Ruchit Patel

CSE13S, Spring 2021

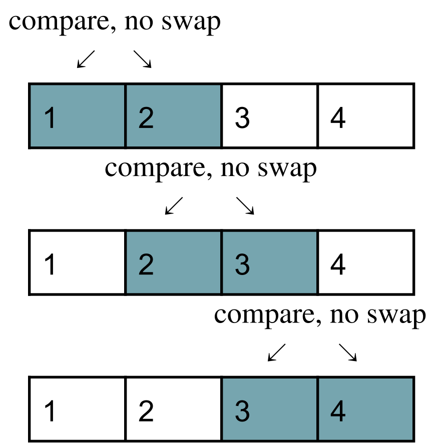
Prof. Darrell Long

21 April 2021

Assignment 3: Sorting: Putting your affairs in order

Analysis of Bubble Sort

1. Time Complexity
   a. Best Case
      i. Bubble sort would have to go through the whole array at least once and compare the elements. Thus, the best case time complexity would be order of n or O(n).
      ii. For example, let A be an array with 4 elements and sorted in ascending order. Now if we run bubble sort to sort in ascending order, we get:



      iii. Thus, if the array is already sorted, n-1 comparisons and 0 swaps are required . Thus it has O(n - 1) best time complexity which could be written as O(n) since the constant can be ignored as the n term dominates as n increases.
      iv. Graph of sorted arrays vs bubble sort comparisons and moves

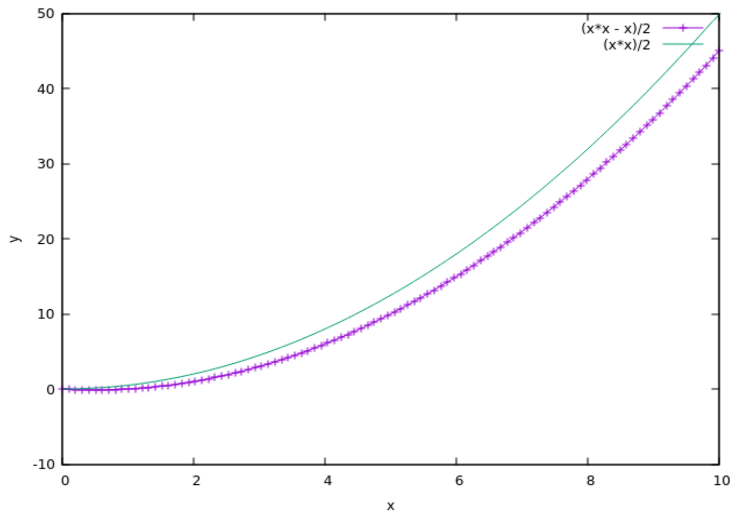
   b. Average/Worst Case (O(n^2))
      i. If the array is not already sorted, the each time bubble sort would have to compare one less element than the previous round.
      ii. For example, the first time bubble sort would have to compare n-1 elements, the second time bubble sort would have to compare n-2 elements and so on.
      iii. Writing down the comparisons for each round, we get total comparisons:

$$= (n - 1) + (n - 2) + (n - 3) + \dots + (n - (n + 1))$$
$$= n(n - 1) - (1 + 2 + 3 + (n - 1))$$

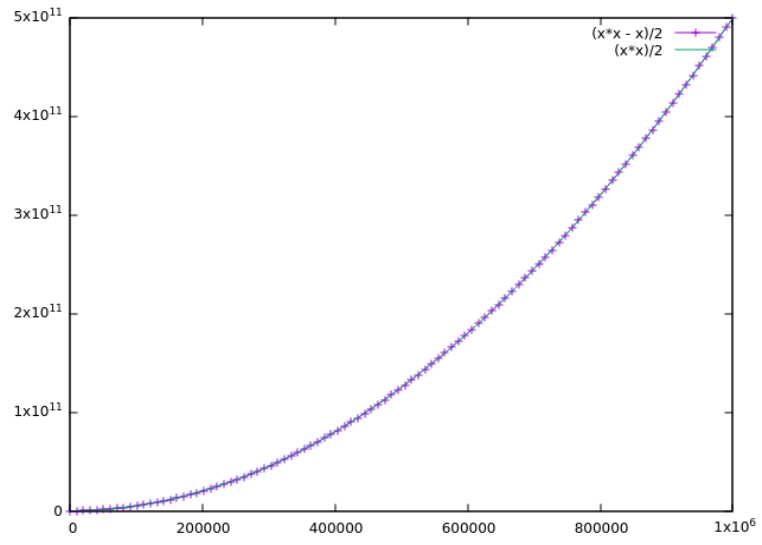
$$= n^2 - n - (\frac{(n-1)(n)}{2})$$
$$= \frac{(n^2-n)}{2}$$

iv. Thus, we could ignore the term $\frac{n}{2}$ since $\frac{n^2}{2}$ dominates as n grows larger. That could be clearly seen from the following graph:

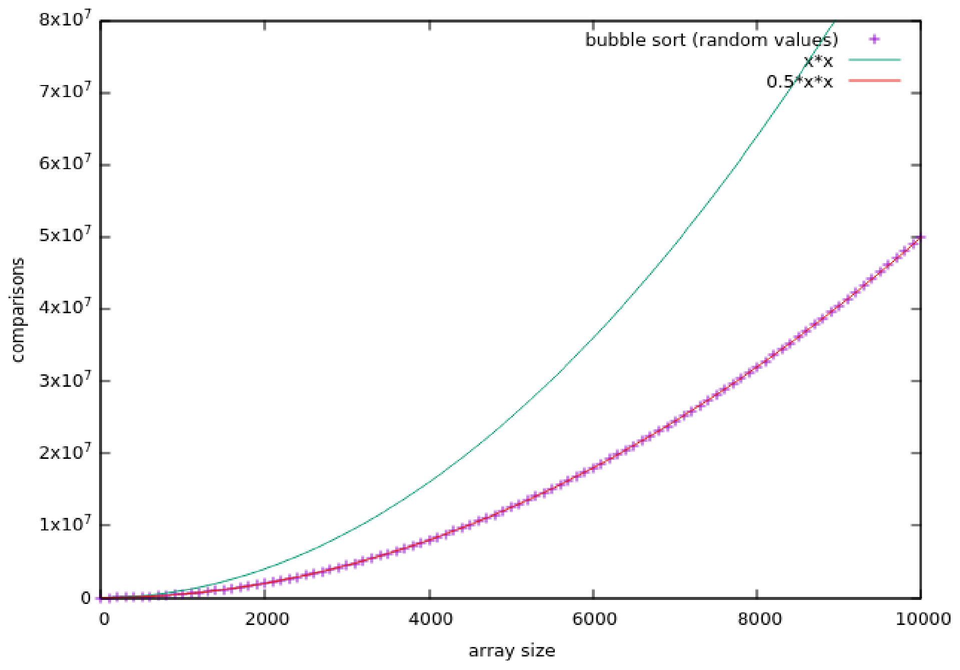
Small x values:



Larger x values:

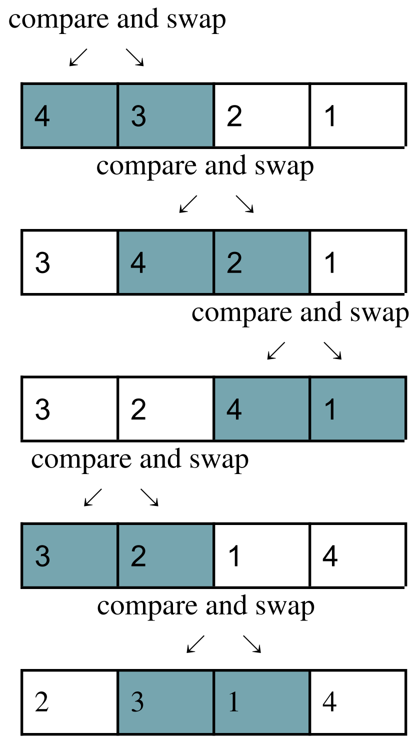


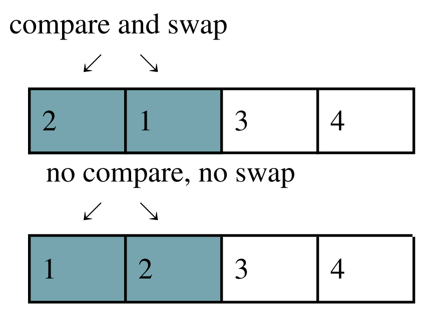
v. However, I think the constant $\frac{1}{2}$ in $\frac{n^2}{2}$, on the other hand, plays a huge factor. Since n^2 grows twice as fast as $n^2/2$ the constant should not be ignored. The above claim can be noted in the graph below as well:



vi. Since it is $\frac{n^2}{2}$ it could be written as order of n^2 or $O(n^2)$ where constant is ½.

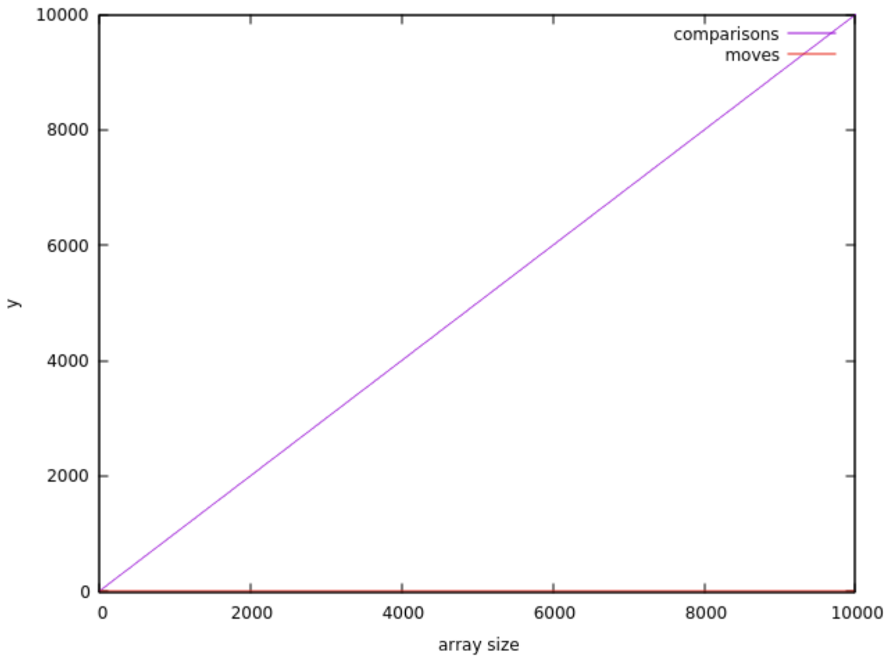
vii. Example: Array with 4 elements in reverse order (worst-case).





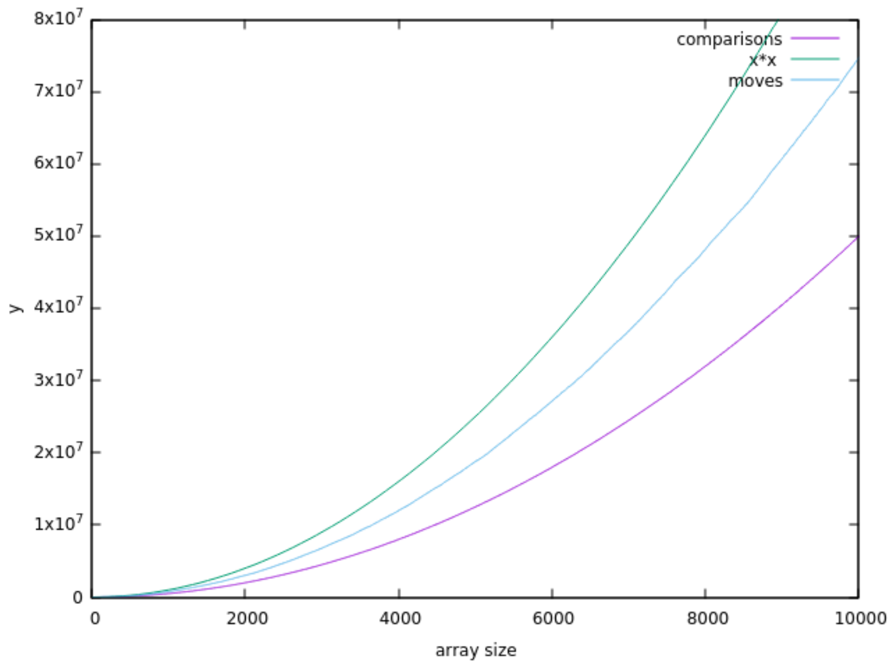
- Thus, the number of moves are 6 which is approximately $\frac{n^2}{2}$ or 8.

2. Graphs
   a. Graph of sorted arrays vs bubble sort comparisons and moves



i. As previously noted, the best case for bubble sort is when the array is already sorted.
ii. Another previous claim that best case complexity of bubble sort is O(n) can also be seen in the above graph where there are no moves and the number of comparisons are exactly n-1.

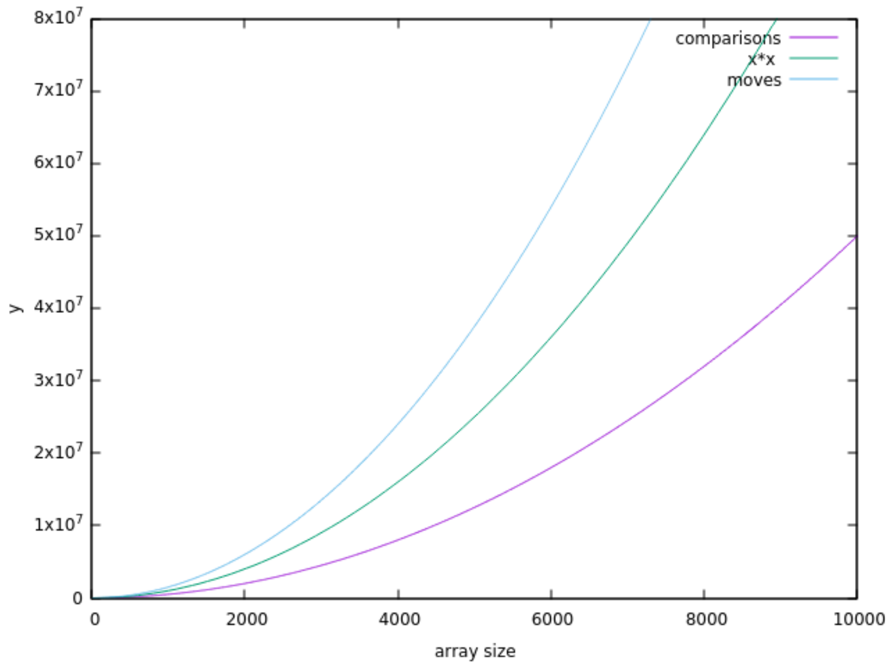
b. Graph of random arrays vs bubble sort comparisons and moves



i. The above graph shows that the average case for bubble sort is in the order n^2. The constant for the number of comparisons in this case is exactly ½ as derived previously.

ii. Similarly, the number of moves are in the order of n^2 as well; however, the constant is a bit higher since the largest element has to be moved to the end which on average happens frequently.

c. Graph of reversed arrays vs bubble sort comparisons and moves

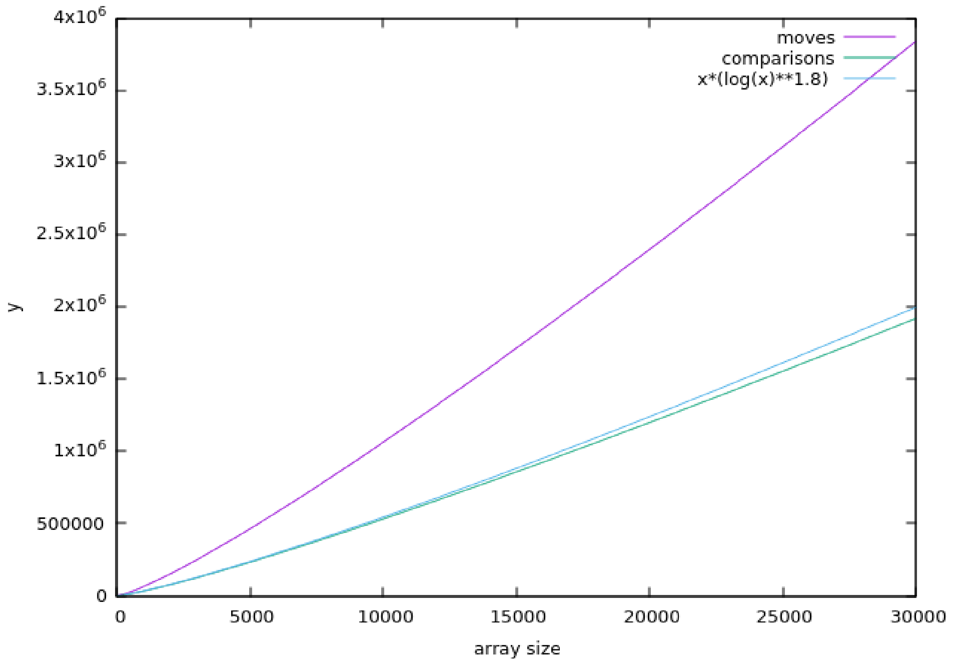


i. Unsurprisingly, the number of moves for a reversed have a larger constant since the largest element has to be moved from its position at the beginning to the position before the maximum element at the end.

Analysis of Shell Sort (w/ 2^p3^q sequence)

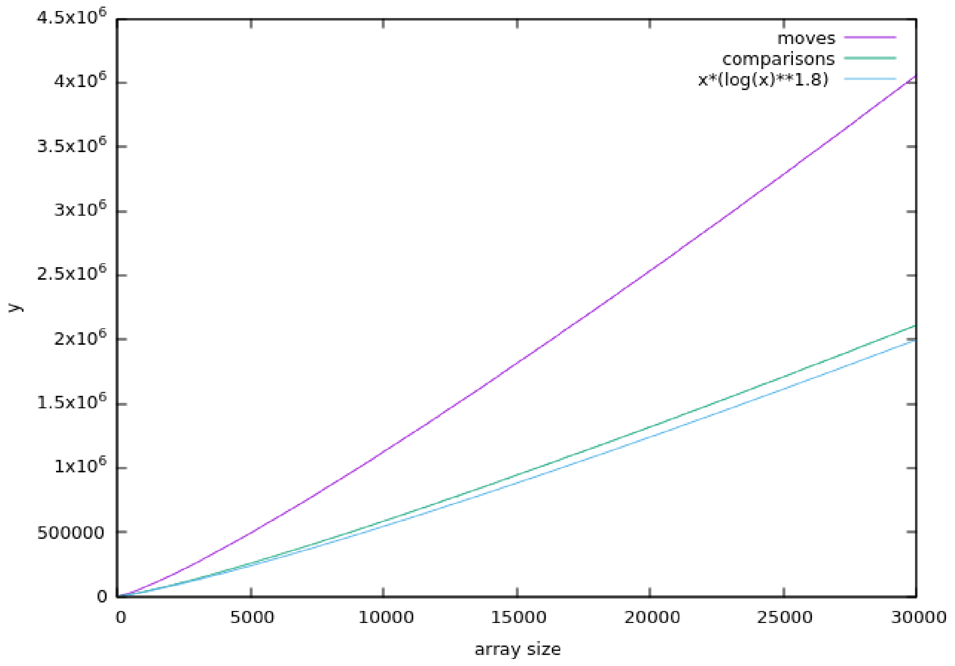
1. Time Complexity
   - If I understood it correctly, the time complexity would not be logarithmic (since the 3-smooth gap sequence does not half the next value each time) but rather it would be polylogarithmic as it reduces the gap and the list every time by some factor k.
   - My initial guess was $x*\log^k(x)$ where k is a constant. I was not too sure about the time complexity and thus I tried to understand it better with graphs.
   - For the graphs below, I found k near 1.8 to be the best representation of the time complexity for shell sort using 2^p3^q sequence.

   a. Best Case
      i. The best case for shell sort is also going to be the case when the array is already sorted.
      ii. The reasoning behind it is that shell sort would never have to backtrack any of the elements since all of them are sorted already.
      iii. The number of moves, however, would vary. Why? In shell sort, we still have to assign a temp variable to the value of gaps and we also have to assign it back to the position where we end up backtracking. Thus, they count as two moves.



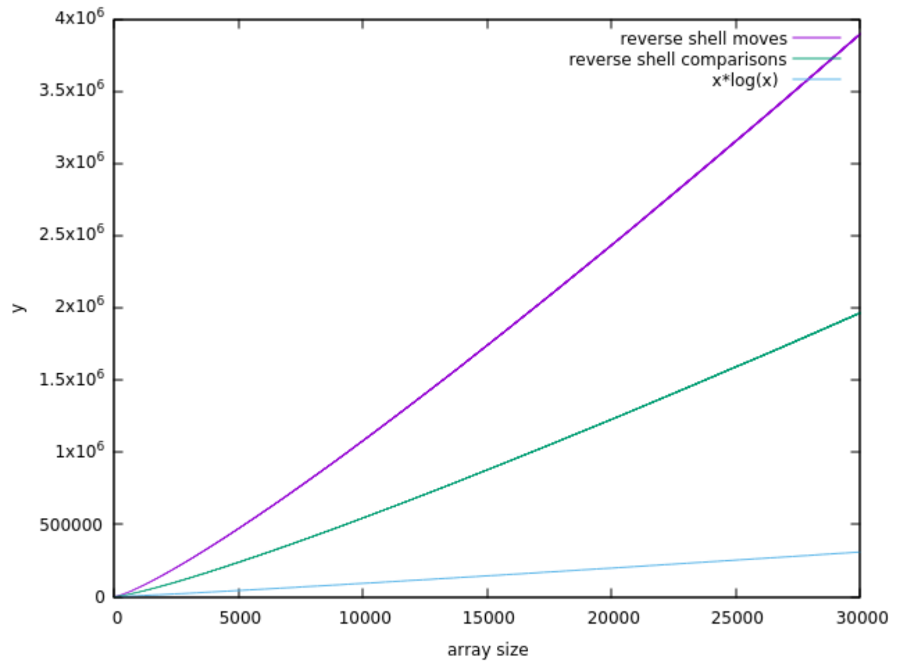
b. Average/Worst Case

   i. The average case for shell sort, like other sorting algorithms, is when the array is random.



   ii. Naturally, the number of moves would be higher since the probability of elements being out of order is higher and thus we would have to backtrack a little more.

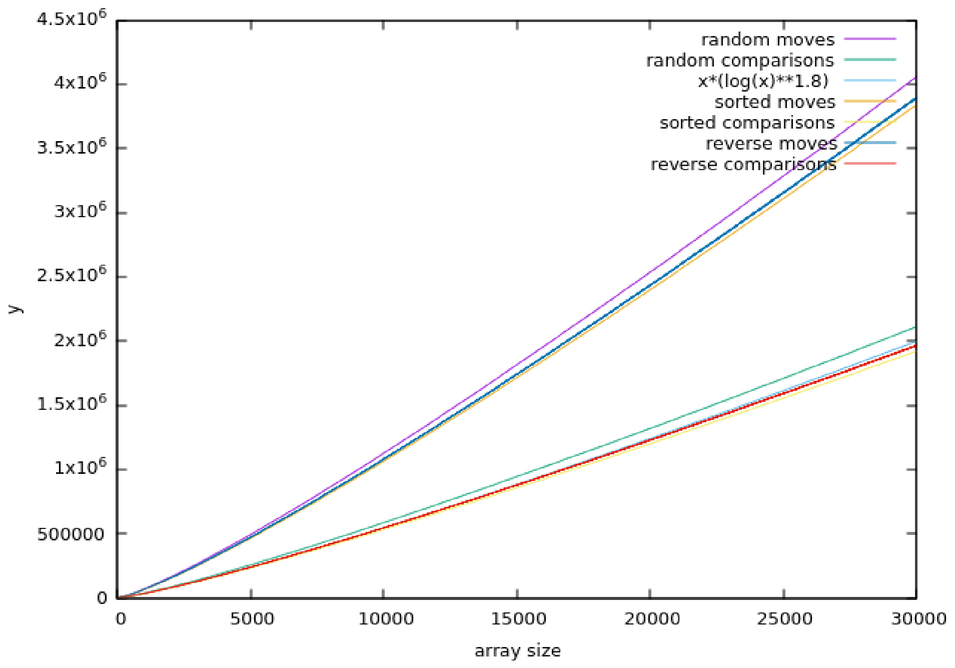
c. Reverse Array



i. Since shell sort (with the 2^p3^q sequence) compares far elements first, it would not have to waste a lot of moves in backtracking because the array is reversed and thus the elements would be put in order relatively quickly.

- Putting each of the arrays (reversed, sorted, and random) helps understand the working of the shell sort better.

d. Graph of all types of array with shell sort



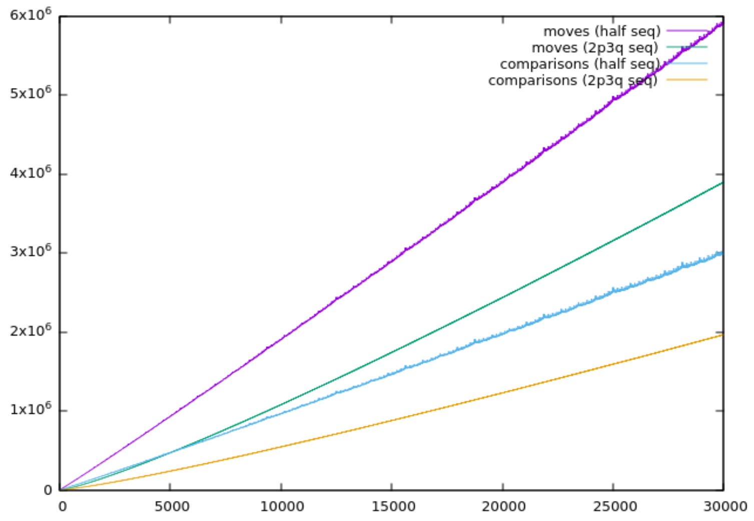
i. From the above it can be seen that the average/worst case, as the array size increases, is when the array is randomly sorted.

ii. Unlike bubble sort, however, the difference between the best and worst case is for shell sort is lower.

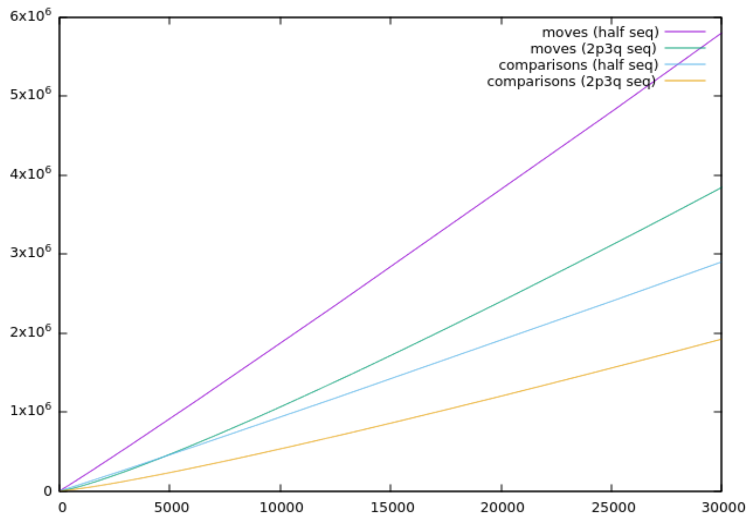
iii. It can also be seen that the reverse array performs relatively similarly to sorted arrays since the gap sequence puts the lower elements from the end at the front faster.

2. Gap Sequences
   -> Like quicksort's complexity depends on the pivot point, shell sort's complexity depends on the gap sequence.
   -> I experimented with a gap sequence that halves each iteration. An example gap array would be {50000, 25000, 12500, ….}.
   -> This gap sequence would not perform that well since it leaves out a great number of elements that could be compared and moved at the beginning.

   a. Halfing each time
      i. Graphs for support
         1. Random



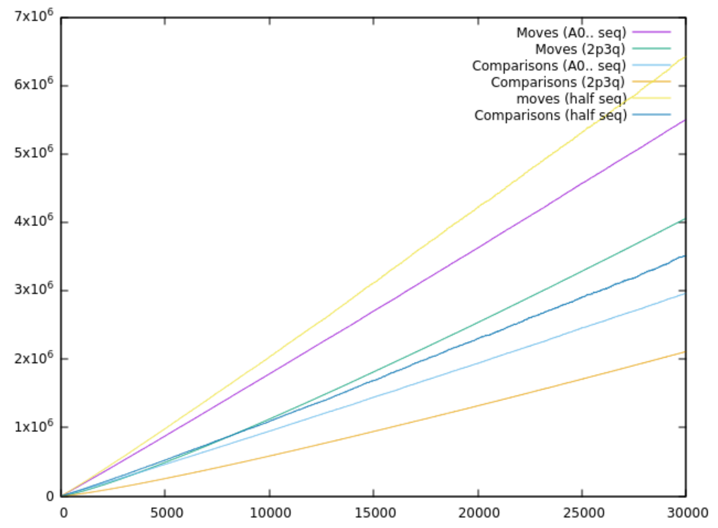
         2. Sorted (Reversed had similar results)



- Clearly, it can be seen that a 2p3q sequence performs extremely better than just halfing each time as stated before.

b. A036569
-> I was looking around on the wikipedia page and found this series interesting. I wanted to try and compare its results with the 2p3q sequence.
-> NOTE: I do not completely understand this but I was just curious so I implemented it.
-> CREDITS: Based on the sequence description mentioned here: oeis.

1. Graphs
    a. Different array types
        i. Random (Sorted and Reverse arrays had similar results)

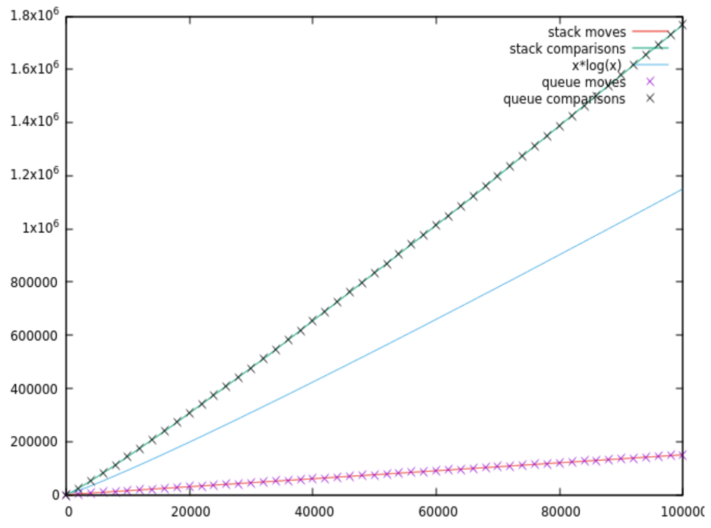


- A036569 sequence is a lot better than halfing each time but, as it could be noted from above, 2^p3^q still outperforms both sequences.

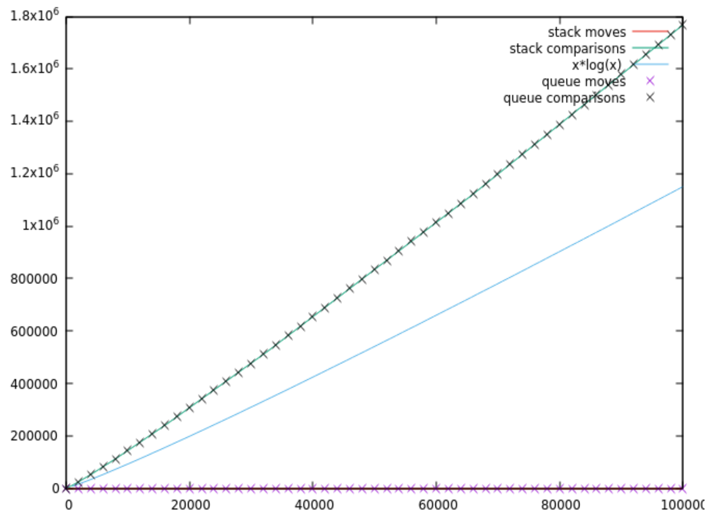
Analysis of Quick Sort

1. Time complexity
    a. Quick sort constantly divides the array into subarrays of half the size and it does that at least array size times. Thus, the time complexity for quicksort is in the order of nlogn where the constant is 1/log2 since the array is divided into half sizes each time.

2. Graphs
    a. Different array types
        i. Reverse

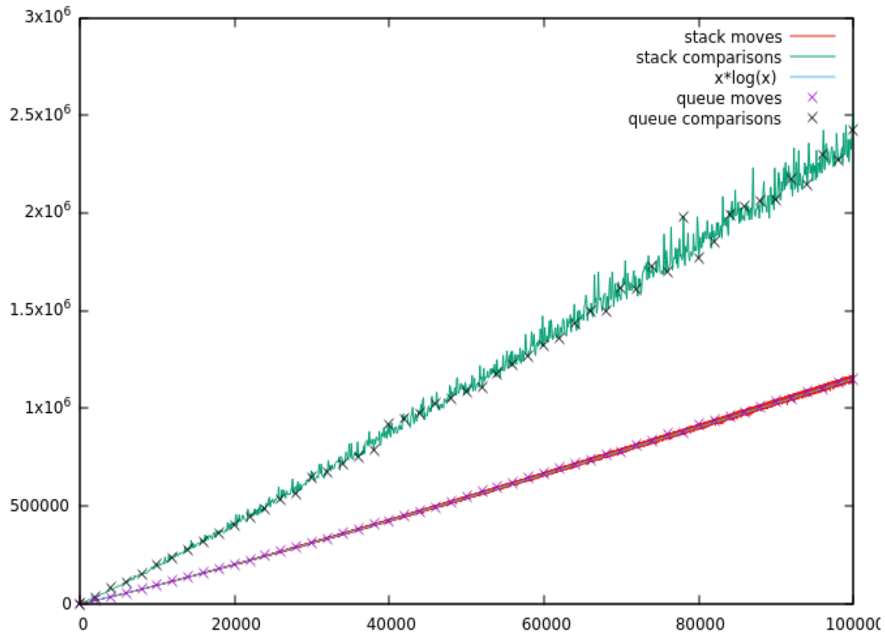


ii. Sorted



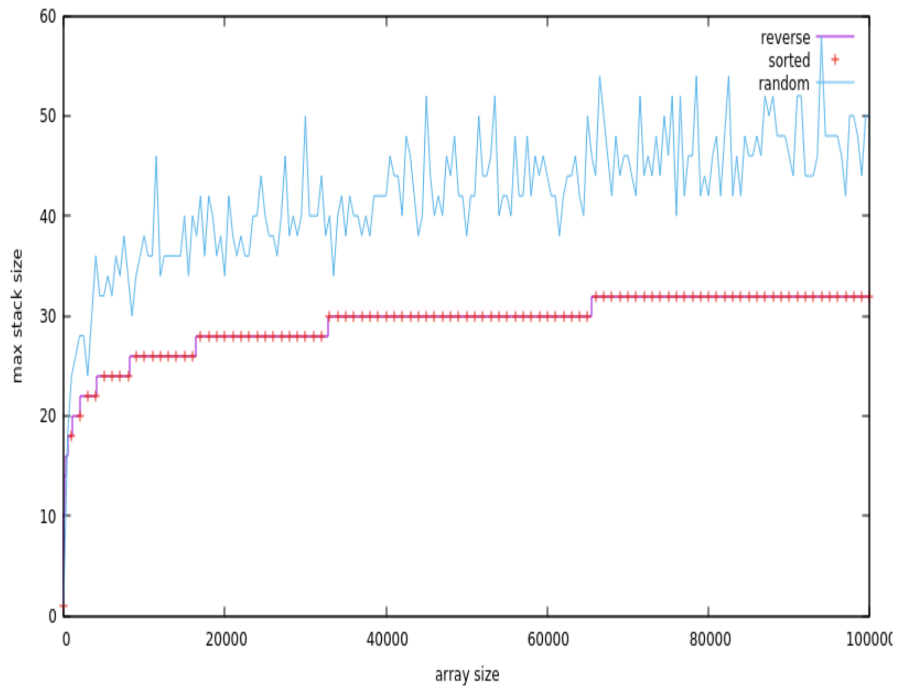
- The number of comparisons in both of the above cases stays the same since both of them are in the order of n logn.
- The number of moves in the sorted arrays are obviously zero but in the case of reversed arrays are in the order of n logn. However, the constant is a little higher than $1/\log_2 n$ since each time only (n-1)/2 elements are moved.

iii. Random



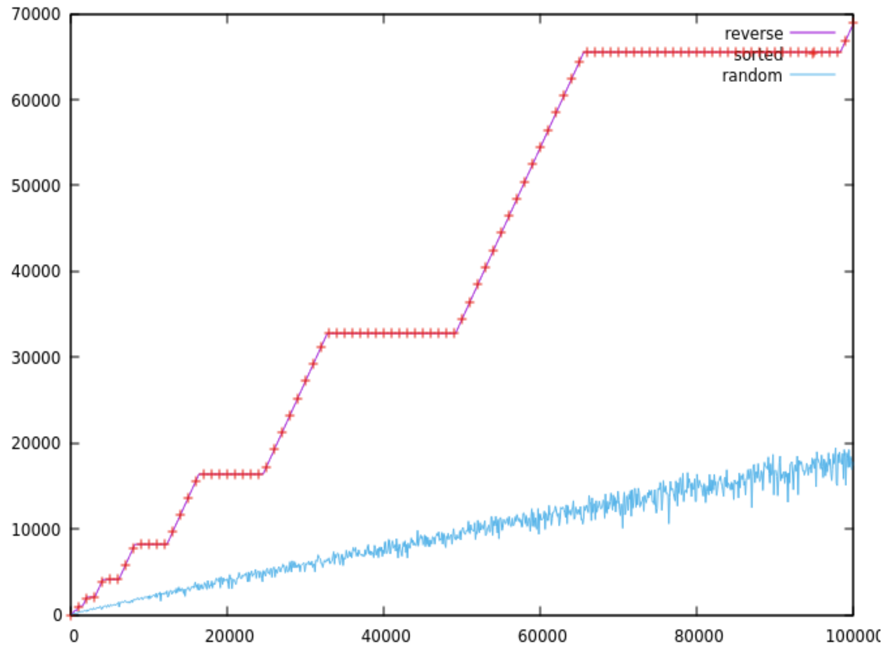
- Despite showing an increasing linear trend, the number of comparisons and moves for randomly sorted arrays are a bit uneven compared to its neighbors.
- One possible reason might be since the elements are in a random array, they would not be greater than the pivot element every time. Thus, they would not be moved or compared to the elements on the right side of the pivot (to find a lower element) and hence the moves and comparisons being uneven.
- NOTE: The queue comparisons and moves seem even because only every 1000th element was graphed to keep it clear. In reality, they are as uneven as the stack counterpart.

b. Stack sizes



i. From the above graph it can be noted that the stack size for randomly and sorted arrays behave like an increasing step function.
ii. The gap where it stays constant approximately doubles each time.
iii. A possible explanation for such behavior can be the fact that low and high are pushed on the stack for the partition to sort the subarrays and return a pivot position. Thus, the number of subarrays (by halving the original array) created stay about the same unless the input size gets doubled. In that case, two more subarrays would have to be sorted which explains the step of 2 in stack size as the input doubles.
iv. Random arrays, on the other hand, have random stack sizes. As explained before, this is because the number of subarrays that would have to be created and thus unknown number of pushes for pivot point remain uneven.

c. Queue sizes
   i. The explanation for the change in step-like increase in queue size is similar to that of the stack.
   ii. The only difference is that queue sizes linearly increase until they move onto the next step which, unlike stack, doubles each time.
   iii. I am not totally confident but one reason for it might be the fact that the queue, unlike stack, might enqueue all subarray pivot points before completely dequeuing them at the end of recursion.
   iv. The explanation for linear but uneven increase of queue size with random arrays stays the same as the stack's explanation.

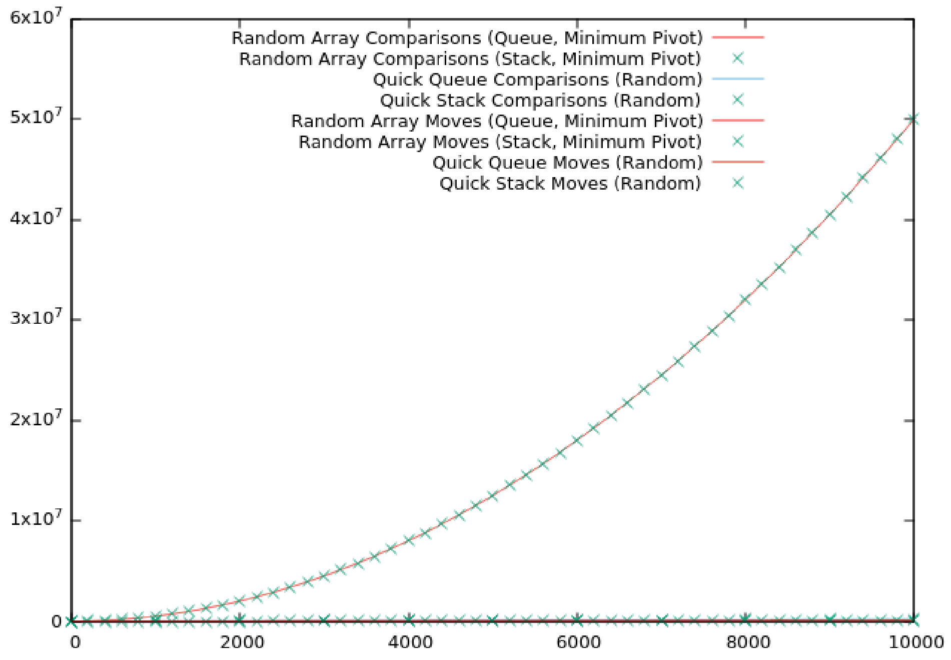


3. Pivot Points

   -> The number of moves depends directly on the location of the pivot point. In the above case, the pivot point was always at median. To experiment with it, I chose four pivot points: always the first element of a subarray, always the last element of a subarray, always the minimum element of a subarray, and always the maximum element of a subarray.

   -> Note that choosing maximum or minimum element would be the worst case as explained in the design and the number of moves would be in the order of O(n^2).

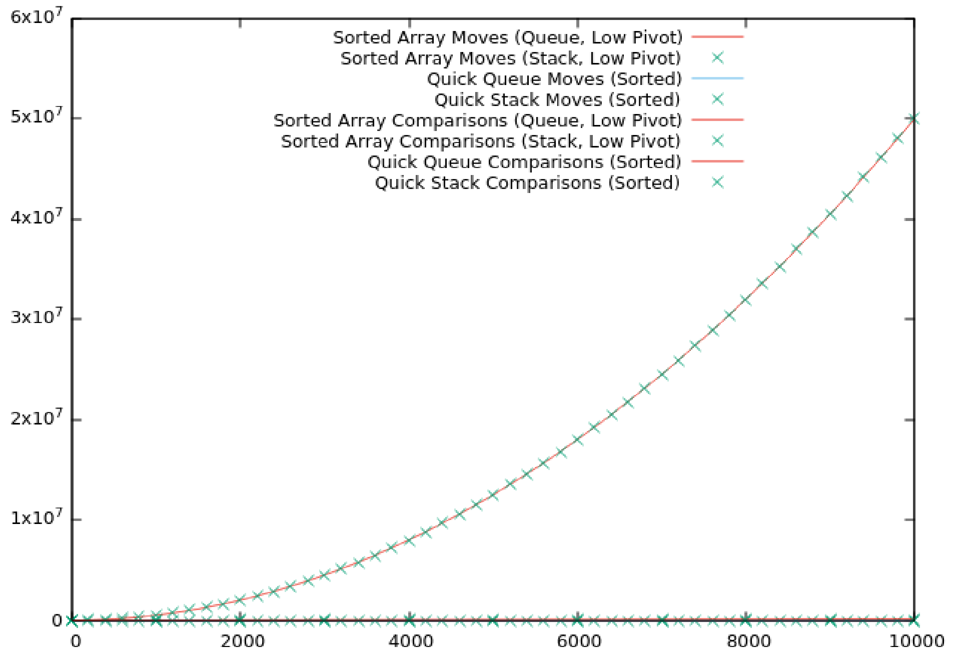
   a. Minimum element



i. The previous prediction that the order of comparisons would be a lot worse can be clearly seen in the above graph.

ii. NOTE: The results for reverse and sorted arrays follow the same general pattern. Graphs for arrays with maximum elements as their pivot also yield the same results.

b. Always first index



i. Weirdly enough, always choosing the lowest index results in a similar inefficient outcome which is highly similar to choosing the minimum element as a pivot.

ii. NOTE: The results for reverse and sorted arrays follow the same general pattern. Graphs for arrays with the last index as their pivot also yield the same results.

NOTE: I had an unusual pattern (might be an error) while plotting stack and queue sizes for arrays with different pivot points. For sorted and reversed arrays, both, maximum stack and queue size, remain constant at 2 or 4 respectively.