100, orange is list size = 125, grey is list size = 150, yellow is list size = 200.

### Discussion:

Based on the graph and data, we can see that there is a general trend where the time it takes to sort decreases as the k value increases from 5 to 15. Once at 15-20, tim sort starts to slow down; it appears that the breakpoint for k is somewhere between 15-20. I was surprised as I initially thought the results would be like question 1, where the turning point was at size of 50. The only issue with this was understanding what to be testing/changing for tim sort.

### Conclusion:

Under the conditions tested, the breakpoint for partition size for tim sort is in the range of 15 <= k <= 20.

## Q3.) Comparing Dictionary Structures

Hypothesis:   
I believe the built in Python dictionary will out speed the binary tree implementation.

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

Binary Tree: <https://www.tutorialspoint.com/python_data_structure/python_binary_tree.htm>

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

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*[PLACEHOLDER](https://github.com/phil-yu39466/HW4/blob/master/Q2/main.py)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

I used the copy, random, and timeit libraries in my code, like the last problem. The only difference is my setup, stmt, and number parameters that I passed into timeit.repeat. For setup, I have the implementation for binary search tree. I then generated a list of random integers of varying size. Next, I have two code blocks that are repeated when passed into timeit.repeat. The first is bst\_code, which creates a deep copy of the list of random integers. It creates a root node with the first element in the list, and then inserts the rest of the elements into a binary tree. For the other code, I create another deepcopy of the list of random ints, and then I just add each element to a dictionary. I set number = 1, as there was no need to repeat the insert. then run timeit.repeat on each data structure to get the times it takes to insert all the elements.

The independent variable in this testing is n, or length of the list of random integers. We start at 10, increment by a factor of 10.

### Result:

|  |  |  |
| --- | --- | --- |
| **n = len(array)** | **BST** | **Python built-in dict** |
| 10 | 2.17E-05 | 1.18E-05 |
| 100 | 8.7E-05 | 5.3E-05 |
| 1000 | 0.0010696 | 0.0006567 |
| 10000 | 0.0084611 | 0.0059156 |
| 100000 | 0.0740906 | 0.0542201 |
| 1000000 | 0.8816133 | 0.5256766 |
| 10000000 | 8.4514822 | 5.7114793 |

Shown above are the table and graph corresponding to the testing. On the graph, the x-axis represents the number of elements to insert into the data structures, and the y-axis represents the time it took to insert all the elements (in seconds). We can see from the graph that the built-in dictionary data structure is strictly faster than the binary search tree implementation. The blue data represents the binary tree, and the orange represents Python’s built-in dictionary.

### Discussion:

Based on the graph and data, we can clearly see that the BST is strictly slower when it comes to inserting elements. I was not surprised by the results, as I knew beforehand that Python’s built-in dictionary was efficient. There was no issue collecting data, the only difficulties were figuring out what dictionary backends to use.

### Conclusion:

Under the conditions tested, the insert operation for the data structure for Python’s built-dict is significantly more efficient than a binary search tree’s insert.