

Gary Huang

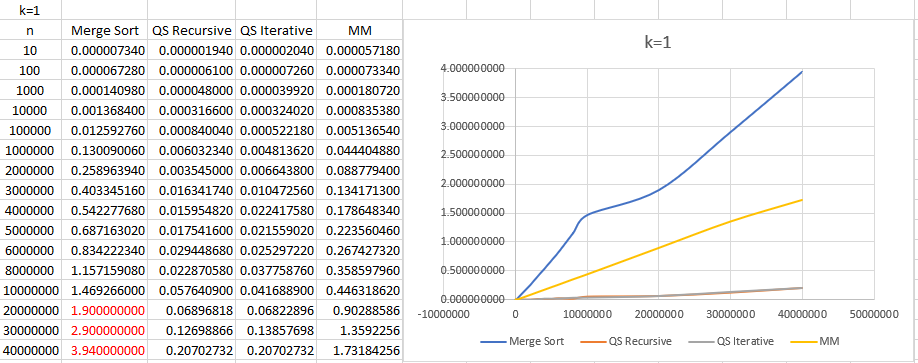
**Data Sets, Test Strategies, and Results**

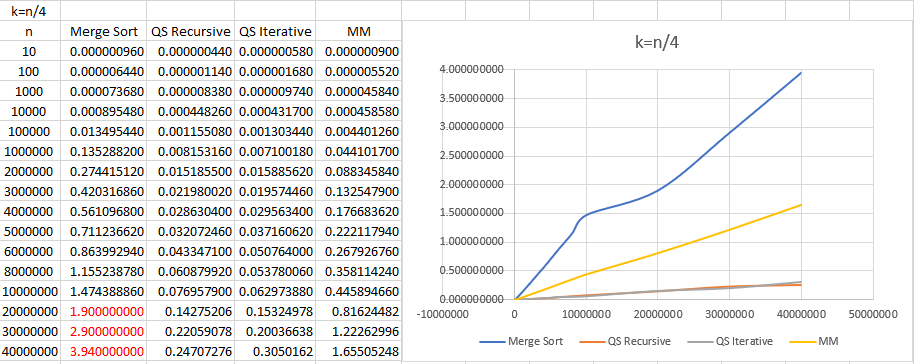
**Data**

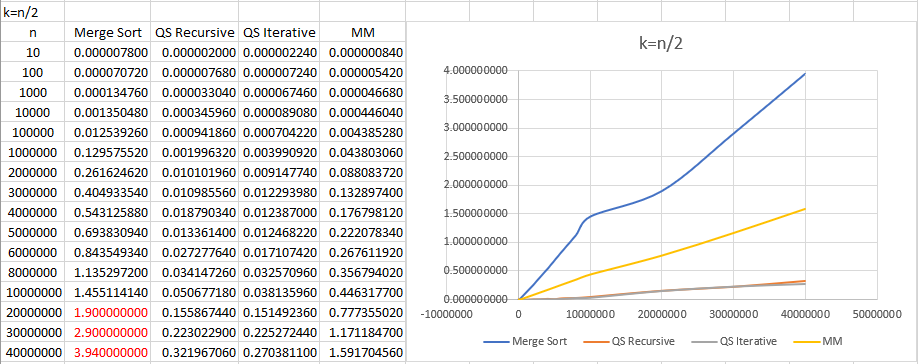
**NOTE: array values here are set between 1 and 10**

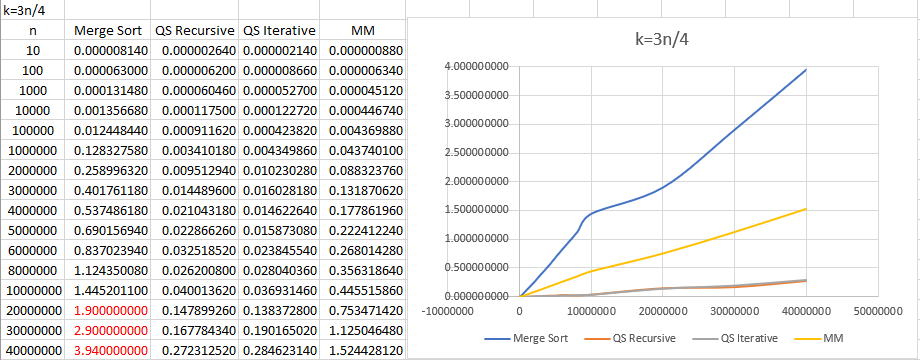
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| k=1 |  | | | |
| n | Merge Sort | QS Recursive | QS Iterative | MM |
| 10 | 0.000014300 | 0.000002880 | 0.000002760 | 0.000120580 |
| 100 | 0.000075520 | 0.000008880 | 0.000007800 | 0.000460480 |
| 1000 | 0.000197720 | 0.000094040 | 0.000099000 | 0.001492720 |
| 10000 | 0.001276440 | 0.000301740 | 0.000363340 | 0.024725280 |
| 100000 | 0.007930520 | 0.000840880 | 0.000961040 | 0.225106820 |
| 1000000 | 0.063165160 | 0.003344840 | 0.002937140 | Stack Overflow |

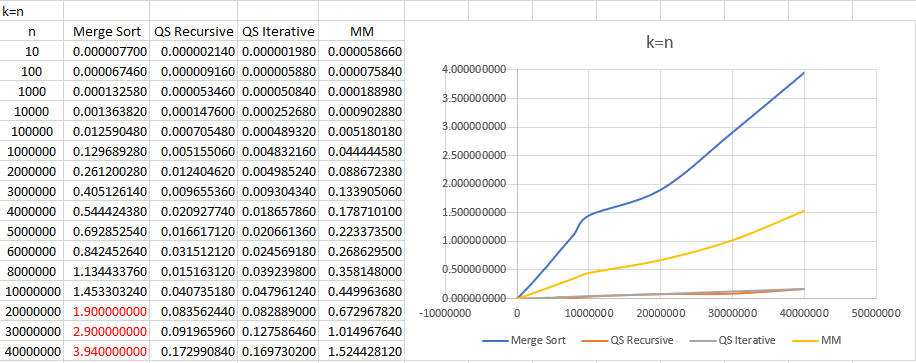
**Note: from this point forward, array values randomized with upper bound SIZE (array length)**

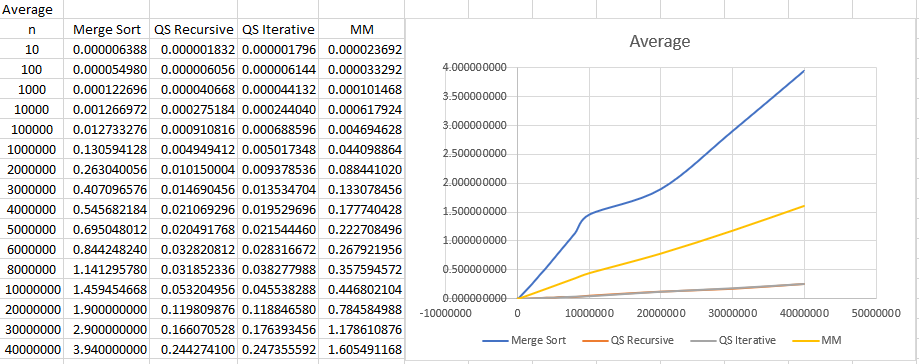


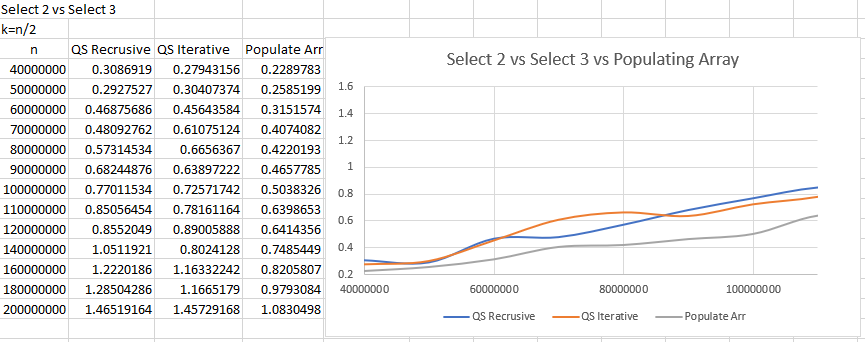


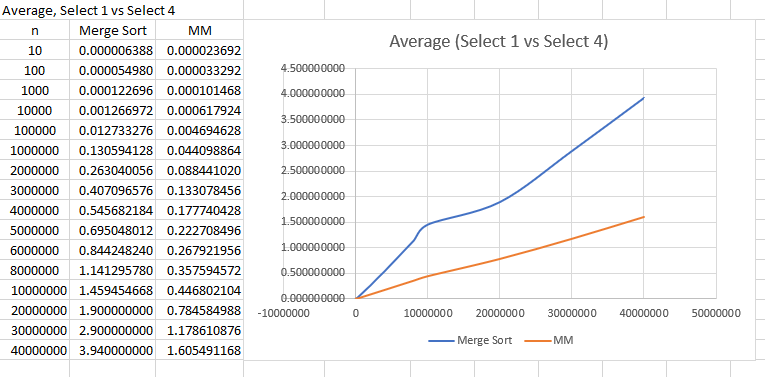












**Test Strategies**

Testing the program came with using different values for the arrays, choosing different k values, and size of the array. While I chose to utilize a more automated system of testing, I ensured that average runtime values were valid by testing each algorithm separately.

The program ran into a couple stack overflow errors while running due to values not being randomized enough. For instance, while using values of 1-10 in an array of size 10 million, I saw that there were constant overflow errors with no bugs that I could find. Upon changing array values to match the size of the array, I obtained no runtime errors.

One particular issue I ran into with Algorithm 1 (merge sort) was with system time overflowing. The values of the runtime got so high that I was unable to record proper run times and therefore I referenced Stanford’s study on their average runtimes for merge sort.

The program went through several runs to ensure that average runtime values were appropriate and made sense. Furthermore, I was able to reference Stanford’s report on their average runtimes to gauge the validity of my own program.

By choosing k values are both extremities and everything in between, the algorithms were put to the test in all possibilities that were recommended.

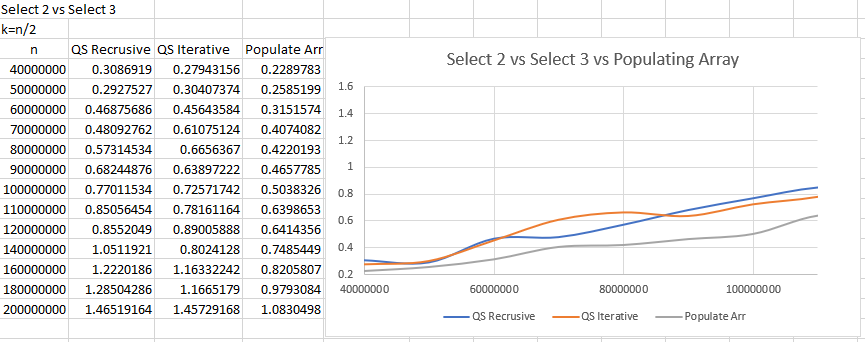
Testing select 2 and select 3 was done by having a separate test to push the limits of run time. Values went as high as 20 billion.

**Results**

**Select 2 vs Select 3**

Based on the results I gathered from running algorithm 2 (recursive) and algorithm 3 (iterative), I found that they tend to swap back and forth in terms of run times. Sometimes 2 would finish faster than 3 and vice versa. This is interesting because I have assumed that the recursive method would run slower than its iterative counterpart.

Looking at this average runtime graph, we see that there are multiple point were 2 runs faster than 3. This may be the case because the recursive portion is not nearly as expensive as our previous project when we dealt with matrices. Since these recursive calls are only responsible to running the original partitioning method call, this makes it so that the algorithm runs basically like an iterative solution. We see that in the iterative solution, there is a while loop that controls how often the partition function is called. Likewise, in the recursive solution, we see that the function will always be called based on whether the conditions are met or not.

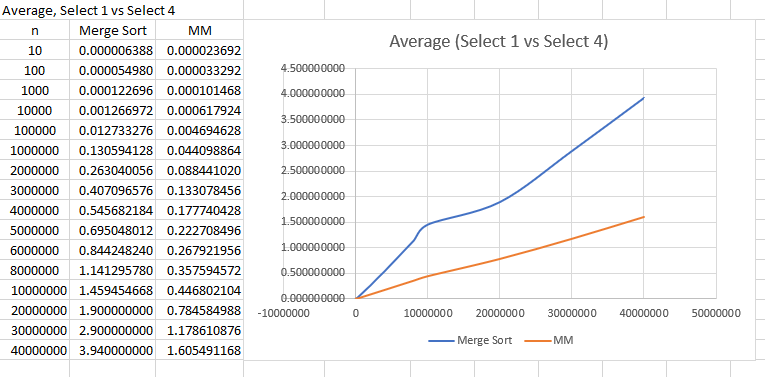


The above photo uses populating an array as a baseline of O(n) runtime.

Upon further testing, it seems like the cost of recursion begins to show ever so slightly around obscene values of 20 billion values, at which point the iterative method begins to run faster. Even if this is the case, we notice that previously the values were constantly weaving between each other, which leads me to believe that it is very possible that recursion could potentially get a faster runtime somewhere along the line.

This test proves that if recursion is used properly, and used for more simple tasks, the difference in run time is negligible.

**Select 1 vs Select 4**



Referencing the average runtimes of 1 and 4, we see that MM is clearly the faster algorithm. The graph itself also shows a linear line which matches the theoretical O(n) runtime. MM seems to be much more effective since we can guarantee a worst case O(n) runtime, whereas in our merge sort, we are given a O(nlogn) which is clearly, not as efficient.   
  
Because 4 uses a partitioning method that effectively picks a medians and then narrows its focus into a particular half of the array, the cost of running the partitioning method becomes less and less expensive, such that the sum of the entire runtime will be n. Merge sort is more naïve in its approach since we are requiring that the entire array be split which is log(n) but then upon merging, we have to run n times to combine everything before extracting our kth element.

I believe we could definitely improve our MM algorithm is we were to implement a hybrid version that included Quick Select; however, that still begs the question as to why we would ever use MM or a hybrid version since we are fully aware that Quick Select is just the overall better performer.