## Report

2. Create a table.

| | | | | **D (list sizes below)** | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | **A** | **B** | **C** | *50 (e-5)* | *500* | *1000* | *2000* | *5000* | **E** |
| **Bubble** | 4.41 | 0.651 | 5.59 | 9.46 | 0.0100 | 0.0431 | 0.174 | 1.12 | 0.673 |
| **Insertion** | 1.72 | 0.000618 | 3.42 | 4.67 | 0.00368 | 0.0163 | 0.0674 | 0.431 | 0.263 |
| **Selection** | 1.56 | 0.396 | 1.93 | 4.40 | 0.00394 | 0.0156 | 0.0629 | 0.395 | 0.245 |
| **Merge** | 0.0258 | 0.0120 | 0.0257 | 7.91 | 0.00104 | 0.00208 | 0.00446 | 0.0128 | 0.00958 |
| **Quick** | 0.0155 | 0.799 | 3.46 | 3.43 | 0.000503 | 0.00115 | 0.00253 | 0.00715 | 0.00558 |

Table 1. Average performance timings for each algorithm for each experiment in seconds, rounded to 3 significant digits.

3. These experiments suggest that not all algorithms are suitable for all scenarios. Answer the below questions (in max 3-4 lines each) based on your observations:

a. Which algorithms in which experiments had best performance? Why? Ground your reasoning in complexity analysis.

Quicksort in random list experiments (A, D, E) performed best because when not processing a worst-case input, its complexity is linearithmic—leagues better than the quadratic algorithms, and slightly better than mergesort (linearithmic) due to smaller constants (partition is much less costly than merge). But when the input is near-sorted lists (B), insertion sort runs significantly faster because it is functionally linear in this case. Merge was best for C, steadily performing well while the only faster algorithm, quicksort was drastically affected (reason in answer below).

b. Which algorithms in which experiments had the worst performance? Why? Ground your reasoning in complexity analysis.

Quicksort performed much worse than its usual with a reversed list (C) and worse than usual with a near-sorted list (B) because this version's partition had been implemented taking the pivot from one end of the list, and the more order the input list has, the less likely that value is to be around the middling range. C was actually the worst-case input for quicksort, making its complexity quadratic. C was also worst-case and quadratic for insertion sort and bubble sort, both having to essentially move one item in place each pass over the list. However, in general, bubble sort performs the worst because its complexity is always quadratic, with high constants due to costly operations of only being able to move elements by swapping with neighbours.

c. Under which experiments Merge Sort performed better than Selection sort? Why?

Mergesort was better than selection sort in all of the experiments except for the smallest sized list trial in experiment D because mergesort is linearithmic while selection sort is quadratic, and the former is much faster than the latter and the cost of the algorithm climbs much slower as the size of the input climbs. However, when the input size is small, mergesort's large constants means it in fact runs slower than selection sort (essentially doing unnecessary work).

d. Traditionally, Insertion Sort is worse than Merge Sort. What were the scenarios in which this is not true? If these scenarios were not covered in these experiments, then what would that scenario look like?

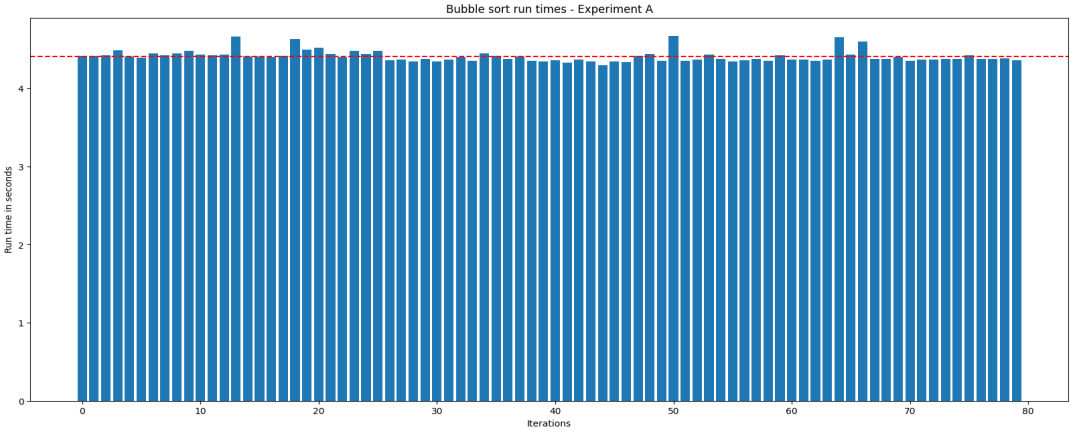
Insertion sort was slightly better than mergesort for the 50-items-long trial in experiment D for the same reason that selection sort was. Insertion sort was significantly better than mergesort (linearithmic) in experiment B with near-sorted lists because that is the best case for insertion sort, in which case insertion sort's complexity is linear. This happens when the number of inversions in the input list's order is small enough to be linearly proportional to the list's length.

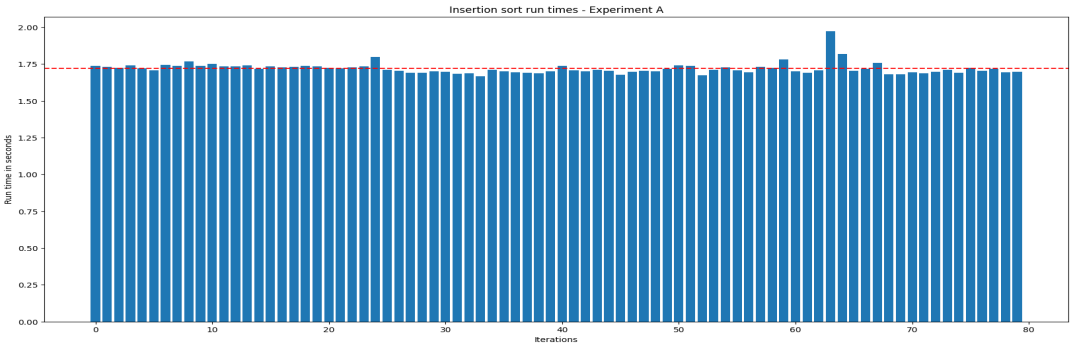
e. Which experiment represents an average case for Quick Sort? If this was not covered in the experiment, then what would that scenario look like?

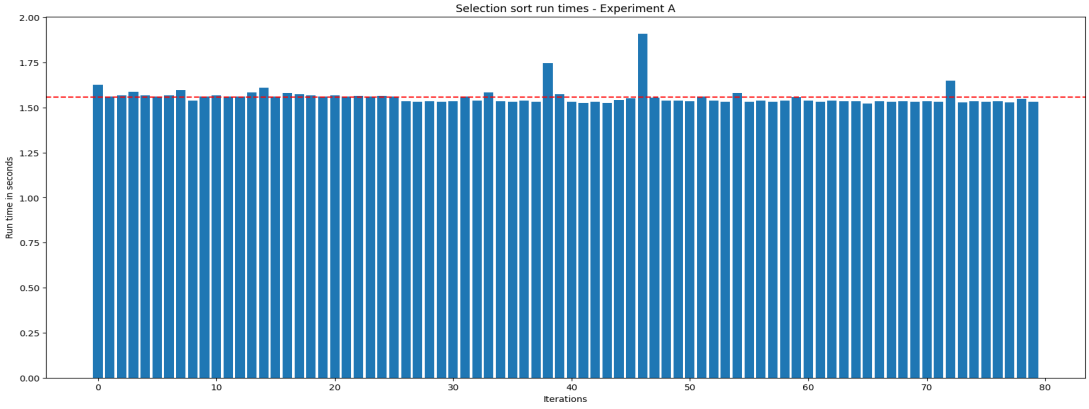
Experiments A, D, E, the ones where quicksort performed the best, the ones where the order of the items in the lists were random, represent an average case for quicksort. Average case complexity for quicksort is only a constant factor larger than best case complexity for quicksort, and is still linearithmic. In those randomly ordered lists, an item on the end chosen to be the pivot by the partition function has a good enough chance to the split the input into two lists of roughly equal enough lengths, often enough to be reliably efficient.

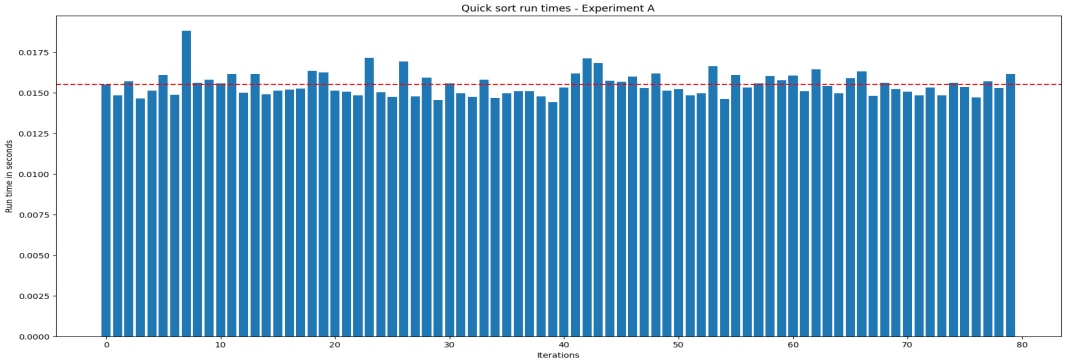
# Appendix

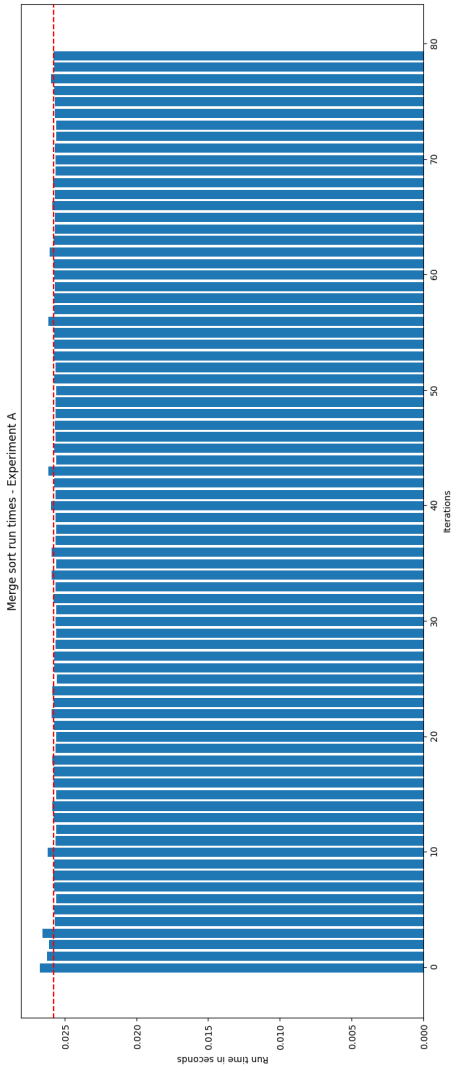
## A. The sorting algorithms on a random list of 10,000 items, 80 trials



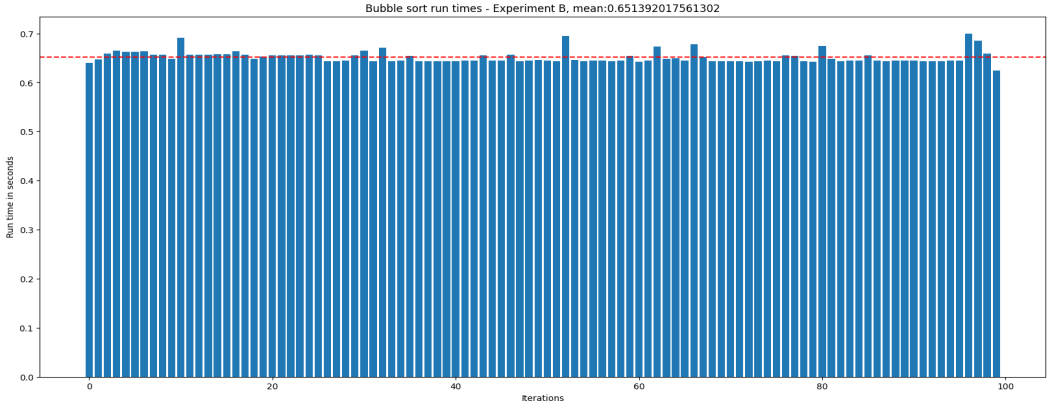



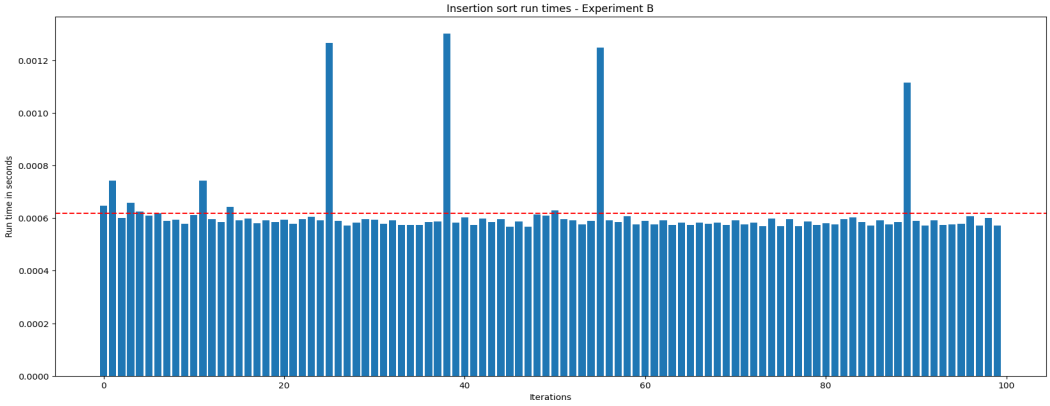



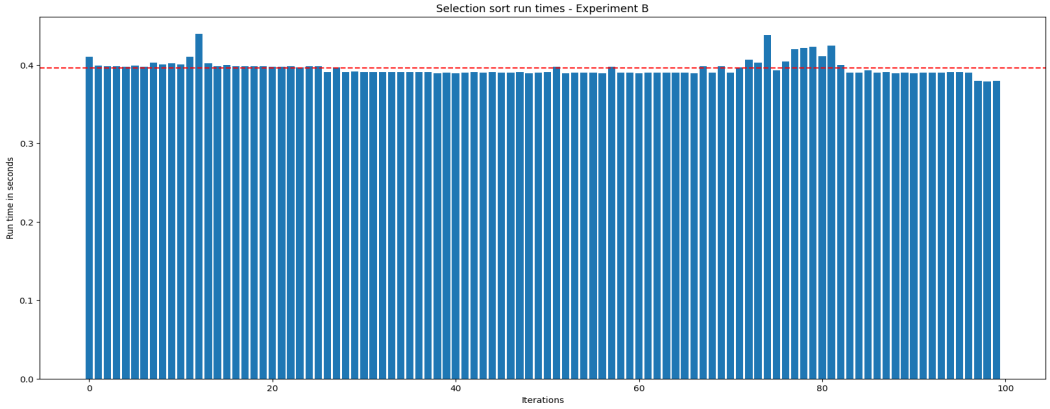



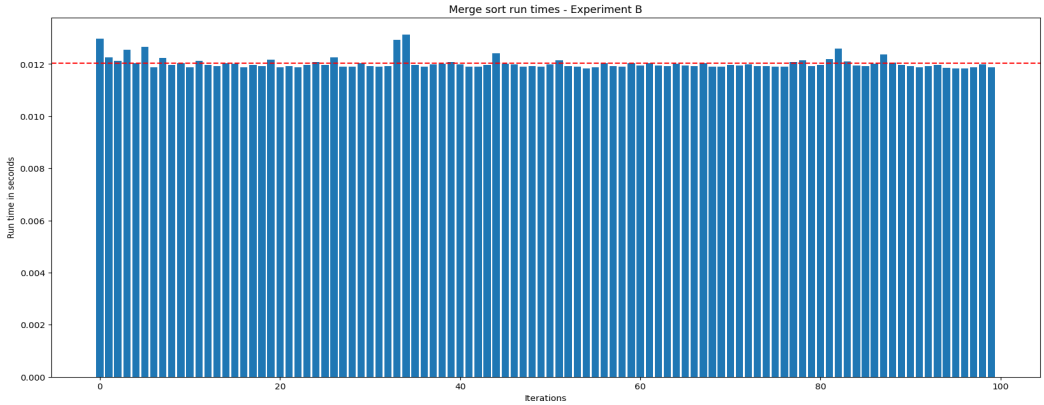



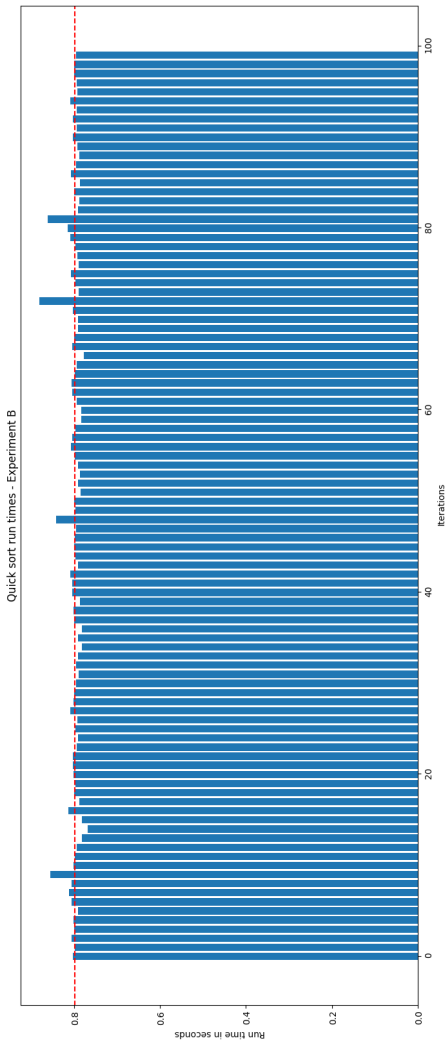
B. The sorting algorithms on a near-sorted list of 5000 items, 100 trials



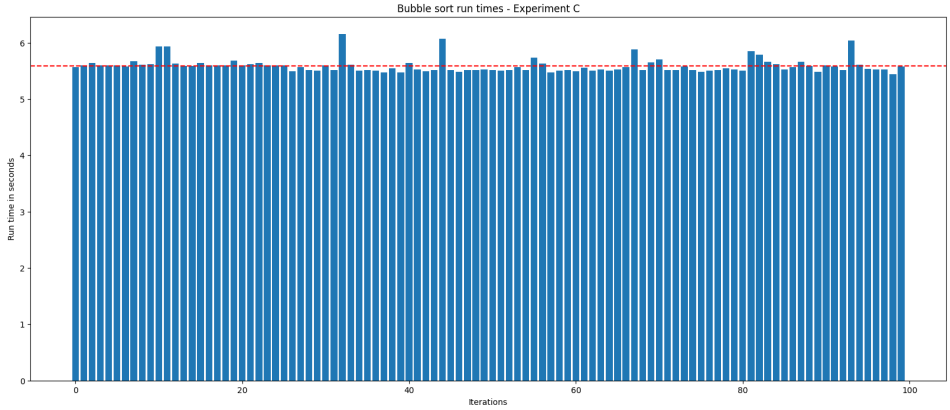



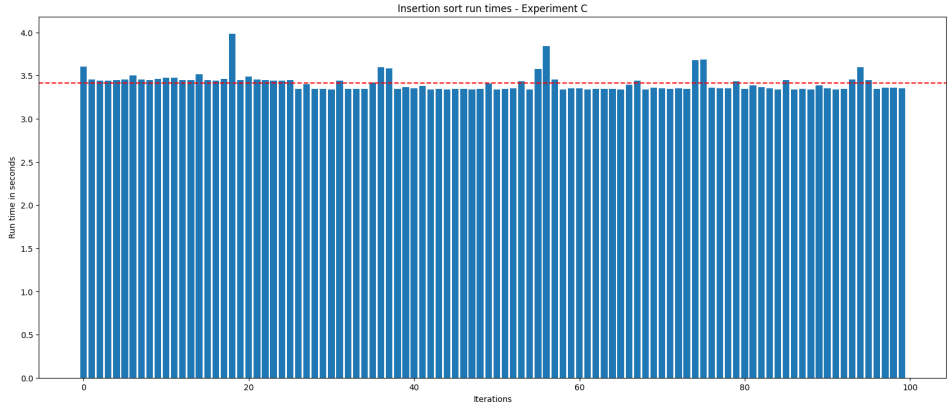



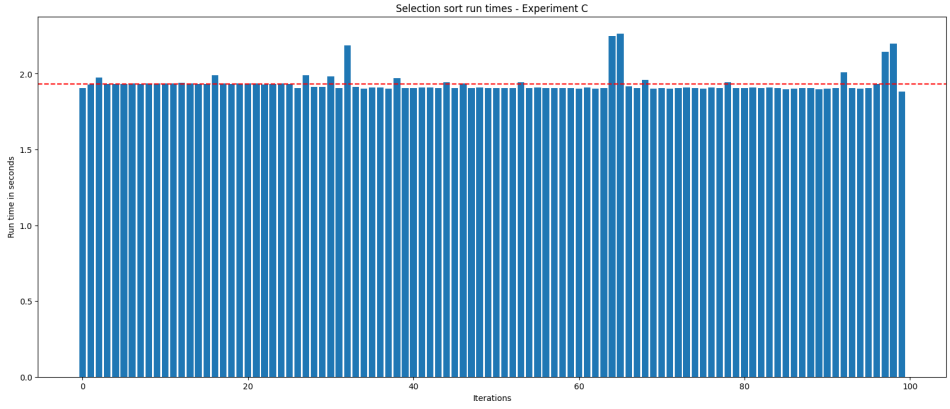



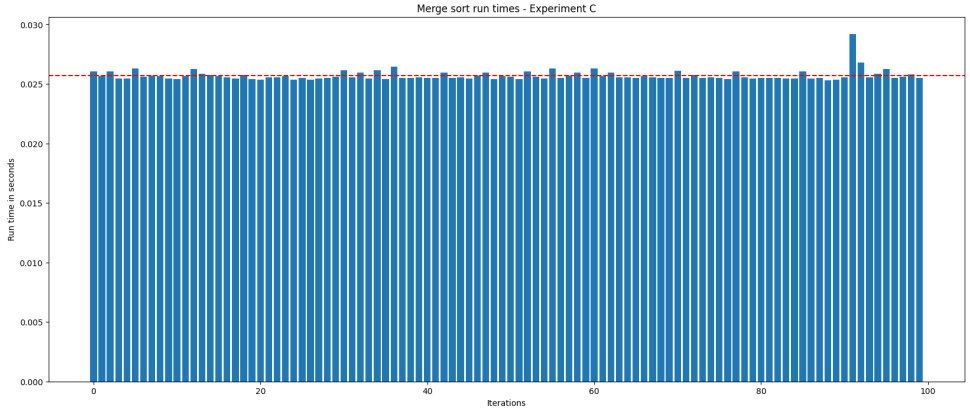



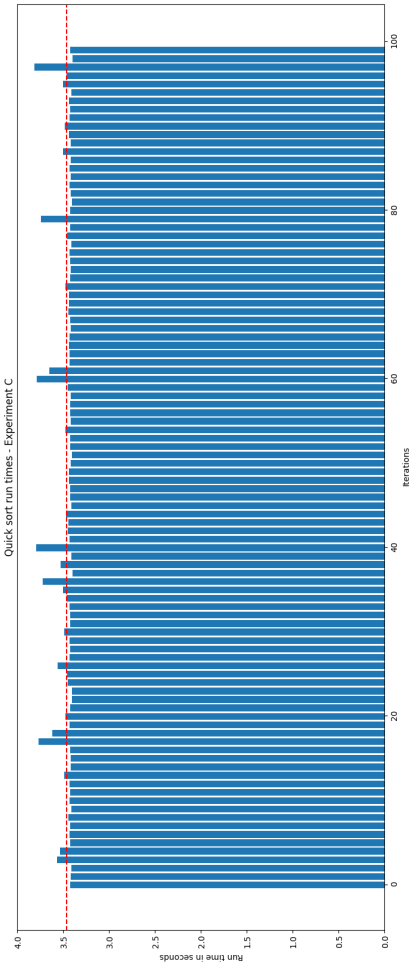
C. The sorting algorithms on a reverse-sorted list of 10,000 items, 100 trials



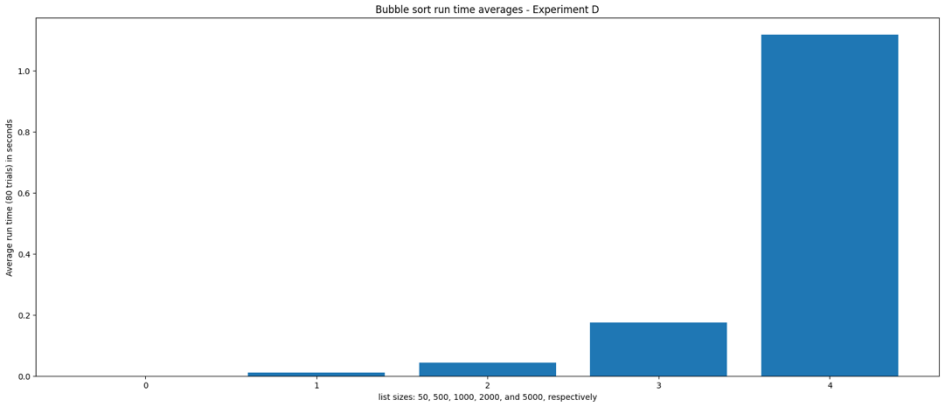



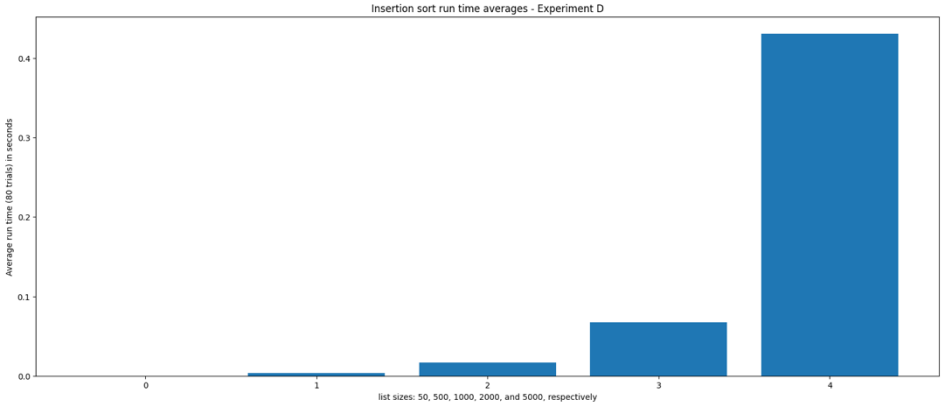



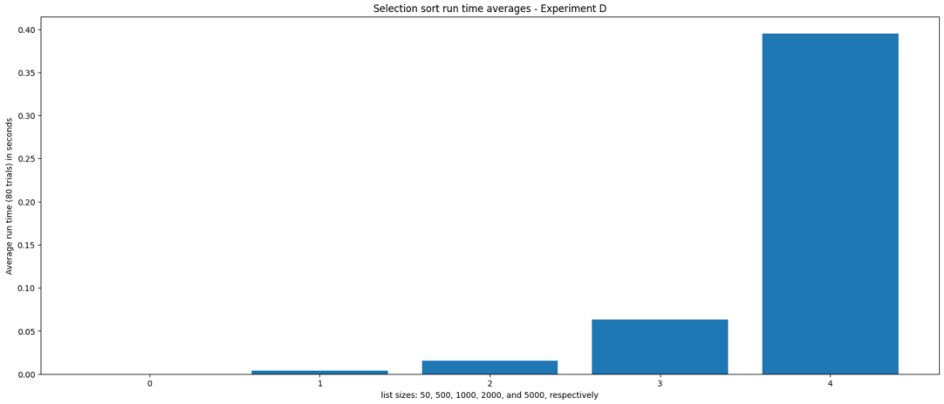



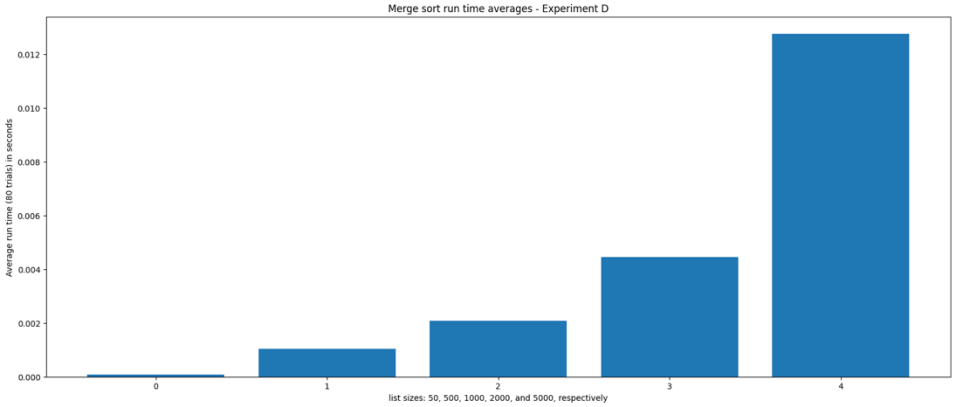



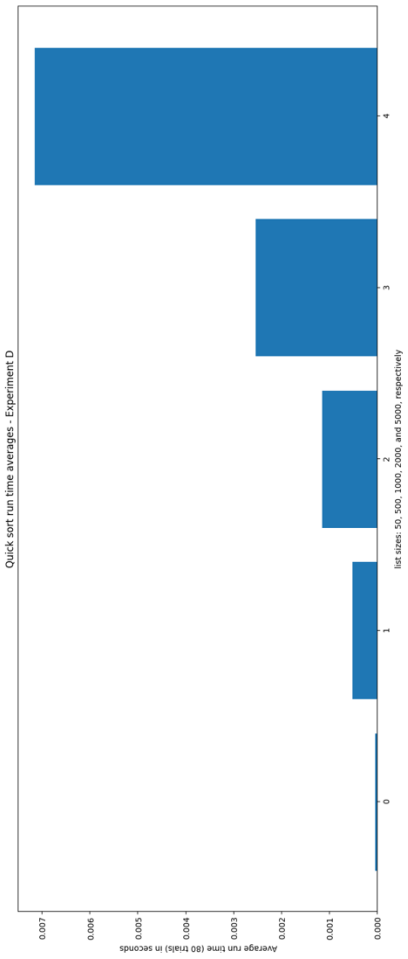
D. The sorting algorithms on variable-length unsorted lists, 80 trials



# E. The sorting algorithms on a reduced unique list of less than 5000 items, 100 trials