2XB3 Lab3

Yi Luo
400254211
luoy94@mcmaster.ca
L02

Frank Yang
400243777
yangf51@mcmaster.ca
L02

Harrison Chiu
400261400
chiuh@mcmaster.ca
L02

February 2, 2021

Abstract

This paper analyzes different time complexities of quicksort. Specifically it looks into the time complexities of various quicksort implementations, including multiple pivots, different pivots, small lists, and worst case performance.

Quicksort Inplace

Quicksort Multi-pivot

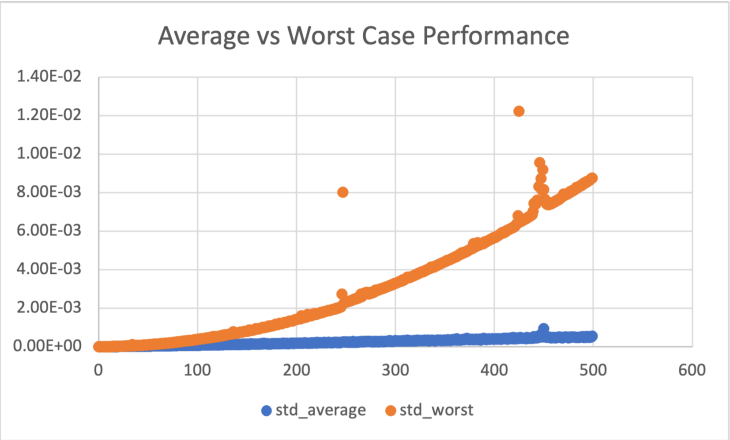
Worst case

The worst complexity of quicksort is n^2. It happens under the following circumstances. The given array is already sorted or reversely sorted with the leftmost or the rightmost element chosen as the pivot. The given array's elements are the same.

All of them would lead to a situation where the divided arrays are totally unbalanced such that one array has only one element while the other has $n-1$ element with n being the size of the parental array. As a result, the time complexity would be as such:

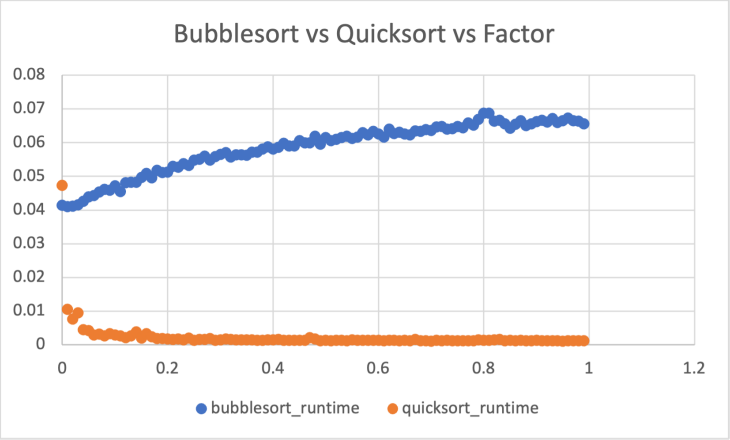
$$n + (n-1) + (n-2) + (n-3) + \ldots + 2 = n * (n+1)/2–1 = \Theta n^2$$

A graph is attached showing the average case performance vs the worst case performance vs n.



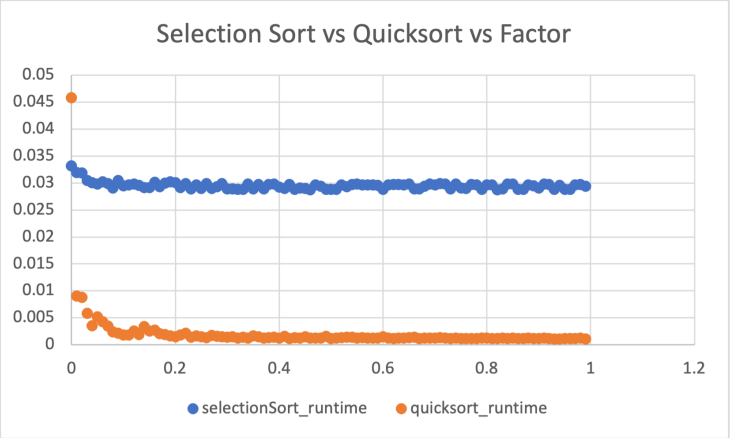
The function `create_near_sorted_list` creates near-sorted-list based off some input factor. The lower the factor, the closer to being sorted the created list is.

Bubble sort will out-perform quicksort when a given list is close to being sorted since it only takes one or few iterations for the bubble sort to terminate, yielding a linear complexity(Θn). Attached is a graph comparing their performance with factor ranging from 0 to 0.99. Note that the lists sorted are of length 996 instead of 1000 as specified since it would otherwise generate `RecursionError`.



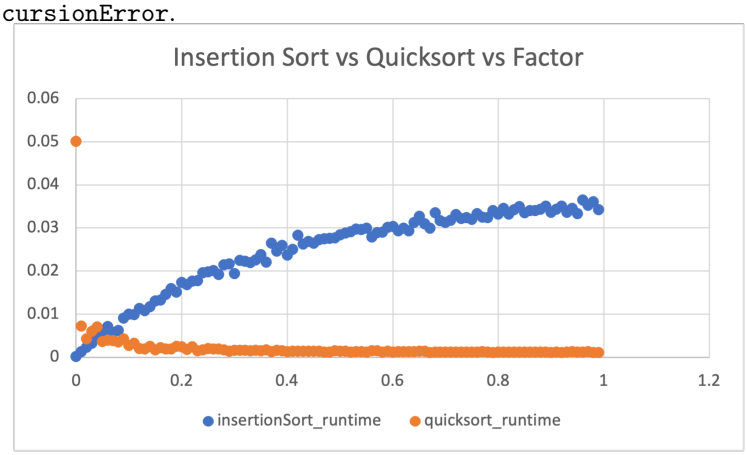
Bubble sort out-performs only when factor is extremely close to zero.

Selection sort always has a complexity of n^2. Attached is a graph comparing their performance with factor ranging from 0 to 0.99. Note that the lists sorted are of length 996 instead of 1000 as specified since it would otherwise generate `RecursionError`.



Selection sort out-performs only when factor is extremely close to zero.

Insertion sort will out-perform quicksort when a given list is close to being sorted, yielding a linear complexity(Θn). Attached is a graph comparing their performance with factor ranging from 0 to 0.99. Note that the lists sorted are of length 996 instead of 1000 as specified since it would otherwise generate `RecursionError`.



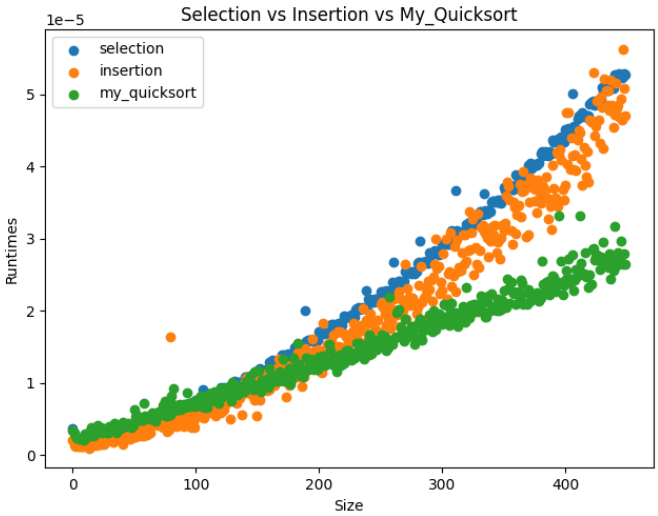
Insertion sort out-performs when factor approximately is smaller than 0.05.

Small lists

For small lists (small values of n), elementary sorting algorithms' time complexity is measured. Selection and insertion sorts' running time were tested. The results were (also seen in the graph but these are the averages for values of $N < 50$) `{'std': 9.19201014767168e-06, 'insertion': 5.821419908897951e-06, 'selection': 9.8939301096` (std is the `my_quicksort`) where the value of the dictionary is the average over 100 trials of running the function in seconds. It is clear that the standard quicksort is very slow compared to insertion and our final sort for small values of n

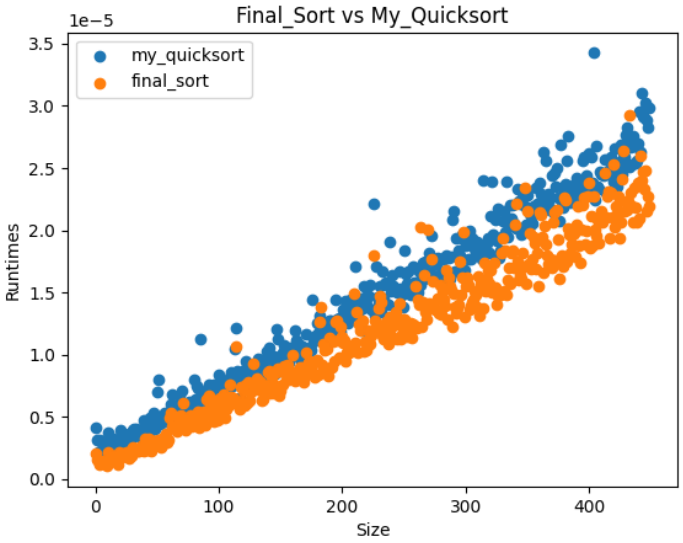
even though they have $O(n^2)$ time complexity compared to quicksort's time complexity of $O(nlog(n))$. The difference is approximately $4e-6$ between `std` and `insertion`. This is likely because for small values of n, quicksort requires more memory from its recursive calls and insertion sort requires few swaps and comparison for small arrays. Furthermore, $O(n)$ is a measure of its running time for large values of n which ignores all constant factors. Only for large values of n do the constant factors such as $+c$ are ignored because the growth of the function, $O(n)$, dominates all other factors. Selection sort is quite slow at $9.89e-6$. This is expected; its running time is always $O(n^2)$. This is because no matter the ordering of the array, selection sort will traverse through the array approximately $\frac{N^2}{2}$ times to find the minimum N times.

The graph also shows that for small values of $N < 200$, insertion sort has smaller runtimes than the other algorithms. However, this changes as N becomes much larger and its runtimes become the expected value from its big-O values. To conclude our most notable observations, insertion sort is faster than quicksort for small values of n and can be used to improve our quicksort.

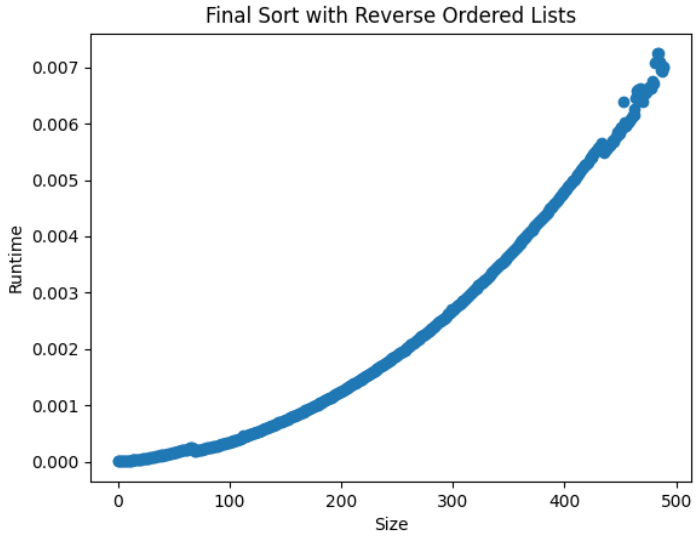


Our implementation of an improved quicksort is a hybrid of insertion sort and quicksort. From our observations, insertion sort has a lower running time than quicksort for small values of n. Our main goals were to improve the standard quicksort given in `my_quicksort()` both in terms of time and space complexity. Using this to our advantage, we can switch between quicksort and insertion sort to use both of them during their ideal conditions. It will still mainly use quicksort because for large values of n, it is still much faster than insertion sort ($O(nlog(n))$ ¡ $O(n^2)$ for large n). However, for small values of n (we arbitrarily chose 10 as the small value), the sorting algorithm will switch to insertion sort. When the array has reached the condition to start the insertion sort, it will have already been partly sorted. Insertion sort is very quick for partly sorted arrays because it will need to do less comparisons, and therefore, less swaps. Its

running time in this case (small array and partly sorted) will be close to $O(kn)$ for some $1 \geq k$. Furthermore, the quicksort algorithm does not follow the style of `my_quicksort` because it requires more space when it physically creates new subarrays to recursively sort them; it has a smaller space complexity. Since the sorting algorithm now stops its recursion when its subarray is approximately 10, its recursion depth is significantly lower.



The worst case of this algorithm is a reverse sorted list. It is the worst case for both quicksort and insertion sort (which our algorithm uses). For quicksort, it causes the partition to be very unbalanced because every number will be less than or equal to it (if we choose the first number as our pivot number). This will also maximize the recursive depth to a size of nearly $N - 1$. Furthermore, the final subarrays in which insertion sort runs will also be reverse sorted and so, it will have a maximum runtime of 10^2 (we arbitrarily chose 10 as the breaking point for insertion sort to run). Since both of these sorting algorithms are N^2 in the worst case, our algorithm is also N^2 as seen in the graph. It has a similar shape to N^2.



Since our sorting algorithm is still predominantly quicksort and still mostly uses quicksort to sort, the best case will still depend on quicksort's best case. Insertion sort is only used for very small lists, its time complexity is negligible. Its best case is when the pivot number is exactly in the middle such that it partitions the numbers evenly. An example is when the array is randomly ordered because a random pivot will be approximately in the middle. This also minimizes the recursive depth (and thus the memory usage) to $\log_2 N$ (height of the binary tree). Quicksort will have a best case time complexity of $O(N \log N + c)$ or $O(N \log N)$ where c is some constant from insertion sort's time complexity. As seen in the graph, the running times increase somewhat linearly with a small curve at the beginning, similar to $O(N \log N)$.