Asgn3 Writeup by Derick Pan
Dpan7
Cse13s

Bubble Sort:

Time Complexity:

To calculate the time complexity for Bubble sort, we can follow the code I implemented off of the pseudocode from Dr. Long. We need to do N iterations through an array and begin swapping. If after the first N iterations and the list isn't sorted yet then we'll do it again, then again, then again. But, each time we do it again, we Decremement N by 1 because the last item in the array is already in sorted order. If we were to put that into writing, it would look like this:

First + second + third + fourth

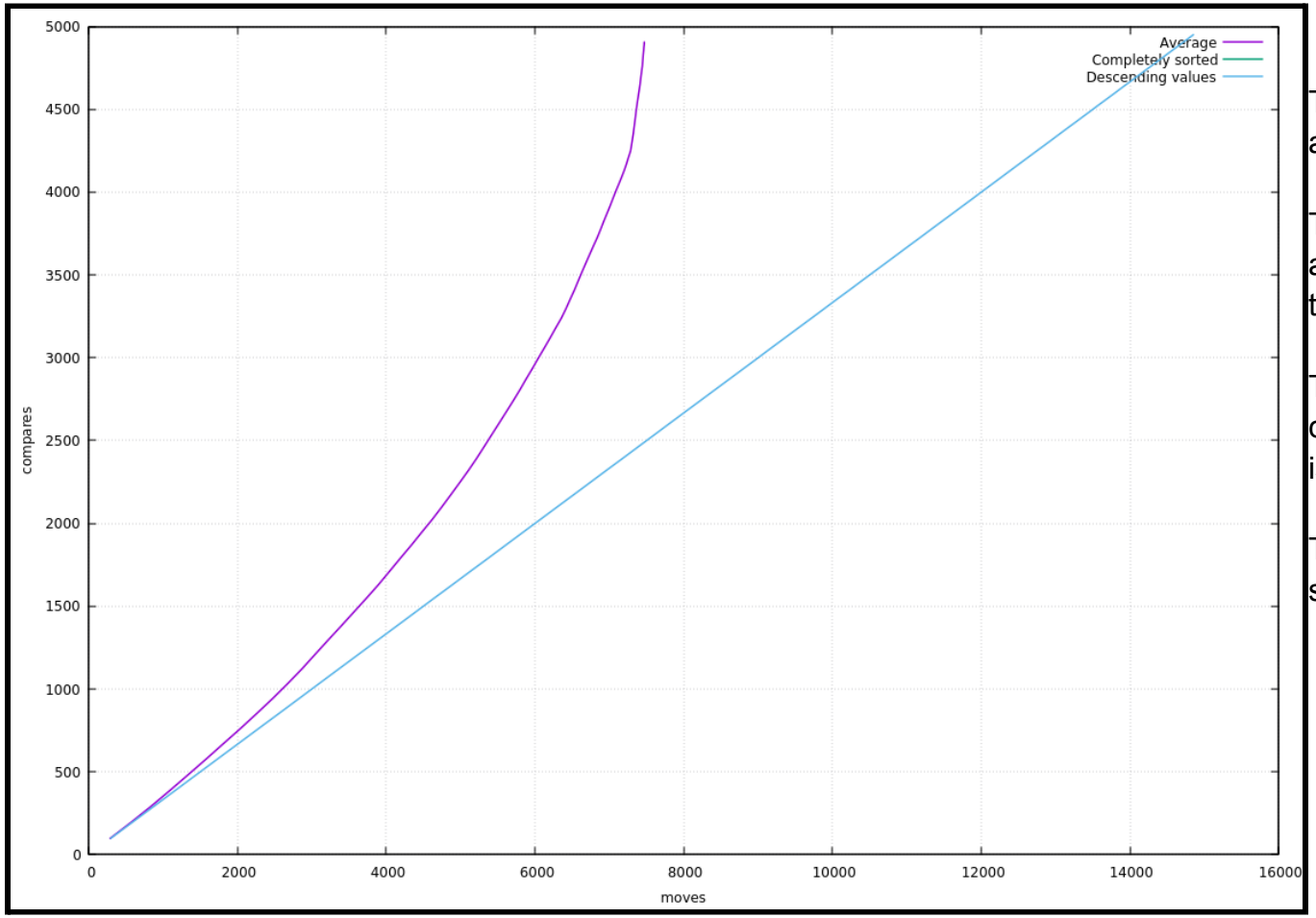
N + (N-1) + (N-2) + (N-3) + ……. + 3 +2 + 1 Summing up to the formula N(N+1)/2 and having a time complexity of O(n^2).

The Constant from the time complexity of bubble sort won't change even if we have an ALMOST perfectly sorted list, or a list in complete descending order. This is because Bubble sort will ALWAYS iterate through every element no matter if the list is in order or not, assuming that that a list is perfectly in order, except for one element being in the middle.

We still need to do N iterations for the first one, N-1 Iterations for the second iteration, and N-2 for the next and so on until there's an iteration where no swaps occurred and the loop could terminate.

Something interesting bubble sort taught me about Time complexity is the time complexity will stay as O(n^2) because it couldn't turn into log(n) since the amount of sorting does not shrink logarithmically like how the quicksorts do using Partition.

Bubble Sort Graph



The x axis is the amount of moves, and y axis is the amount compares.

The **purple line** represents The average amount of moves and compares needed to sort an array of 100 elements.

The **blue line** represents the WORST case scenario where a 100 element array is in descending order.

The **Green line** represents a completely sorted 100 element array.

As stated above, the average time complexity for bubble sort is O(*n*^2). No matter the order of the array, bubble sort will always have to compare N(N+1)/2 elements. Sorting a random array using bubble sort will always generate a similar exponential graph, whether it be 100 values, or 10,000 value

Shell Sort:

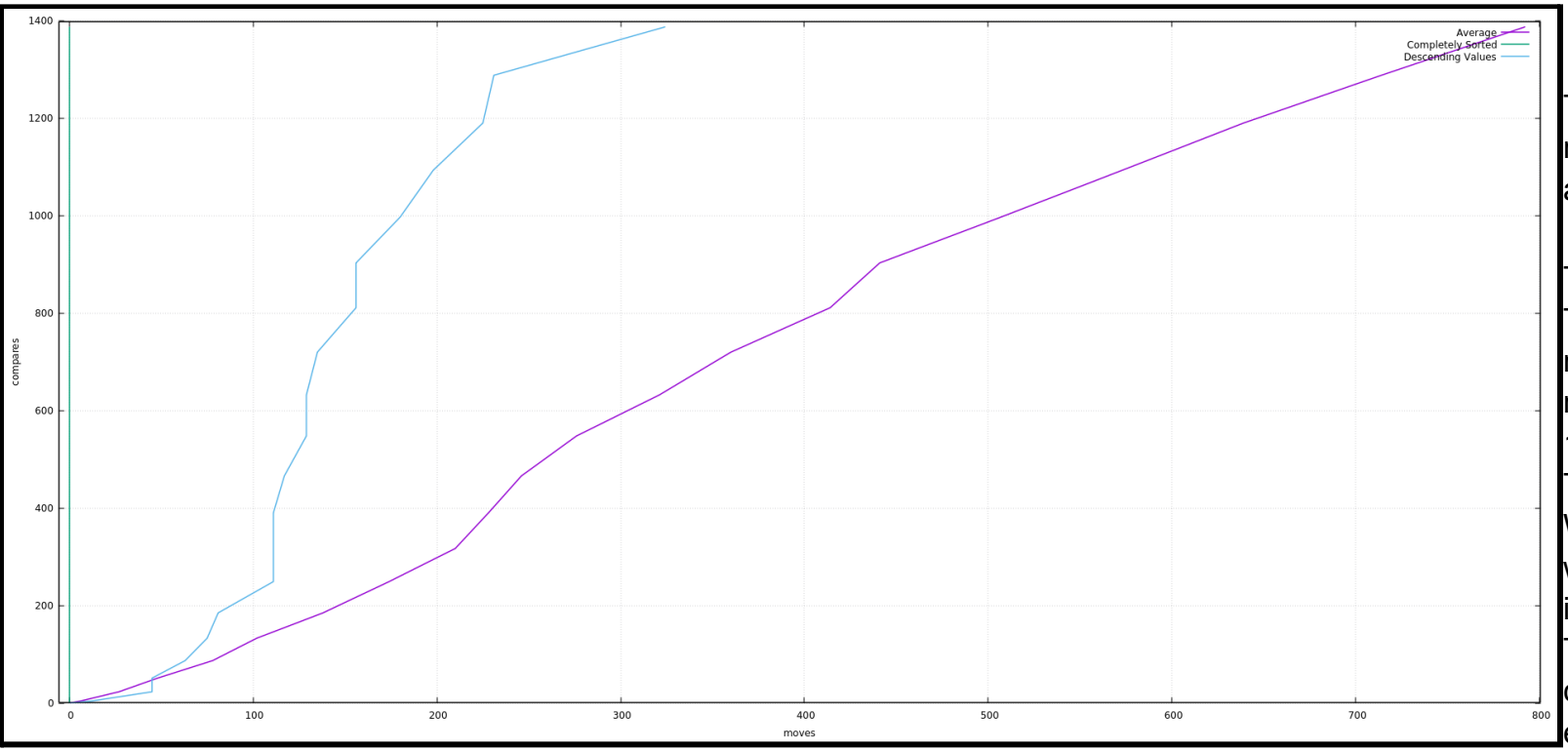
Time Complexity:

Shell sort doesn't follow the same way to identify time complexity as bubble sort does. As this write up isn't about explaining what shell sort does, I will directly be talking about the time complexity regarding Shell sort, specifically using the provided gaps from the github repo, which is also the Pratt sequence.

The following time complexities are from: *https://www.programiz.com/dsa/shell-sort*

- My Shell sort works by Starting from the largest gap and shortening the gap after every iteration. (Sequence provided by Dr.Long)
- With each gap, I test if that current iteration's array value is smaller than the gap value being tested. If it is then great, we swap, and Subtract change the length of our gap respectively.
- The best case time for Shell sort is O(*n*log(n))
    - The reason for the average and best time complexity being O(*n*log(n)) is because the array is already sorted, and the inner while loop never has to be iterated over. As our increments aren't constant we can't have a time complexity of O(*n*).
- Worst case time for Shell Sort is O(*n*^2)
    - In the worst case scenario, our inner while loop needs to compare and swap nearly every value there is, and because that while loop is enclosed by two for loops, it's possible for that while loop to iterate over (*n*^2) the amount of gaps and values. So that is the worst case time complexity, and the constant makes sense; the while loop is under two for loops, two for loops creates a lot more values for while to compare.

Shell Sort Graph



The x axis is the amount of moves, and y axis is the amount compares.

The **purple line** represents The average amount of moves and compares needed to sort an array of 100 elements.
The **blue line** represents the WORST case scenario where a 100 element array is in descending order.
The **Green line** represents a completely sorted 100 element array.

In the best case scenario (Green line), shell sort doesn't need to move and values around because it's already perfectly sorted. Of course there's still going to be as many comparisons as the other lines because the function still needs to iterate over everything and check all of the values. This similarly reflects the Time Complexity O(*n*log(n)) as the amount of comparisons we need to do is still logarithmically correlated to the function.

Now we have the average case, the **purple line.** On average, shell sort won't be sorting perfect lists nor will it be sorting lists that are in the complete wrong order, it'll be somewhat proportional to have half of the elements sorted and half of the elements not sorted. So, we can see that for every hundred moves, there's about another 200 comparisons. This linear correlation is because the gaps used in shell.c follow the formula N/2. So with every iteration, we see the gaps getting smaller each time.

Now the WORSE case, the **blue line**. We have a list in complete descending order, and the graph looks like it's going all over the place. Although, it kinda looks like an exponential graph, which is also the time complexity of the WORSE case: O(*n*^2). With a nested while loop inside of two for loops, it's clearly exponential behavior. If the array size was larger, then we'd see a sharper exponential growt

Stack and Queues

Time complexity:

- The following time complexities are from "*Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People*." and "*http://ssp.impulsetrain.com/big-o.html*"
- **Both Stack and Queue's time complexities are similar** in this situation because of how similar the structure of loops are implemented.
- In my version of the Quicksort, I'm using an iterative Quicksort rather than a recursive quicksort. Both of these methods are pretty similar beyond the part where I need to allocate/deal with my stack with helper functions rather than the recursive quicksort.
- Best time Complexity: O(*n*log(n))
    - nLog(n) found as a time complexity was found by thinking about how the partition function works. When we first use "partition," it splits the array into two parts, one part greater than the other array.
        - In the second the pass "partition" does the same thing, but this time it's ½ the size. In the third pass, it's ½ the size again, but with the previous partition. So even though there's another partition, it will get smaller and smaller after each iterative sort.
    - Therefore, we have the time complexity of log(*n*), everything is proportional to the original size.
- Worst case time complexity: O(*n*^2)
    - Similar to all of the other algorithms, we must ask ourselves, "how many inner loops are there?" and is the time complexity quadratic?
    - Let me make myself more clear, If there are 100 items in the array we need to sort. We'll need to do 100^2 operations, (Worst case), because of how partition has several while loops.
    - Therefore, looking at the inner nested loops, we can find the time complexity.

Stack and Queues

How big do the stack and queue get? Does the size relate to the input? How?:

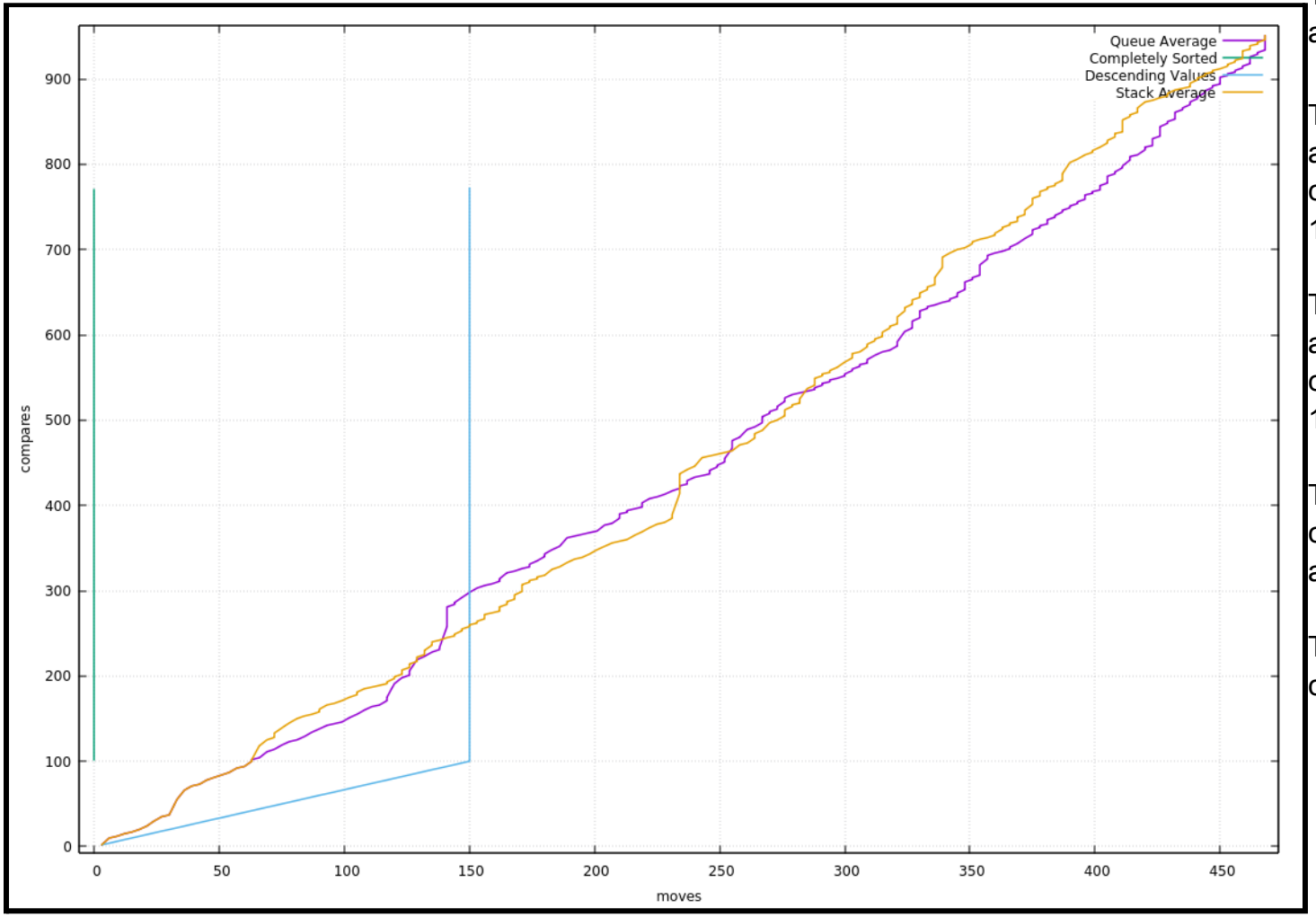
Stack:

The Stack size never gets bigger than the size of the input, because of how stacks only add and remove from the top of the list. We physically can't add more items into the stack because there are no more items to add onto the stack. (Unless we add already sorted elements, which would be pointless) Furthermore, after each time partition is called, we split the amount of items we need to work on by half. We can only add so many items onto the stack before our list is already sorted. The size of the stack will be logarithmic to the list we need to sort.

Queue:

Similarly to the Stack, Queue will grow logarithmically. At first thought it wouldn't make sense, queue should grow linearly large because a queue is like a line, it'll just keep on growing and getting bigger. But, no that isn't the case for my implementation. I implemented a circular queue rather than a linear one. Doing so, my queue doesn't get extremely large, it instead stays the same size and works in a circle.

Stack and Queues



The x axis is the amount of moves, and y axis is the amount compares.

The **purple line** represents The average amount of moves and compares needed to sort an array of 100 elements using Queue Sort.

The **orange line** represents The average amount of moves and compares needed to sort an array of 100 elements using Queue Sort.

The **blue line** represents the WORST case scenario where a 100 element array is in descending order.

The **Green line** represents a completely sorted 100 element array.

I decided to Merge the Quicksort graphs because of how darn similar both of them are. I'm not surprised to see how both Queue and Stack share a similar graph. Since they both have similar structure, share the same Partition function, and have the same time complexity, I'm not shocked to see the similarly.

In this graph, both queue and stack are sorting the same random 100 item array. There isn't anything too special about the average array to sort. But, there is something unique and rather odd about sorting the WORST case array. The first time I produced this graph, I thought that there had to be something wrong with my values because things just didn't make sense. Then I tried it again with no resolve. So i thought about it…

❗THEN IT CLICKED❗

Quicksort takes use of a Partition function, that takes an element as the pivot position, places numbers smaller than pivot to the left, and greater to the right.The "Worst case array" is an array in the complete opposite order, but it's technically also a GOOD case scenario. The array is already sorted… just backwards. Inside of the partition function, we get the value to the immediate right and left of the pivot position and search for items that pass the comparison tests. Very quickly all of the values should pass the comparison tests because everything is already sorted just backwards. So yes! The time complexity: O(*n*log(n)) makes total sense.

What did I learn from the sorting algorithms?

The biggest thing I learned from these 4 different sorting algorithms is, Time complexity. Prior to starting this assignment, I wasn't sure what the big O notation was nor what time complexity really meant.

At first, I thought time complexity meant the amount of real world time it takes to do something on average. I understood that even if I have an incredibly fast CPU, the biggest thing that matters is the efficient code. If my sorting algorithms required more nested loops than necessary, it would drastically increase my sorting time.

Another big thing I learned was memory management and ADT's. I was only introduced to memory management a few months ago, and had no idea just how much work was done behind the scenes while coding in python. I learned that there are functions like malloc(), realloc(), and free() to grant me the ability to dynamically allocate memory whenever I need it.

How did I experiment with the sorts?

Before even experimenting with the sorts, I drafted a design of my code and ran through it in my head before coding. I wrote down small lists and mentally went through each sorting algorithm to understand how things were being sorted. Once I did this, I had an idea about how these algorithms work. With this knowledge, I began to code and ran into many unexpected problems like declaring how much memory to set for an array and how to add more memory into an array when necessary. From there, I just did a lot of trial and error: I typed out my code, drafted designs onto paper, mentally went through the algorithm like I was the compiler, and read the textbook & Piazza forum for knowledge.