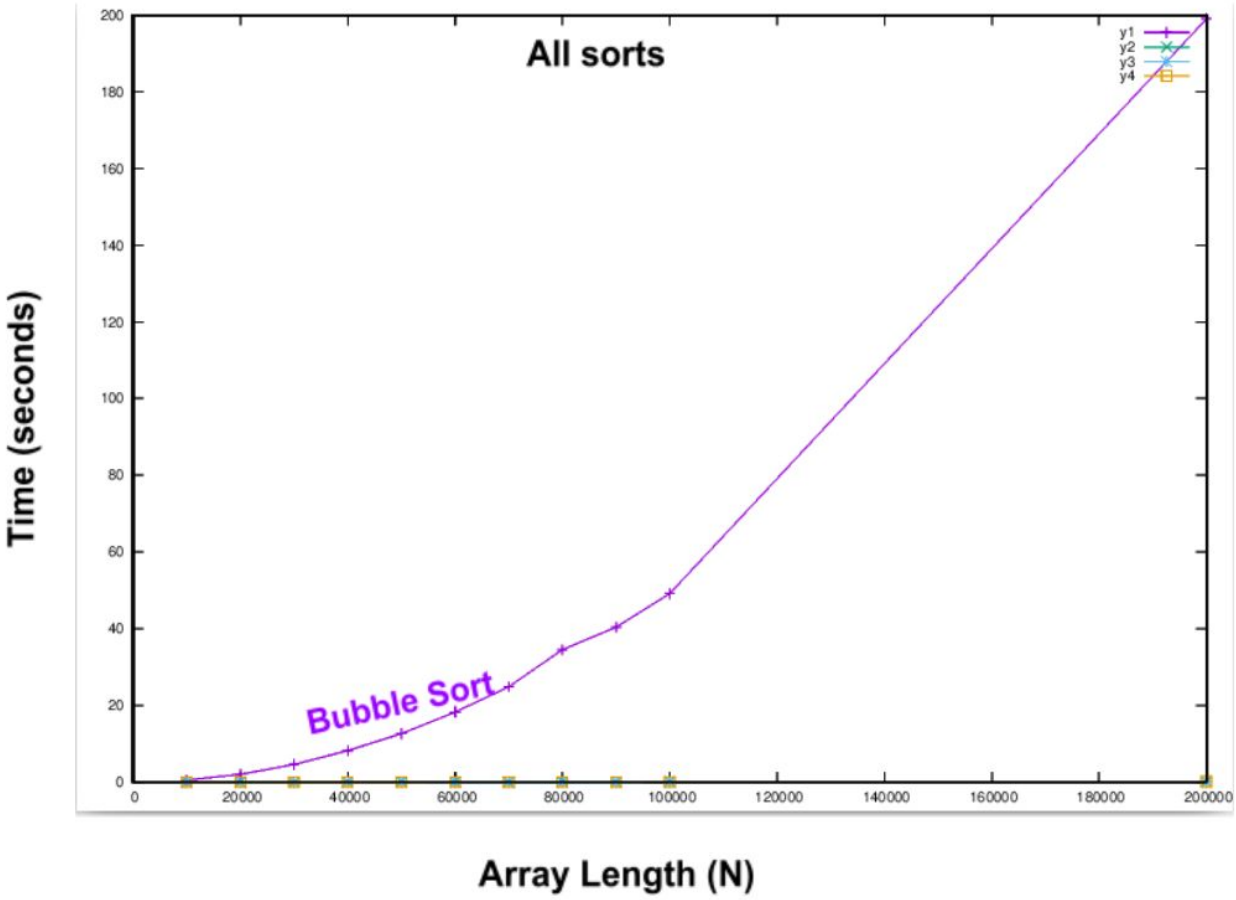
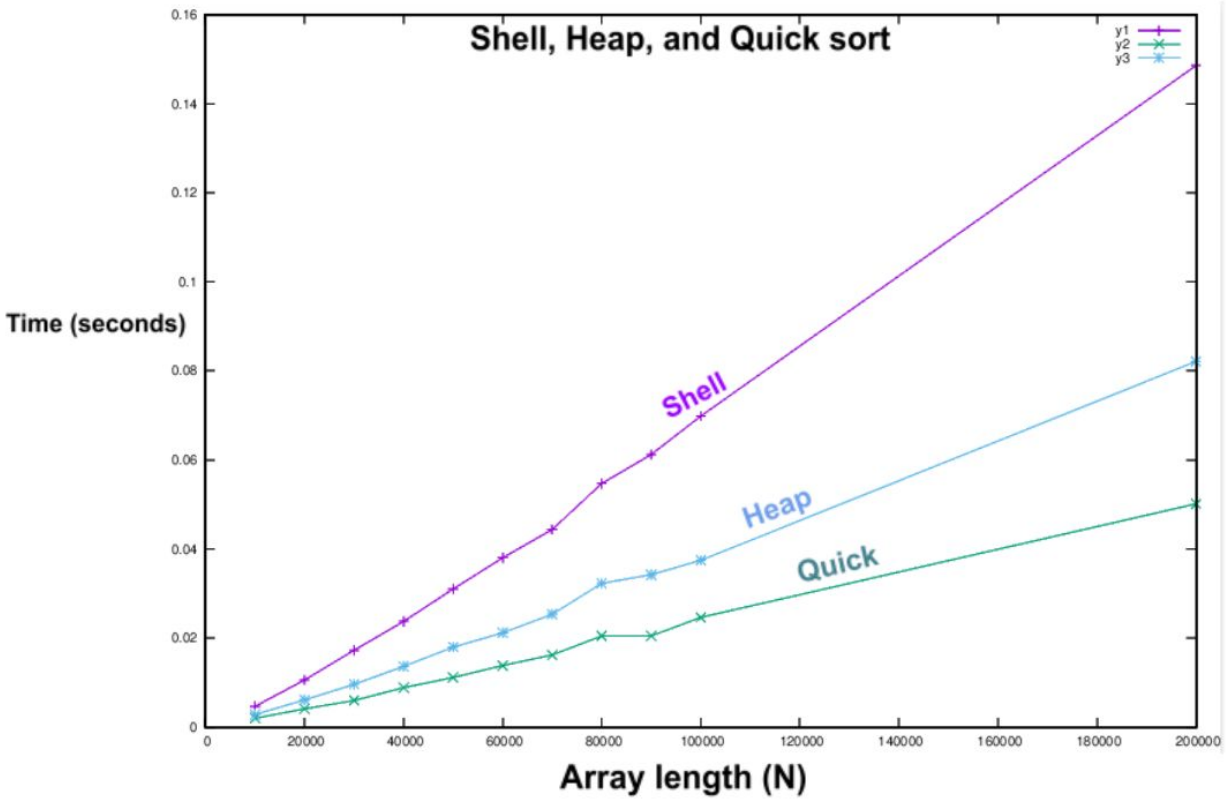
Bubble Sort

In bubble sort's best case scenario, there will be a single iteration with 0 swaps, implying that no two indices are inverted and that the array is already sorted. The best case then reduces to O(n), or linear time, because twice as much data will take twice as much time, etc…. Now in time complexity theory constants are dropped because we are more concerned about the scalability of code but in practice it can matter much more. The constant is reduced by 1 per iteration because the amount of items we iterate through decreases by 1 each loop, which pretty much halves the number of comparisons. Worst case is very ugly however, displaying O(n^2) time. If the array is unsorted, even reversed, we will be iterating through the second loop until (n-1) = 0. Bubble sort degrades in value exceptionally fast, showing its parabolic-like attributes very early on. At around the



10,000 item mark it begins diverging from the rest of the sorts. I've gathered that bubble sort is extremely inefficient and should ONLY be used in lists that are nearly sorted or very small. Randomly

inserting 1 or 2 values into an array and sorting with bubble outperforms and takes less compares on average, (more moves of course). Also, an array of around 10 or less elements produces less compares on average than most other sorts. So bubble sort has its uses, and should only be considered in these cases.



Shell Sort

Shell sort's best case is O(n), or in some cases O(nlog(n)) time complexity. The inner for/while loop will not be entered because no values will be out of order, thus it will loop for each gap until its < N making no comparisons. Now for the average case of shell sort it varies depending on the gap sequence, but on average we can say O($N^{\frac{5}{3}}$). Shell sorts are great for moving drastically out of order items early on and clearing up work for the last gap, aka the insertion sort property. The constant really depends on the gap sequence, which transitions into: What about the worst case for shell sort? That solely depends on the gap sequence. On large sequences of data shell sort begins to lag behind, especially around the 100,000 item mark.

Many gap sequences have been tried and tested, the problem lies in finding a happy medium. Too many gaps can create redundancy and too little will really slow the process down, relying on gaps closer to insertion sort for the bulk of sorting. I found this video very insightful for understanding how gap sequences have evolved from Donald Shell's in 1959 to Ciruas in 2001, and the consequence having a good good gap sequence. For instance we are using the 3-smooth gap sequence which on average is more than double a Cirua sequence. Shell sort, like bubble sort, should be implemented for small data groups. It begins diverging from quick and heap sort once they get into large data sets. Since its implementation is fairly simple, it seems like a go to option for small data sets, especially if the array is reversed. Its comparison count also diverges pretty early on at around n = 20, but between 10 and 20 its comparison count beats bubble.

Quick Sort

Quick sort, in practice running the fastest out of most sorts, has a best case of O(nlogn), average case of O(nlogn) and a horrendous worst case of O(n^2). I really don't consider the quicksort worst case possible if quick sort is implemented by choosing the median or average value of the array as the pivot, that way the pivot won't be located at either end. However if the pivot is located at the end then, slowly but surely, the array will continue to become partitioned with no i and j swaps. In its best cases however, the divide and conquer aspects of it really shine, and perform really well on random or reversed arrays. And of course once array size scales its comparison count outperforms all other sorts and also outperforms as far as time taken. If linear time is like : for every x increase of 2, y increases by 2. For nlogn, it's like, for every x increase of 2, y increases by 2log(2), or .6. This is EXTREMELY helpful for scaling up to large data sets and should be chosen as a great default sorting option. Quicksort has a constant, k, which varies depending on the number of terms located below the pivot. We can choose an ideal k for the Quicksort formula : T(n) = T(k) + T(n-k-1) + O(n), where T(k) is a recursive call and T(n-k-1) is a recursive call (O(n) is the partition process). A good k should be at index n/2 ideally, so each recursive call takes T(n/2), or 2(T/2) when combined.

Heap Sort

Heap sort has an average time complexity of O(nlogn). Its formula takes O(logn) time to create a max heap, and O(n) time to build the heap. There is a constant value associated with O(n) -> O(nk) where k represents the distance a value is from its ideal position. If an

array is reversed and we use a max heap then no swaps will be necessary, but it will still consume O(n) time. As n values scale, heap performs close enough to quicksort to be considered in practice, definitely beating shell sort. Its comparison count also clings close to quicksort, however never really quite beats it on average. Something interesting I noticed was the amount of moves it performs compared to quicksort on average… most of the time more than triple quicksort's move count, though their comparison count remains similar enough until larger data in which quicksort once again outperforms. This probably has to do with the repetition of building the heap and moving the max value to the top, comparing it with all the necessary values.