Simon Lee
CSE 13

Write Up - Assignment 3

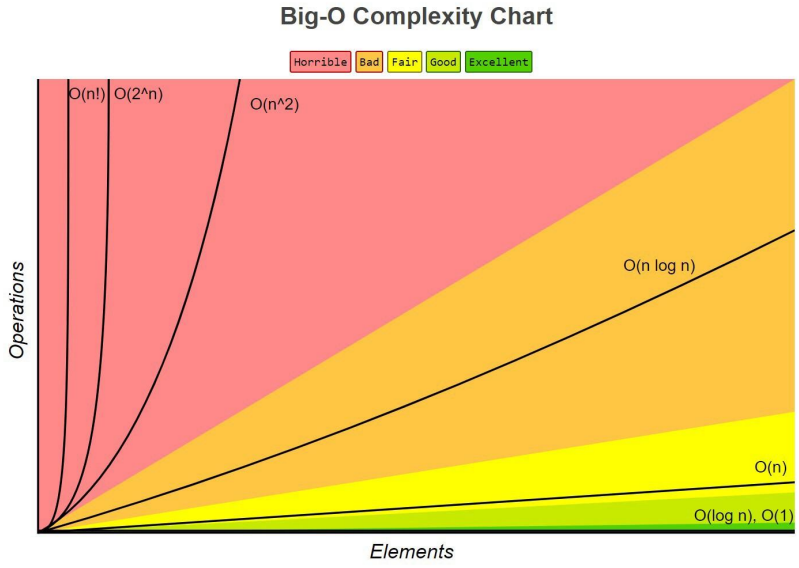
Introduction:

Sorting is one of the fundamental concepts in computer science. Ordering a sequence of anything can be done for you nowadays, but this assignment was designed for us to understand how these sequences get organized. We were asked to make a bubble sort, a shell sort, and two versions of quicksort (one utilizing the stack and one utilizing the queue). Doing so we are also asked to calculate the amount of moves and compares to show why some sorts are quicker than others. I will be discussing these sorts and why their time complexities are the way they are, what I learned from each individual sort, and a visualization of some sort to demonstrate my understanding of these concepts.

Bubble Sort:

Bubble sort is by far the slowest. In the worst case which is when the array is in descending order, it's time complexity is $O(n^2)$. The average case for bubble sort is the same as its worst case which is$O(n^2)$ The reason being is if there are more elements than the standard 100 given in random, it will likely be slow. Refer to the graph below by the following source. The reason I did not generate my own graphs is because I once again had trouble operating gnu plot.

source:

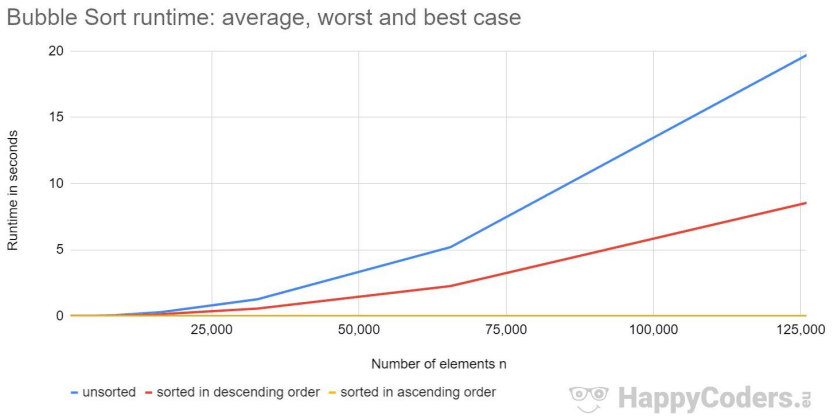


https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7

In this image there are a few time complexities that will be referenced. But in the case of bubble sort, in it's best case it is $O(n)$ which in the graphic has an average time complexity, meaning it is a somewhat okay algorithm if the array is already sorted. But in its worst and average case it has a time complexity of $O(n^2)$ which appears to be very bad. Obviously there is only one case that it will be in its best case and one case that it will be in its worst case but we can just assume based on the graphic that bubble sort overall is a very inefficient sorting algorithm. If time complexity doesn't make much sense, allow me to explain to you what big O notation means. Big O notation is describing the execution time of a sorting algorithm and shows how many steps it requires to complete it.

Ex:

So let's say we have an array with 100 elements. In its best time complexity which means it is already ordered, it is $O(n)$. This means that it will only require 100 comparisons of elements[i] and elements[i+1] and since they are already ordered, it will require no swaps. However let's say the arbitrary array still had 100 elements but they were in descending order. This means its time complexity is the worst case $O(n^2)$ and will require 10,000 moves to be organized. Obviously this means that it is slow but what if we began having more elements inside the array. So the relationship between the bubble sort is that if there are more elements, the less efficient the bubble sort will become because in its worst case the time complexity will grow with whatever the amount of elements are squared. So



Sources: https://www.happycoders.eu/algorithms/bubble-sort/

Moves Chart - average case

1,000 elements means 1,000,000 moves.

10,000 elements means 100,000,000 moves.

100,000 elements means 10,000,000,000 moves.

1,000,000 elements means 1,000,000,000,000 moves.

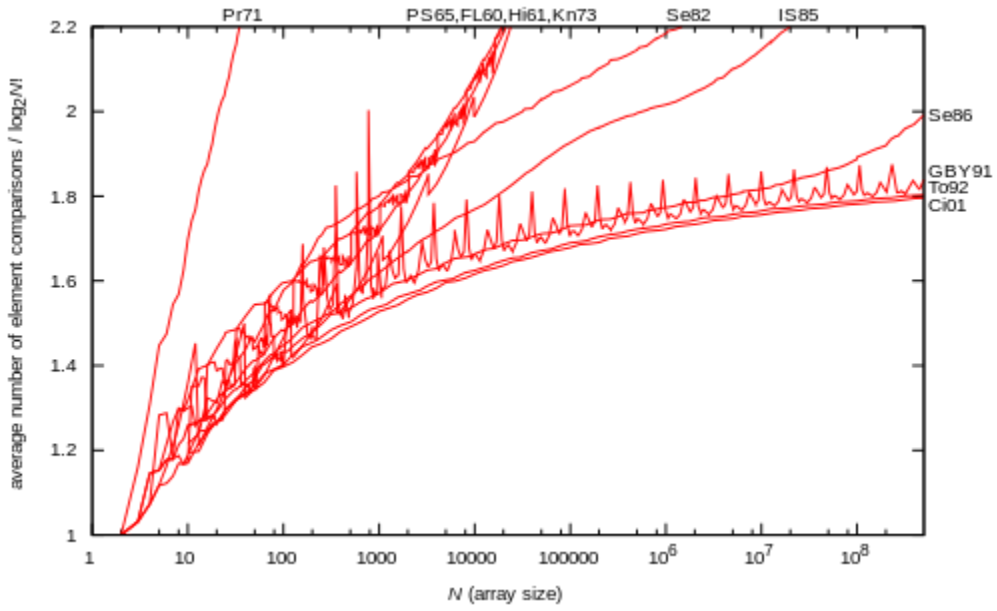
…

Shell sort

Shell sort in its best case is $O(n * log^2(n))$ which if we reference the graph is also pretty average. The reason being is because it utilizes a gap, and if it is already ordered the gap is essentially a waste of time because it has to check some arbitrary gap n, then divide n/2 in the next iteration, then another n/2 until we eventually get to a gap of 1 in which it executes essentially a bubble sort. In the worst case it is once again $O(n^2)$. However in its average case it is like it's best case $O(n * log^2(n))$. And this makes sense because it is not as inefficient as a bubble, but it still requires a great degree of maintenance. I used to think that shell sort was worse than bubble because it had a slower best case, and the same worst case but after running this program, it turns out that shell sort is definitely far more efficient than bubble sort. Also it is important to mention that shell sort has many different versions based on the gap sequence and the version we were told to implement was the Pratt sequence which can be represented as 2^p3^q. Like bubble sort I will provide an example to further understand shell sort.

Ex

If we have an arbitrary array which contains 100 elements, in its worst time complexity it will require 10,000 moves. If we have it in its best case we have to plug in $O(100 * log^2(100) = 664.386$. Obviously 664 .386 required moves is a little slower than the bubble sort but as previously mentioned, the average case which is the most likely to simulate the programming assignment is also $O(n * log^2(n))$.



Sources: https://en.wikipedia.org/wiki/Shellsort

Moves Chart - average case

100 elements means 664.386 moves.
1,000 elements means 9967.784 moves.
10,000 elements means 132877.123 moves.
100,000 elements means 1660964.04 moves.
1,000,000 elements means 19931568.569 moves.

Quicksort

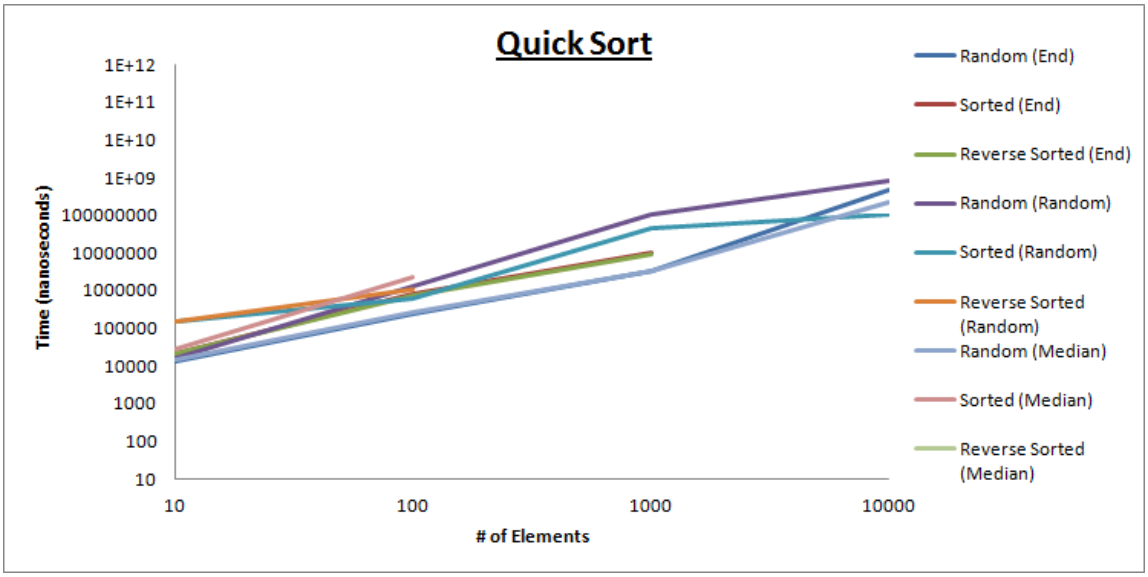
Quicksort was the last variation of the sort we had to implement. Quicksort is expected to be the quickest compared to its counterpart bubble sort, and shell sort. The best case scenario for quicksort is $O(n * log(n))$ which is not the same as shell sort since its base 10 instead of base 2. So it should be far quicker in all cases versus the two other sorts. Though they make the same amount of comparisons and moves, the stack and queue should have different max_capacities. I will talk more about that in a later section. The worst case though which can happen if the pivot is the first or last element is $O(n^2)$which is very contradictory to its name, but it is very unlikely that the pivot is the first or last element. And in our program it is impossible for it to be the first or last element because we put it through the equation:

```
pivot=arr[lo+ ((hi-lo) // 2)];
```

The average case for quicksort is the same as its best case which is $O(n * log(n))$. So in the example we will assume that there is some arbitrary randomly dynamically allocated array:

Ex:

In



Suppose there is ar
its best and averag
how big the differe

Source: https://my.eng.utah.edu/~bryans/CS%202420/hw5.html

Though this graph displays time versus amount of elements, it is a good visualization onto how quickly the sort is in a variety of cases. Though it does not show the worst case because its trajectory would be way out of the graph, you can see that for the most part regardless of how the elements are mixed up that $O(n * log(n))$ remains pretty consistent.

Moves Chart - average case

100 elements means 200 moves.
1,000 elements means 300 moves.
10,000 elements means 400 moves.
100,000 elements means 500 moves.
1,000,000 elements means 600 moves.

Why stack and queue max sizes are different for the same sort

Another interesting part about this assignment takes place in the quicksort algorithm. We implement the exact same program but utilize a stack and a queue abstract data structure. And you would expect the max_size of these values to be the same but they are not. However it does make sense considering that They are essentially opposites in which the stack is a “Last in First out”, and the queue is a “First in First out”. Because they both work a little differently, the sizes vary and in fact the queue size is just bigger because if you want to access a specific element you have to wait until it's gone through every single element before it. More will be discussed about why in the next section.

Does size of array affect max size of queue and stack

Yes and No. So in the default parameters we are required to sort 100 elements. When sorting 100 elements the max_stack_size is at 14. Therefore I experimented by sorting 1,000,000 elements and was astonished to see that the max_stack_size was only 64. So to answer whether the size of the array affects the stack, the answer is yes but at a really slow rate, and it is very unlikely that there will be a memory leak because we go over the capacity of the stack being full. I tried running 1,000,000,000 sorts but the program was killed rather there being a data leak. However the queue size is a little different. The queue size with 100 elements creates a max_queue_size is 36. But at 100,000 elements the max_queue_size is 17,492. With 100,000,000 elements the queue size is 12,167,018. So unlike the stack, the max_queue_size might run into a segmentation fault, but like the previous assessment, the program gets killed when trying to run 1,000,000,000. This makes sense logically to because if you want to access some element in the queue you have to wait till all previous elements have been dequeued. As for the stack you can access an element you need by taking it off the top of the stack. so if you needed a value from the stack it is much more efficient to pop off the element then have to wait through a queue.

Experimenting with the sorts

In the end I experimented with the sorts in a few ways. Obviously I wanted to test each sort to its limit, so I prompted in up to 100,000,000 elements in the command line argument. Doing so I was able to see the runtime for these algorithms first and foremost which has to do with the time complexity. Although bubble and shell sort take a few seconds since they are both very high maintenance sorts, meaning they require to check every n and n+1 element (bubble) or every n and n+gap element (shell). Quicksort still manages to take a few seconds also but in comparison to the other two sorts, it is much faster. I also experimented with the queue size to see if I could get a segmentation fault because we exceeded the amount of memory allocated to the queue, but it turns out that you actually run into an out of range error because the numbers are no longer within the uint32_t scale.

What I learned

Overall I learned alot from this lab and what is required of a sorting algorithm. At first I was questioning why we had to do four whole sorting algorithms but looking at it now, if we just made bubble sort, we would never actually know how slow it was compared to quicksort or even shell sort. Not only that but I also learned quite a bit about time complexity and how each sort requires a certain degree of maintenance. I also learned about the best, average and worst cases for each sort which helped me analyze how and why some sorts take longer than others.