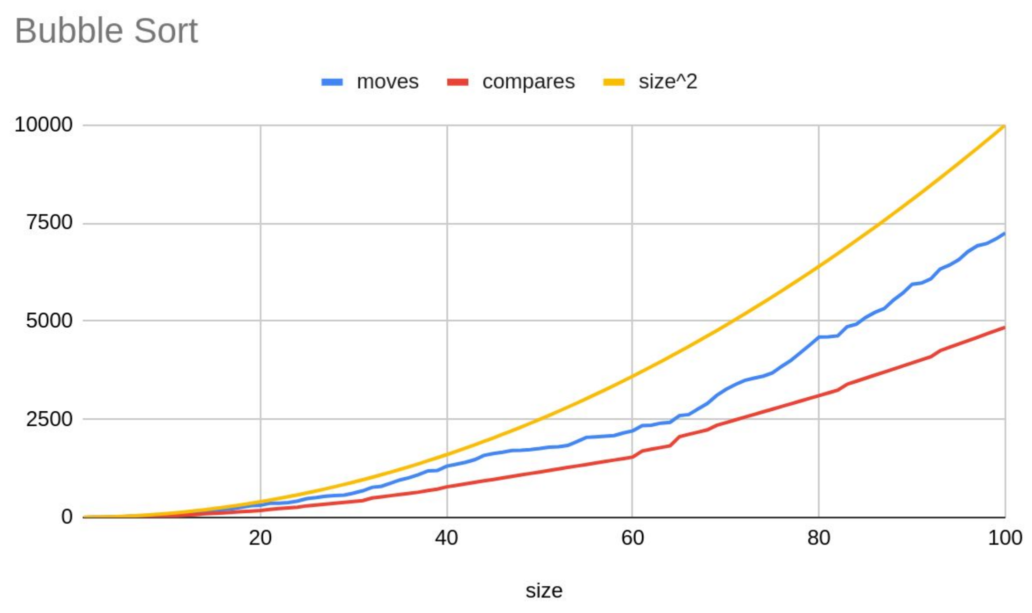
WriteUp

In this writeup we will be analyzing what I learned from different sorting algorithms.

Bubble sort

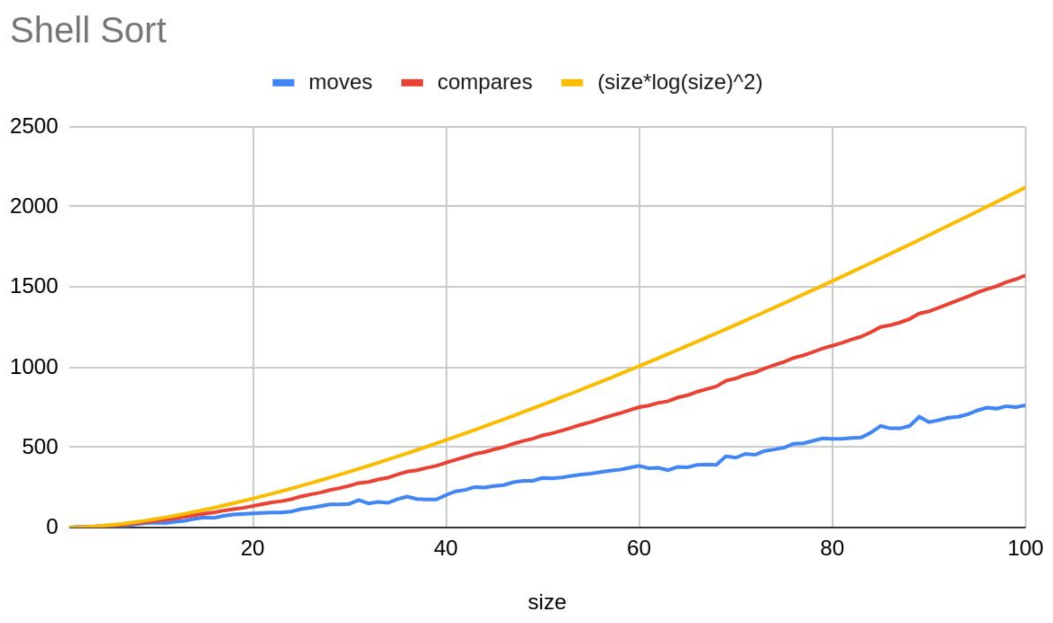


Bubble sort was the most simple to implement, but is incredibly inefficient as it's time complexity it's worst case time complexity is $C(\frac{n(n-1)}{2}))$ this makes sense because if none of the elements are in the right place when it's their turn to float to the top, the computer must go through the array n times and in that array, it must compare itself with another element $n-(whatever\ iteration\ we\ are\ on)$ times as well. So the complexity ends up being: n, n-1, n-2, n-3, … , 1. The last element is 1 bc n-(n-1) is the length of the last iteration and is equal to 1.

The best case for this sort would be O(n). I figured this out while working out this algorithm on paper. While this is very good, it is extremely rare. Why? Well because everything must already be sorted. If nothing gets swapped on the algorithm's first pass through the array. It stops and that is that. I also tried making a reversed array and passing that in. This led to the worst complexity. After examining these scenarios, it became clear that bubble sort is fine, only if you have a nearly sorted array.

As you can see in the graph above, bubble sort's algorithm followed the n^2 graph quite well. This makes sense since the average case of bubble sort generally matches it's worst case. This makes sense since on average, bubble sort will have to go through almost every loop since it just floats the biggest number to the top. The reason for the bumpiness in the graph is, I'm pretty sure, because sometimes the smallest elements are already in order somewhere in the array and the array can stop early if it goes through without swapping. So, since bubble sort only brings one element to the top without altering the other of the other's, the elements stay together till the end and the algorithm stops early.

Shell sort

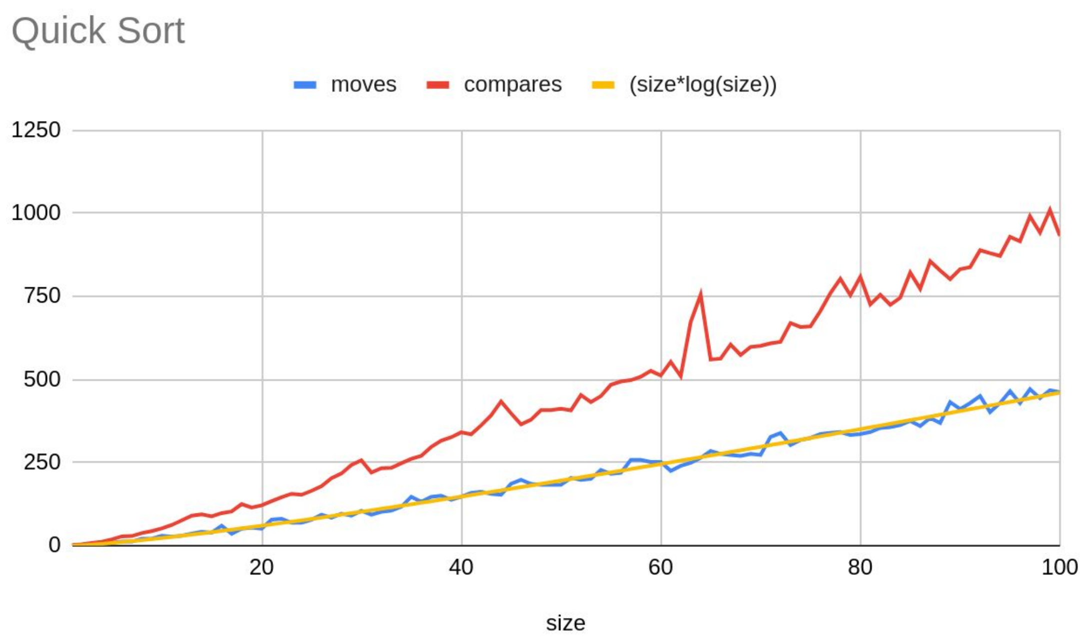


Shell sort was the first sort that I had trouble understanding at first. But in the end, I saw that it was basically an insertion sort with gaps between the indexes being checked. This sort's worst case time complexity is $c(n(logn)^2)$. I was very surprised to see how much the time complexity fluctuated between gap size sequences.

For shell sort, when the numbers were reversed in the array, it took a lot more time to sort then when they were randomly arranged. I guess this is because it works kind of like insertion sort and it needs to scale across the entire array for every element if they are all on the opposite side.

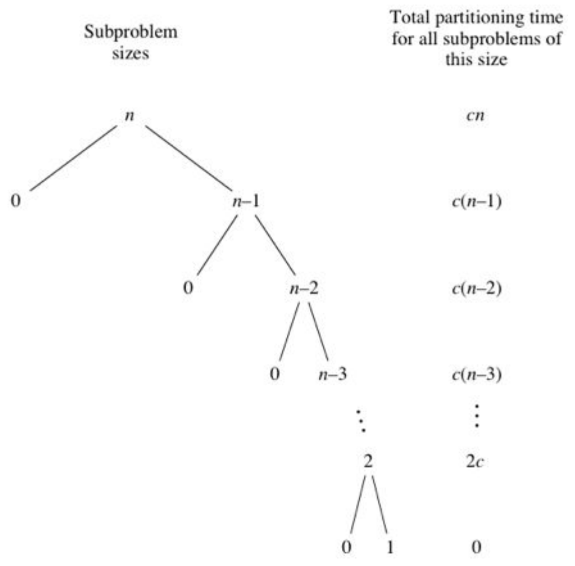
I was surprised how smooth the comparison graph was for this sort. However, when I thought about it, it made sense since the algorithm compares a number until it finds its right spot, which means that it has to scale a pretty constant amount each time.

Quick sort



Quick sort's algorithm created a very jumpy graph! I suppose this is because of the importance of a good partition. If a good partition is chosen, the sub array will be split pretty evenly each time. This is good because less iterations within each new partition need to be done in total to sort the array.

Quick sort's worst case time complexity is n^2 when removing the constants. With the constants it's c((n+1)(n/2)-1). The diagram that helped me understand this time complexity most was this one from khan academy



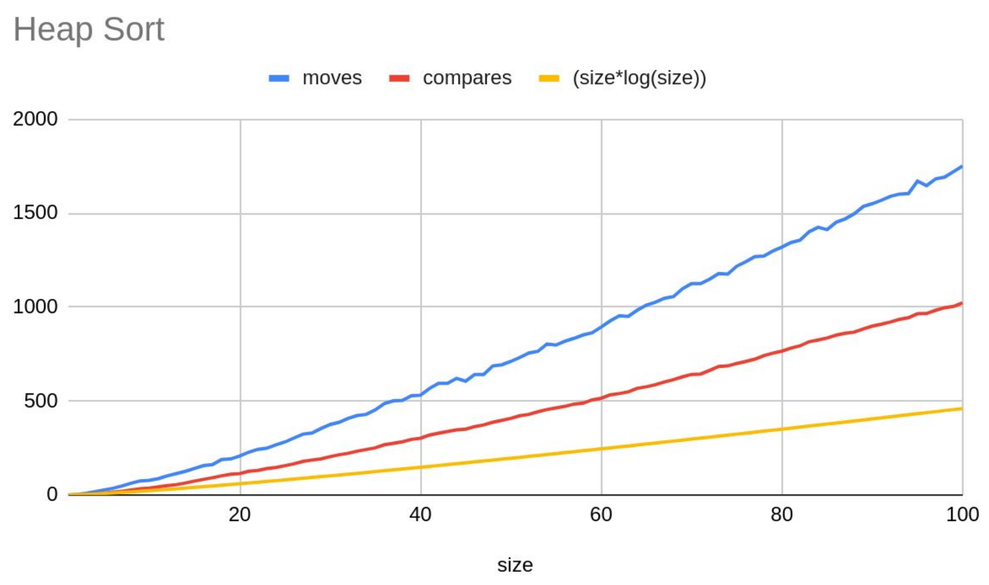
link: https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/analysis-of-quicksort

Like bubble sort, this algorithm also is doing n, n-1, n-2,... but the difference here is that it ends at 0. That is because there is nothing to sort at the final partition because the size of the final sub-array is 1.

I believe that quick sort is very fast in general because it doesn't have to iterate over the entire array over and over again. Instead it goes through once and sorts small arrays until there is nothing to be sorted.

At first I was quite worried about the worst case scenario of quick sort, but after testing out different seeds, I saw that the worst case scenario is extremely rare. One way of getting the worst case scenario is to pick such a horrible pivot every time that it separates the array into 2 sub-arrays, one is at length 1 and another at a bigger number(except for the final split). However, this is so rare that we shouldn't need to worry about it and we can use this algorithm a lot.

Heap sort



Heap sort seems to be the smoothest graph of the bunch. I suspect this is because it creates a heap, fixes the heap, and then puts the heap back into the array in the correct order for every array passed into it. Since there is not much reliance on luck in this one, unlike quick sort, heap sort has a much more predictable time.

Heap sort's time complexity is c(nlogn). This is very good! I'm pretty sure this sorting algorithm has the best formula since even in the worst case, it retains a pretty good time complexity.

I tried so many different types of arrays: reversed, perfect, and random. But they all fit the graph shown above pretty well.

Conclusions

Before starting this assignment I had no idea what the best kind of sorting algorithm was. I didn’t even know how to bubble sort! Now, not only do I understand these algorithms, but I know which ones are best. If I were to sort an array in the future, I would go with heap sort as it is reliable and fast.