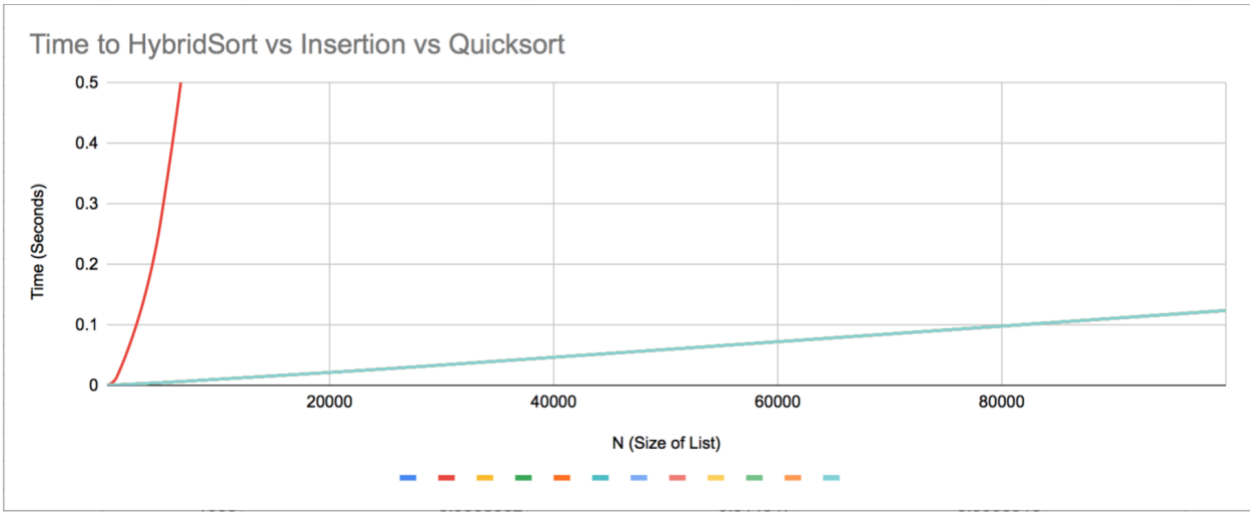
- *Hypothesis*: If we let *k* be the size of an array that, at or below *k*, sorting with insertion sort will, on average, improve the speed of a hybrid (quicksort and insertion sort) sorting algorithm, I believe that *k = 20* will produce the fastest results when compared to a baseline quicksort or insertion sort algorithm, and everything higher than *k = 20* will get progressively slower.
- *Methods*: I gathered data using the following method:
    1.) I defined a global constant integer named *K* which is the maximum value of k that we would test. For these tests, I set it to 100.
    2.) I tested my hybrid algorithm (defined in step 3 in terms of quicksort) with values of *k* ranging from 10 to 100, incrementing by 10 each time. I also made sure to test quicksort and insertion sort by themselves for comparison. Because we are testing the hybrid function mainly against quicksort, I set *N*, the number of items to sort, to 100000 to see how much faster this method would go than quicksort for a very large N.
    3.) I made new functions for testing a hybrid sorting algorithm. The hybrid functions were the same as quicksort, except in the quicksort function, when the number of data points between the lower bound of a partition and the partition index exceeded the defined *k* (between 10 and 100), an insertion sort was used on that partition of the array. Likewise, if the number of data points between the partition index and the upper bound of the partition exceeded *k*, insertion sort was used. This behavior is implemented in lines 142-158 of the source code provided.
    4.) After collecting the time it took for the hybrid algorithm to sort with *k* ranging between 10 and 100, (along with times for insertion sort and quicksort) I put these times into google sheets for further modeling.

NOTE: In the interest of not testing insertion sort and quicksort 10 times over for different k's (because k has no influence over the baseline insertion and quicksorts), I tested these only ONCE for comparison to the hybrid algorithm with ranging k.

- *Results*:

K	N	Quicksort (seconds)	InsertionSort (seconds)	HybridSort (seconds) K=10	K=20	K=30	K=40	K=50	K=60	K=70	K=80	K=90	K=100
10	100	0.0000769	0.0001801	0.0000721	0.0000707	0.0000712	0.0000704	0.0000707	0.0000708	0.0000773	0.0000701	0.0001405	0.0001404
	1000	0.0008582	0.014047	0.0006899	0.0006892	0.0006926	0.0006916	0.0006922	0.000689	0.0006923	0.0006909	0.0006885	0.0006912
	5000	0.0043697	0.2815954	0.0043194	0.0043206	0.0043199	0.0043202	0.0043195	0.0043193	0.0043181	0.0043194	0.0043181	0.0043202
	10000	0.009892	1.151196	0.009874	0.009891	0.009877	0.009872	0.009869	0.009869	0.009869	0.009868	0.00988	0.00987
	25000	0.02705	7.040498	0.027041	0.027104	0.027065	0.027054	0.02706	0.027059	0.027056	0.027042	0.027033	0.027057
	100000	0.123392	111.620991	0.123341	0.123714	0.123334	0.123449	0.12336	0.123528	0.123418	0.12384	0.123417	0.123403



The red line represents the time it took to sort a randomly sorted vector using Insertion Sort. All other lines represent the time it took to sort a randomly sorted vector using Quick Sort and Hybrid Sort using values of k ranging from 10 to 100. As the graph illustrates, Quick Sort and Hybrid Sort using different values of k were virtually indistinguishable.

- *Discussion*: What I found was that while N < 25, insertion sort always performed faster than quicksort and hybridsort, and at N = 25, the three seemed to take equal time. With N > 25, quicksort and hybrid sort performed the same, but always performed faster than insertion sort. It was surprising that Quick Sort and Hybrid sort produced nearly the same results. The crossover point for k for when Hybrid Sort became faster than Insertion Sort was the same as part 1.
- *Conclusions*: For N < 25, Hybrid Sort and Quick Sort were the same, but both slower than Insertion Sort for sorting integers, at N = 25 is where Hybrid Sort, Insertion Sort, and Quick Sort algorithms take roughly the same amount of time. For N > 25, Hybrid Sort and Quick Sort are the same and always faster than Insertion Sort, and get exponentially faster.