Sorting Lab:

An Analysis on Efficiency of Sorting Algorithms

Laeek Ahmed

Raj Rehman

Norman Liang

COMPSCI 2XC3

Professor Maccio

February 9th, 2023

Table of Contents

Table of Figures

Summary

- Of the "bad" sorting algorithms, selection sort performs considerably better than bubble sort and insertion sort.
- "Good" sorting algorithms are not always the most efficient option, and may even perform worse compared to "bad" algorithms, as each algorithm has its own strength dealing with lists of different lengths and degree of randomness and other factors.
- Understanding the strengths of each sorting algorithm is important when choosing what to use in different circumstances, and can help us develop "hybrid" algorithms that may be optimal for many different list types

Experiment 1:

To compare the run times for bubble sort, insertion sort and selection sort

- There are 3 functions `bubble_test(n, m, step)`, `insertion_test(n, m, step)` and `selection_test(n, m, step)` which run tests on the 3 "bad" sorting algorithms.
- They generate 100 random lists (using `create_random_list(length, max_value)`) with different lengths ranging from 1 to 901 elements and get the average time taken to sort each of these lists from 25 runs.

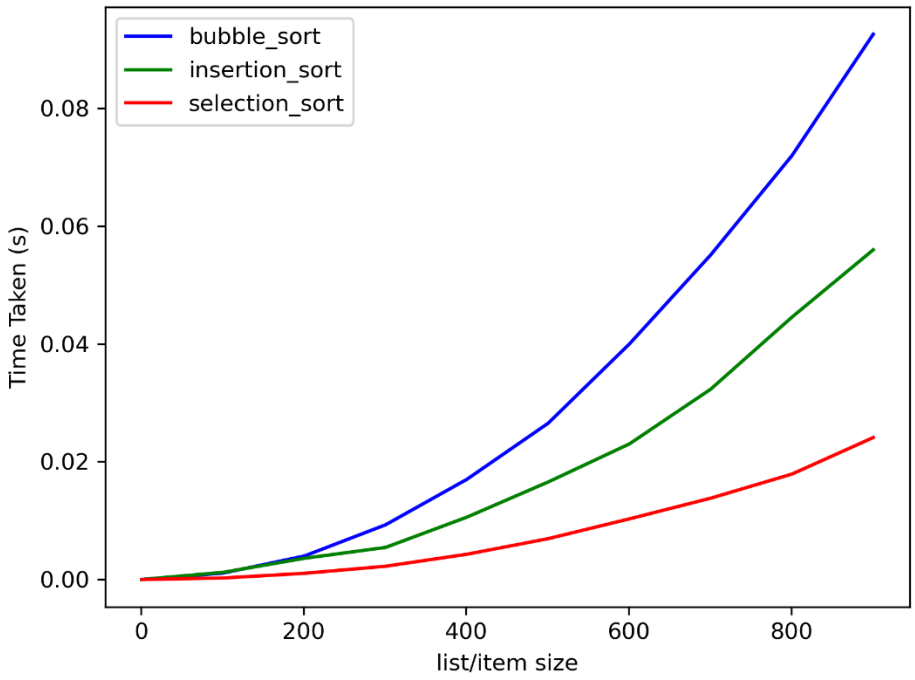

Figure 1.

- As we can see, bubble sort has the worst performance compared to the other 2.
- All the three curves tend to climb up as the list length increases, the only difference is in the rate at which they climb due to the coefficient which is highest for bubble and lowest for selection.

- Bubble sort is the slowest as its approach is very simple and inefficient whereas insertion sort is an adaptive algorithm ie. It depends on the degree of disorder in the array, this is also the reason why it performs better than selection sort in some cases.
- The time complexity of each of these 3 sorting algorithms is $O(N^2)$ which can also be observed from the quadratic growth of the 3 curves in the graph.

Experiment 2:

To compare the run times for bubble sort and selection sort to the input list's length

- There are 2 functions `experiment(n,k,step)` and `experimentMod(n,k,step)` which compares the runtime between the original sorting algorithm and the modified version.
- They generate 100 random lists (using `create_random_list(length, max_value)`) with different lengths ranging from 1 to 901 elements and get the average time taken to sort each of these lists from 25 runs.

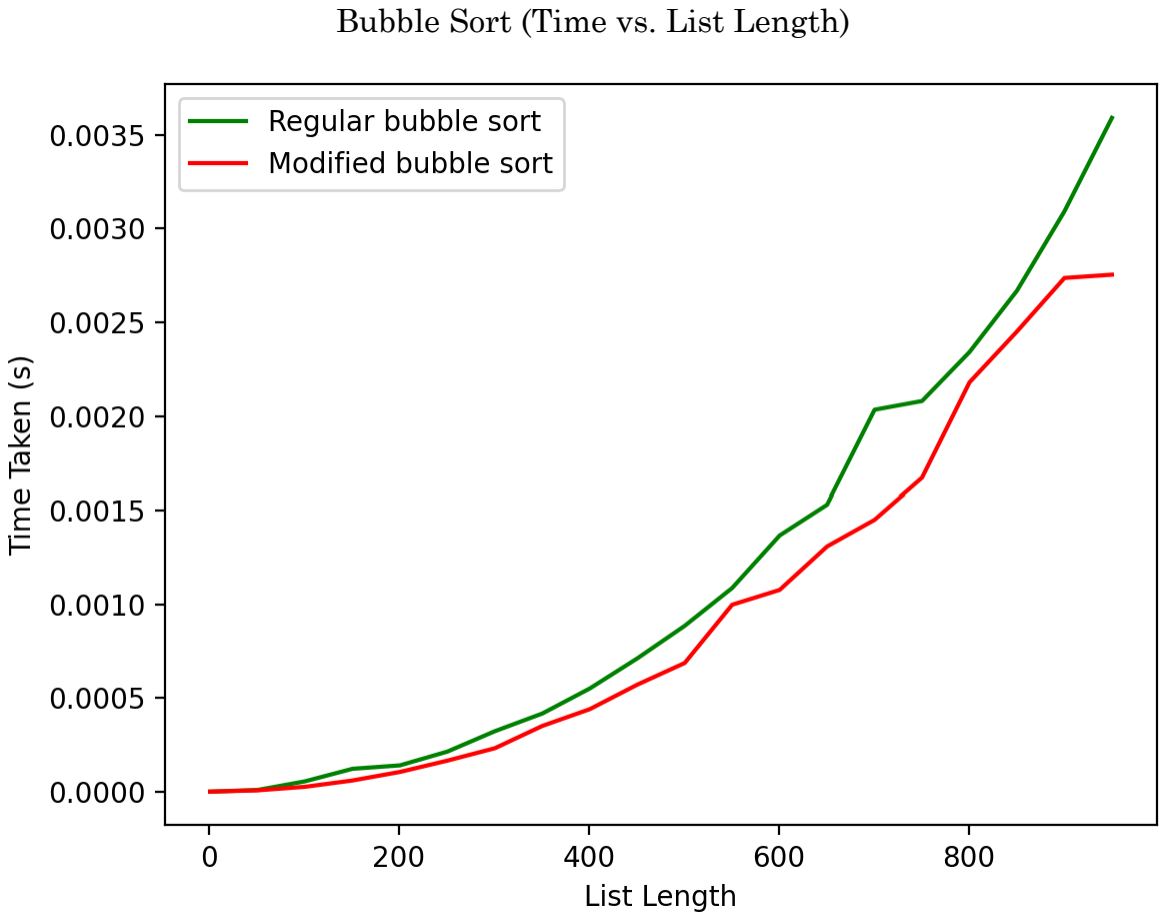

Figure 2

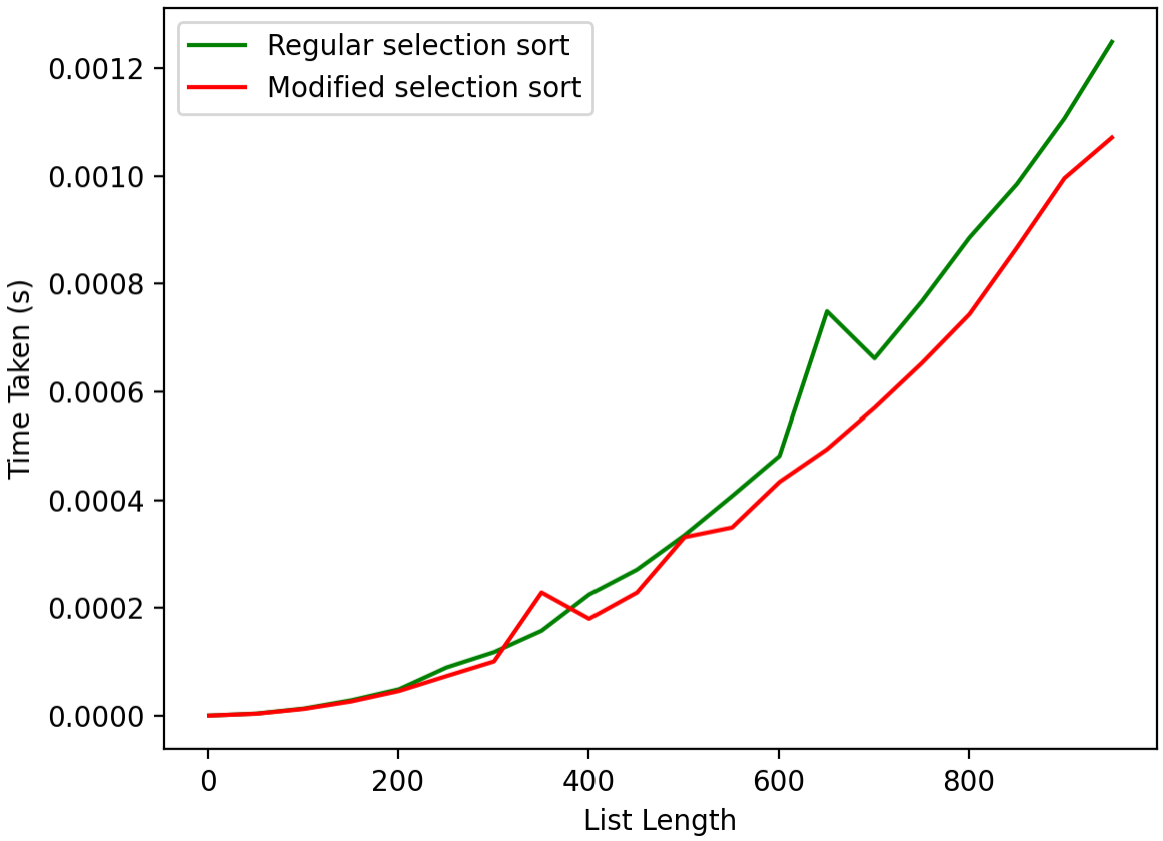

Figure 3

- Bubble sort and selection sort are somewhat like opposites. Both sorting algorithms benefit from the modifications, but bubble sort's improvement is more significant than selection sort's when the list length is not too long.
- In bubble sort's runs, we found that changing the number of times the test is ran does not change the improvement by much, which is similar to selection sort. However, when the length of the list grows, the improvements provided by bubble sort's modifications are lessened.
- On the contrary, the modified selection sort doesn't improve the runtime significantly when the list length is short, but performs much better than the original when the list length is longer.

Experiment 3:

To compare the run times of the 3 “bad” sorting algorithms; insertion sort, bubble sort and selection sort.

- There are 3 functions `insertion_sort_test(n, m, swapp)`, `bubble_sort_test(n, m, swapp)` and `selection_sort_test(n, m, swapp)` which run tests on the 3 “bad” sorting algorithms. About 10-12 different experiments were run using different values.
- The list length is set constant and fixed at 1000. The number of swaps is ranging from 0 to 100. The test functions generate 100 random lists (using `create_random_list(length, max_value)`) and get the average time taken to sort each of these lists from 5 runs.

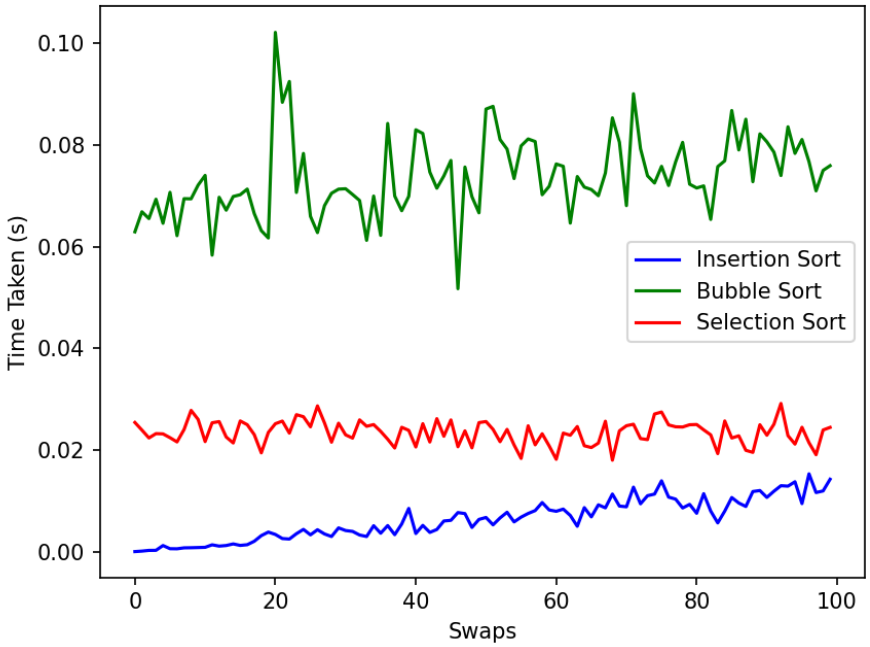

Figure 4

- As we can see, bubble sort has the worst performance, followed by selection sort and insertion sort.
- When the list is the most sorted, insertion sort runs the fastest, compared to bubble and selection sort. As the list becomes more disoriented, insertion sort starts becoming more ‘bad’. Insertion sort has a better run time when the number of swaps is less. As the number of swaps increases, insertion sort becomes more ‘bad’.

Experiment 4:

To compare the run times for heap sort, merge sort and quick sort

- There are 3 functions `heap_test(n, m, step)`, `merge_test(n, m, step)` and `quick_test(n, m, step)` which run tests on the 3 "good" sorting algorithms.
- They generate 10 random lists (using `create_random_list(length, max_value)`) with different lengths ranging from 1, 101, 201 to 901 elements and get the average time taken to sort each of these lists from 50 runs.

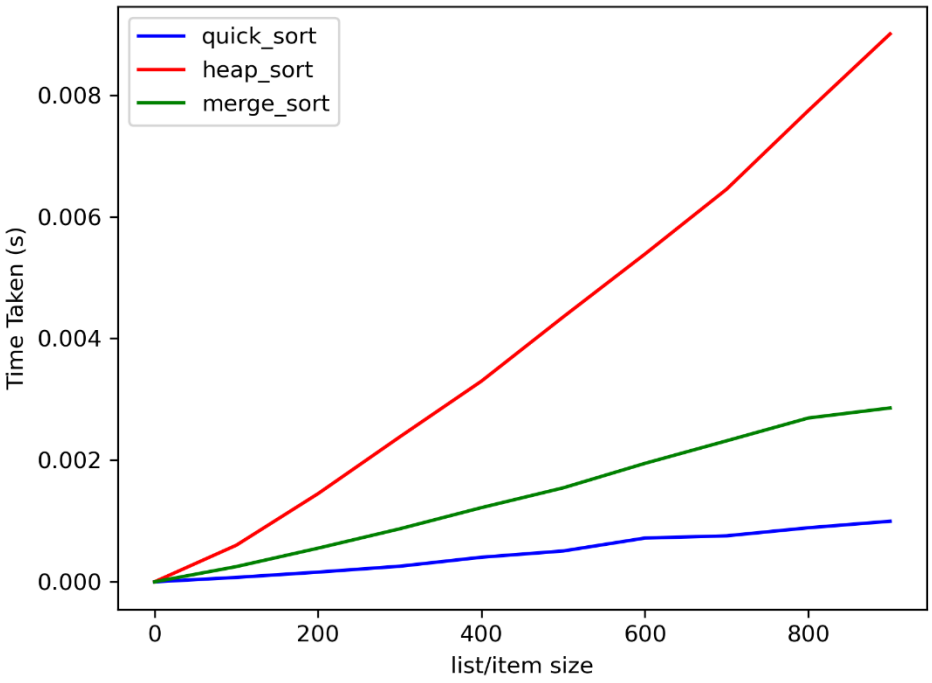

Figure 5

- As we can see, heap sort has the worst performance compared to the other 2.
- All the three curves tend to climb up as the list length increases, the only difference is in the rate at which they climb due to the coefficient which is highest for heap sort and lowest for quick sort.
- Heapsort is less efficient than merge sort in the average case due to the overhead of building a heap.
- Merge sort is less efficient than quicks sort as it is not in place and has a higher overhead as it deals with additional arrays and quick sort has a more efficient partitioning step .
- The time complexity of each of these 3 sorting algorithms is $O(N \log N)$ which can also be observed from the linearithmic growth of the 3 curves in the graph.

Experiment 5:

To compare the run times for heap sort, merge sort and quick sort

- There are 3 functions `experiment(n, k, s)` which tests and compares how the number of swaps away from a sorted list affects the runtime of quick sort.
- They generate 100 random lists (using `create_near_sorted_list(length, max_value, swaps)`) with different number of swaps ranging from 0 to 99 and get the average time taken to sort each of these lists from 50 runs.

	List Length	Times Ran	# of Swaps	Time (s)
Run #1	100	100	100	0.000122
Run #2	100	100	100	0.000110
Run #3	100	10000	100	0.000109
Run #4	100	10000	100	0.000099
Run #5	1000	100	100	0.001429
Run #6	1000	100	100	0.001443
Run #7	100	100	50	0.000146
Run #8	100	100	50	0.000123
Run #9	100	100	10	0.000236
Run #10	100	100	10	0.000219
Run #11	100	100	0	0.000492
Run #12	100	100	0	0.000639
Run #13	100	100	1000	0.000125
Run #14	100	100	1000	0.000123

Table 1

Quick Sort (Time vs. # of Swaps)

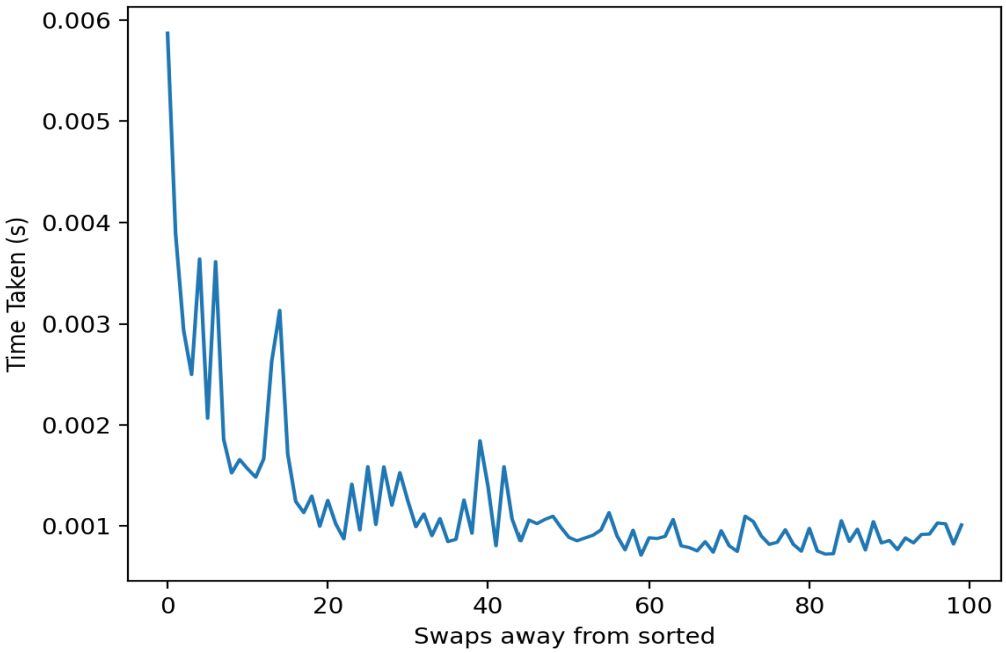

Figure 6

Note: The constant list length used is 100.

- From our experiment, we can see that changing the number of times the algorithm runs and changing the list length does not greatly impact how well quick sort works.
- Changing the list length does cause the run time to wildly differ, but that is to be expected, and the change in time is usually in proportion to the change in size.
- However, the most noticeable change would be the way that the further the list is to being sorted, the more efficient the sorting algorithm is. In other words, the more randomized a list is, the better quick sort works.
- In our graph, we notice that the performance does not improve much further after around 10 swaps from sorted. This is most likely because after 10 swaps, the list has already been fairly well randomized, and any further swaps would simply create a slightly different, but similarly randomized list.

Experiment 6:

To compare the run times for single pivot and dual pivot quick sorts.

- There are 2 functions `dual_quick_test(n, m, step)`, `quick_test(n, m, step)` which run tests on the 2 versions of the quick sort algorithm.
- They generate 100 random lists (using `create_random_list(length, max_value)`) with different lengths ranging from 1 to 9901 elements and get the average time taken to sort each of these lists from 5 runs.

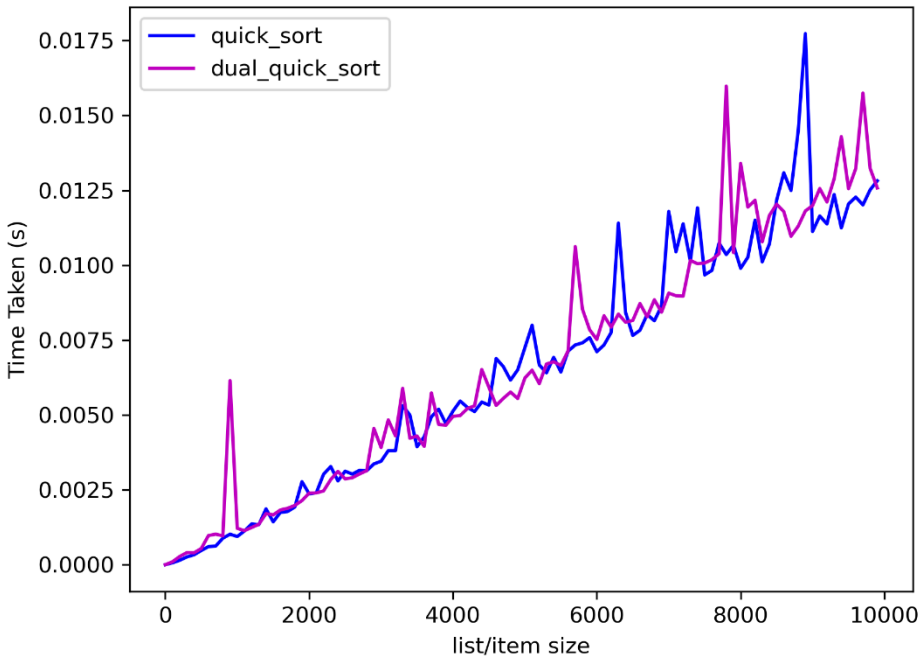

Figure 7

- As we can see from the graph, this change is not worth doing as both the versions perform more or less the same, there doesn't seem to be any significant improvement in the run time.
- The improvement in performance is also not always guaranteed and depends on various factors such as the distribution of data and the choice of pivot elements.
- It is also due to the fact that even though the dual pivot version might be better in some cases, there is a trade-off due to the increased overhead from an additional recursive step and an additional partition.
- The time complexity of each of these 2 sorting algorithms is still $O(N \log N)$ which can also be observed from the linearithmic nature of the 2 curves.

Experiment 7:

To compare the run times for bottom up and the traditional top-down merge sort.

- There are 2 functions `bu_merge_test(n, m, step)`, `merge_test(n, m, step)` which run tests on the 2 versions of the merge sort algorithm.
- They generate 100 random lists (using `create_random_list(length, max_value)`) with different lengths ranging from 1, 101, 201 to 9901 elements and get the average time taken to sort each of these lists from 5 runs.

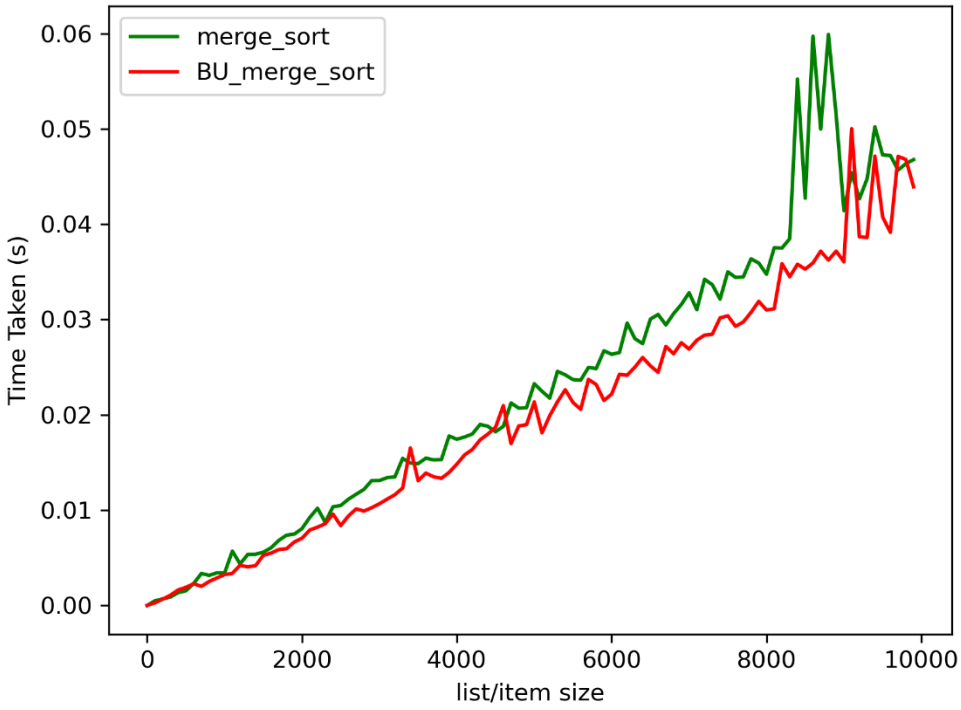

Figure 8

- As we can see, there is a slight improvement which is likely to increase as we test it on bigger lengths.
- Bottom-up Merge Sort is faster than Top-Down Merge Sort because it avoids the overhead of recursion.
- The recursive approach in top-down leads to a lot of function calls, each with its own stack frame, which can be expensive in terms of time.
- The time complexity of each of these 2 versions is still $O(N \log N)$ since we did not change the `merge(left : list, right : list)` function which is $O(N)$. We just replaced the recursive part with an iterative part having $O(\log N)$ time complexity.

Experiment 8:

To compare insertion sort to merge sort and quick sort for small lists,

- There are 3 functions `insertion_sort_test(n, m, step)`, `quicksort_test(n, m, step)` and `merge_sort_test(n, m, step)` which run tests on the insertion, quick, and merge sorts respectively. The functions generate a random list (using `create_random_list(length, max_value)`) and get the average time taken to sort each of these lists from 5 runs. A total of 10 experiment runs were done using different values.
- Keeping in mind the given instruction to have small list length, the list length has been kept as 25.
- There is another function `hybrid_inmer_sort (L, optval)`, which is the 'hybrid' sort of insertion and merge sort, in which optval is the 'optimal value' which can be given by us. It is the value of list length below which we want the function to use insertion sort, and otherwise use merge sort.

Figure 9

- As we can see, insertion sort initially beats quick sort and merge sort. But it finally goes bad, after quicksort beats it at list length = 5 and merge sort beats it list length = 20. As time and list length increases, insertion sort keeps becoming more 'bad'. This shows that insertion sort performs better for shorter lists compared to merge and quick sort.
- All the three curves tend to climb up as the list length increases, the only difference is in the rate at which they climb, which is highest for insertion and lowest for quicksort.
- These results and conclusions help us understand how various sorting algorithms work. They help us understand more about their performance, their runtimes, their merits and demerits. By comparing the performance of these sorting algorithms on lists of various lengths, we are thus able to determine their efficiency.

Appendix:

To find the implementation and code for each experiment, they are within the goodsorts.py and badsorts.py files. For the first three experiments, the experiments conducted in Lab 2, they can be found in badsorts.py. The rest of the experiments can be found in the file goodsorts.py.

The files begin with the provided sample code for the traditional sorting algorithms, the tests and modified algorithms are further down. For some testing functions, you may need to edit the code manually to change the sorting algorithms that you are testing. Please follow the comments within the code for further instructions.