Analysis – Sorting: Putting your affairs in order

Daniel Chang

Fall 2021

1 Introduction

In this assignment we compare four sorting algorithms: Insertion Sort, Shell Sort, Heap Sort, and Quick Sort. The comparison metric we are using is the number of moves and comparisons that each sort uses. We first compare their time complexities given a random array of integers. Then, we look at 2 special cases: an array that is in sorted order, and an array in reverse sorted order. While comparing these algorithms, we take note of and try to understand anything peculiar in the data.

2 Code

Below is a simplified version of the code for Insertion Sort and Quick Sort.

```
void insertion_sort(int *A, int n){
    for(int i=1; i<n; i++){
        int j = i, temp = A[i];
        while(j > 0 && temp < A[j-1]){
            A[j] = A[j-1];
            j--;
        }
        A[j] = temp;
    }
}

int partition(int *A, int lo, int hi){
    int i = lo - 1;
    for(int j=lo; j<hi; j++){
        if(A[j-1] < A[hi-1]){
            i++;
            swap(A[i-1], A[j-1]);
        }
    }
    swap(A[i], A[hi-1]);
    return i+1;
}
void quick_sort(int *A, int lo = 1, int hi = len(A)){
    if(lo < hi){
        int p = partition(A, lo, hi);
        quick_sort(A, lo, p-1);
        quick_sort(A, p+1, hi);
    }
}
```

3 Analysis

Let’s look at how quickly each series converges based on the number of terms:

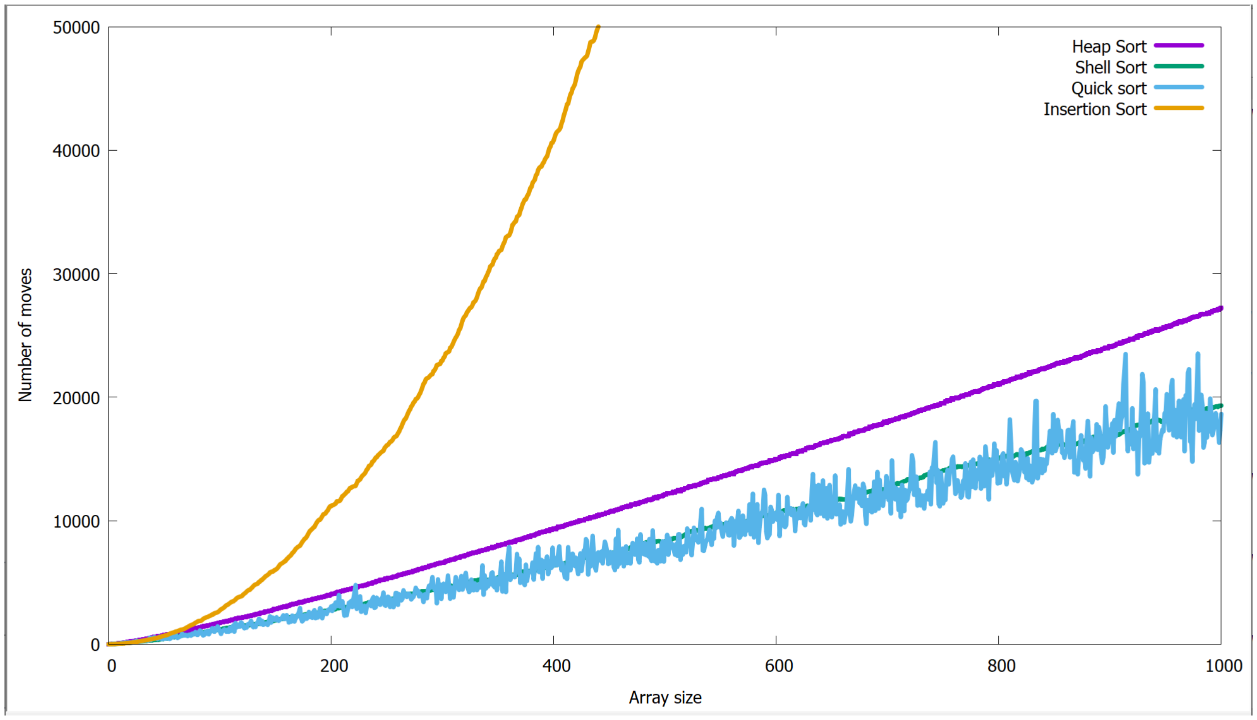
Table 1: Number of moves to sort array

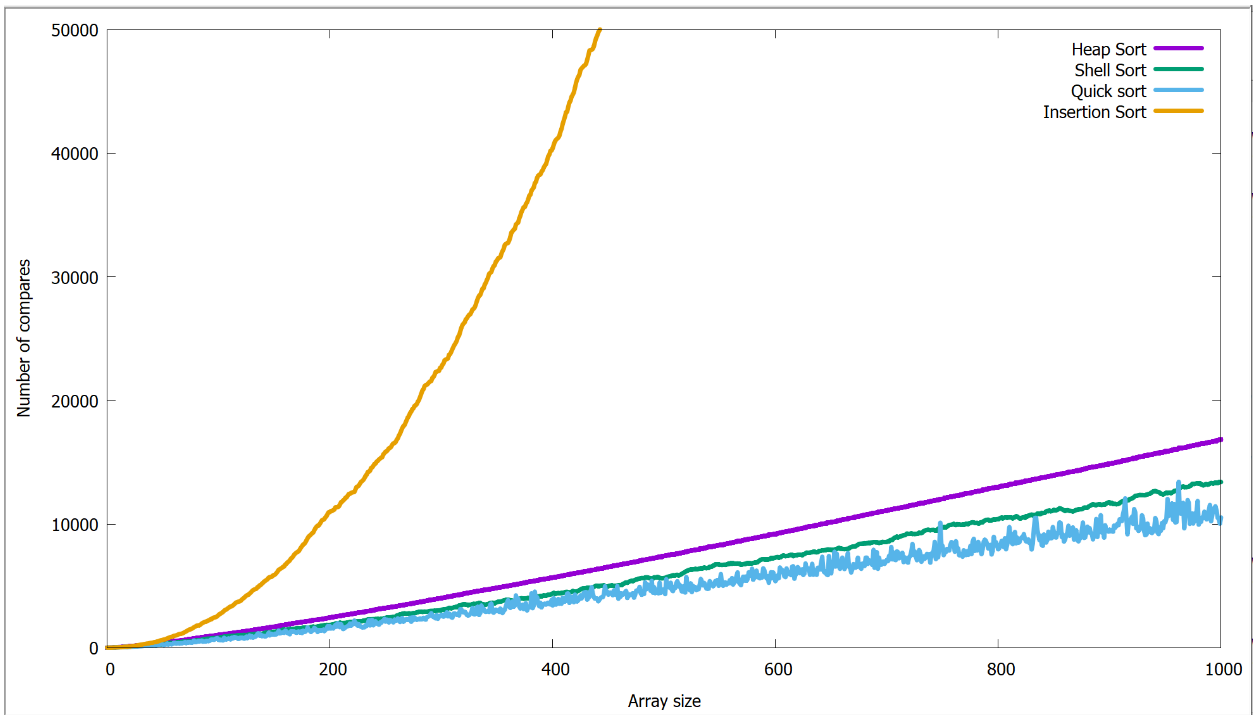
Array Size	Quick Sort	Heap Sort	Shell Sort	Insertion Sort
100	1053	1755	1183	2741
1,000	18642	27225	19323	$2.5*10^5$
10,000	$2.6*10^5$	$3.7*10^5$	$3.1*10^5$	$2.5*10^7$
100,000	$3.4*10^6$	$4.7*10^6$	$5.0*10^6$	$2.5*10^9$
1,000,000	$3.7*10^7$	$5.7*10^7$	$8.0*10^7$	N/A
100 (sorted)	15147	1920	684	198
100 (reversed)	7647	1548	914	5148

Table 2: Number of compares to sort array

Array Size	Quick Sort	Heap Sort	Shell Sort	Insertion Sort
100	640	1029	801	2638
1,000	10531	16818	13394	$2.5*10^5$
10,000	$1.5*10^5$	$3.7*10^5$	$3.1*10^5$	$2.5*10^7$
100,000	$2.1*10^6$	$3.0*10^6$	$4.0*10^6$	$2.5*10^9$
1,000,000	$2.4*10^7$	$3.7*10^7$	$6.8*10^7$	N/A
100 (sorted)	4950	1081	342	99
100 (reversed)	4950	944	500	4950

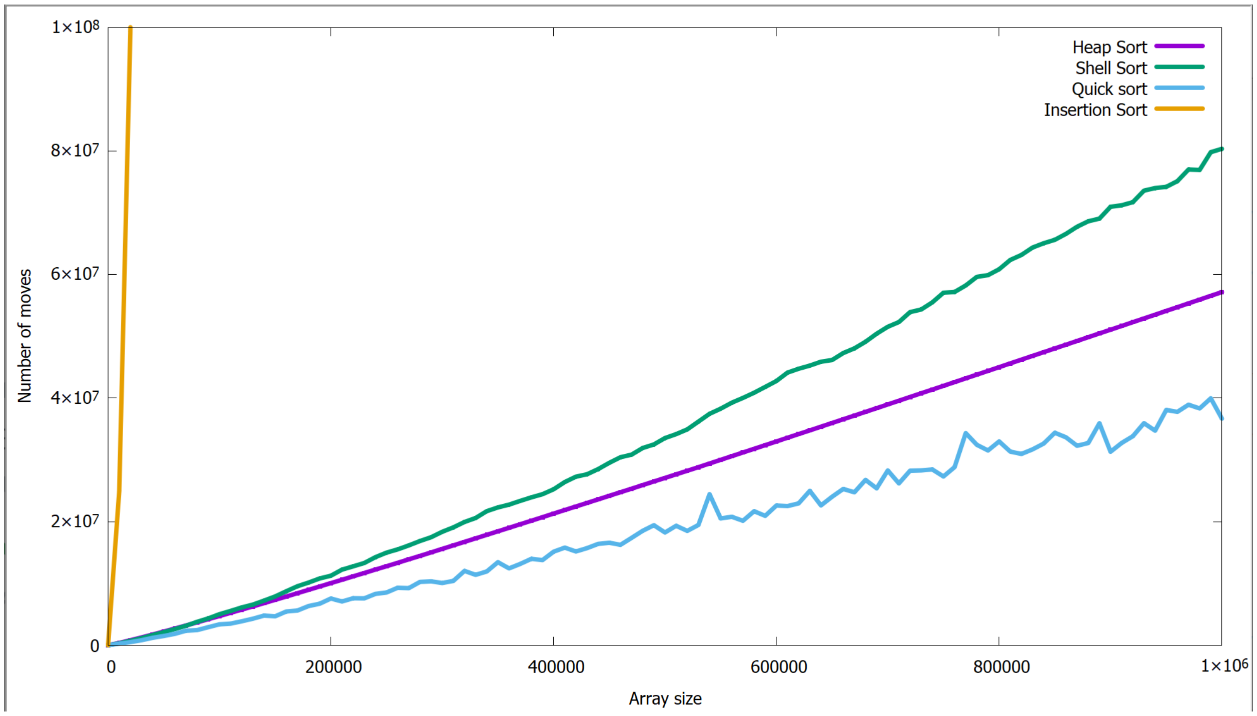
To begin comparing the sorting algorithms, let's first look at their moves and comparisons for an array size up to 1,000:



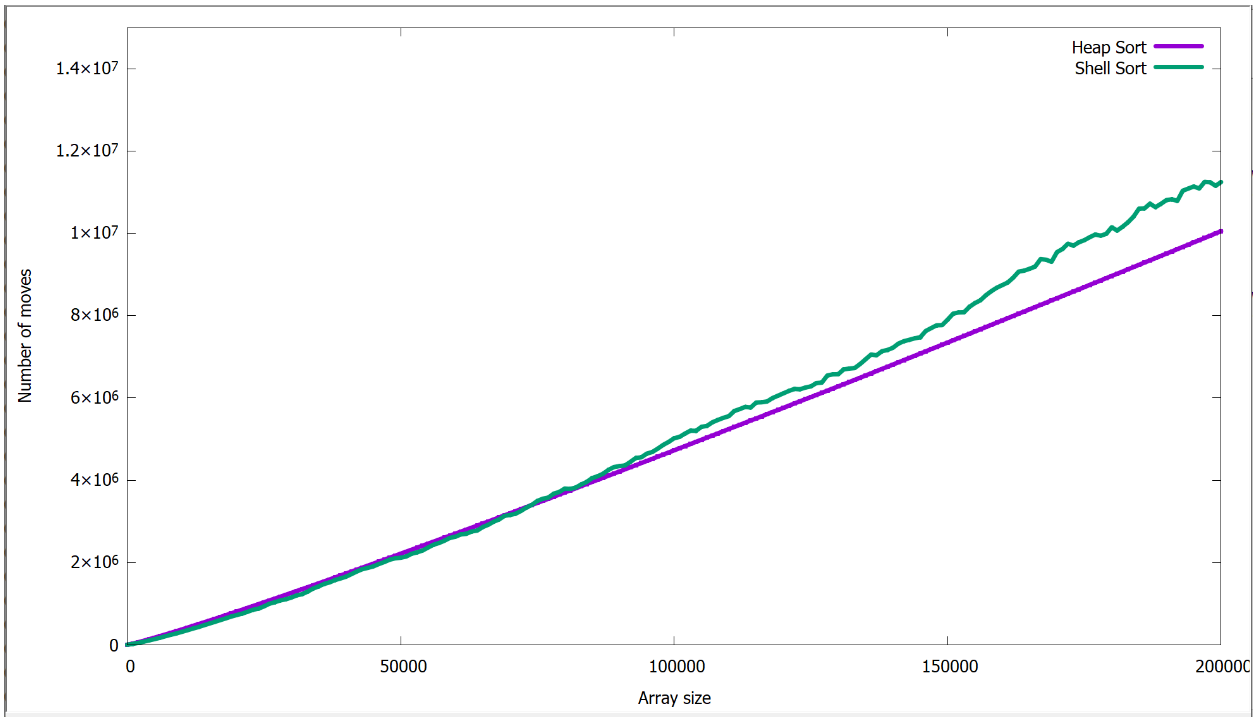


We know the time complexities of each sorting algorithm- Quick Sort is $O(n\log(n))$, Heap Sort is $O(n\log(n))$, Shell Sort is $O(n^{\frac{3}{2}})$, and Insertion Sort is $O(n^2)$. The graph generally seems to follow this pattern. The only two things to note are that Quick Sort has a high variance, and that Heap Sort takes more moves and comparisons than Quick Sort. Well, the Quick Sort algorithm has a high variance because the choice of pivot heavily impacts the time complexity. For example, if the starting array chose the first pivot as its median, the Quick Sort algorithm would be about twice as fast than if it chose its largest element. For the Heap Sort, we can say Heap Sort has a much larger constant *c* in its time complexity.

Here, we can expand our view to an array size up to 1,000,000:



When looking at the above graphs, one thing you might note is that although the first graph shows Shell Sort is faster than Heap Sort, this broader graph shows that Heap Sort beats Shell Sort for larger arrays. We can observe this more closely by narrowing our search range:

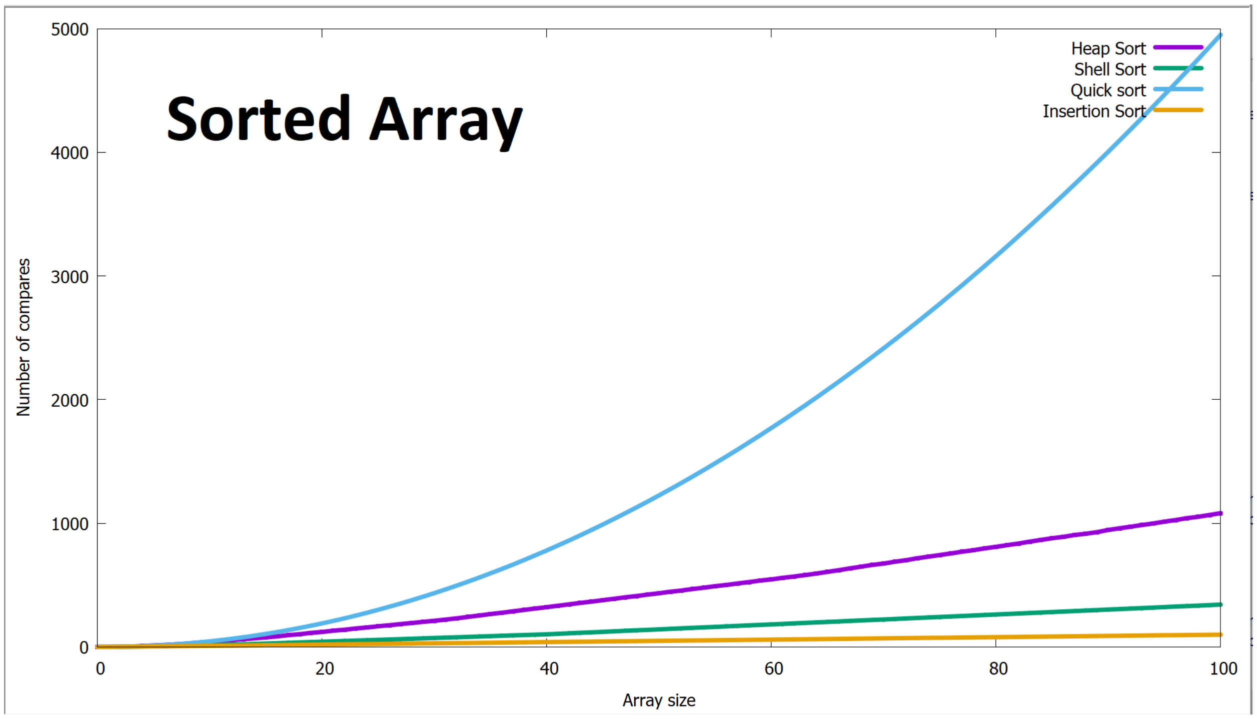


Around an array size of 70,000, Heap Sort begins to beat Shell Sort. Why is this? Well, Heap Sort has a time complexity of $O(n\log(n))$ while Shell Sort has a time complexity of $O(n^{\frac{3}{2}})$. Imagine Heap Sort has a constant c equal to 10. Then, by setting

$$10 * nlog_2(n) = n^{\frac{3}{2}}$$

we get $n = 20519$. Setting c to 17 and we get $n = 75967$. So, we might suppose that Heap Sort's constant divided by Shell Sort's constant is around 17.

Next, we look at how these sorting algorithms compare when dealing with an array that's already sorted:



Insertion Sort does the best, followed by Shell Sort and Heap Sort. Quick Sort is by far the worst. Looking at the code we can seek to explain this behavior.

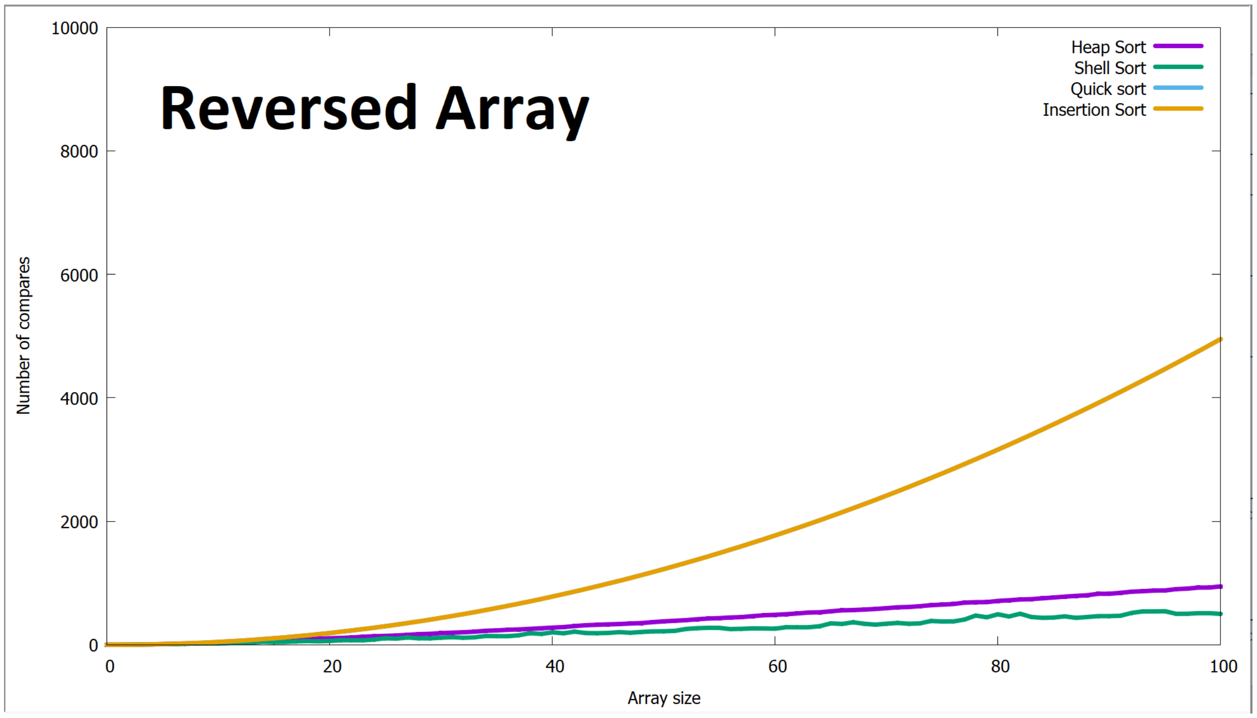
Insertion Sort only swaps pairs of elements such that the left element is greater than the right element. Since the array is already sorted, Insertion Sort does not swap any elements, but makes $n-1$ checks that shows for each pair of elements, the left element is less than the right element. Therefore, Insertion Sort verifies a sorted array in $O(n)$ time.

Heap Sort performs slightly worse on a sorted array of size 100 than it did on a random array of size 100. This makes sense because for any given array, Heap Sort builds a heap then fixes the heap $n-1$ times. For an array in reverse order, building the initial heap would take longer than a random array.

Shell Sort performs much better on a sorted array than a random array. Since Shell Sort is a variation of Insertion Sort, it makes sense that it would be faster, because Shell Sort would not have to swap any elements. However, since Shell Sort has to consider every gap, the time complexity to verify a sorted array is still $O(n\log(n))$.

Quick Sort performs the worst on the sorted array, and looking at the algorithm it's clear why this is the case. The partition algorithm will always be choosing the largest element as a pivot, so every recursive call to Quick Sort will only reduce the array size by 1 instead of the average of dividing it by 2.

Lastly, we look at the case where an array is sorted in reverse order, or from greatest to least:



One thing you might notice right away is that the Quick Sort line is missing. As it turns out, Quick Sort and Insertion Sort have the exact same number of compares for a reversed array. If you consider the code, this makes sense. Insertion Sort would shift the largest element to the left 99 times, the 2nd largest element to the left 98 times, and so on. Quick Sort would choose the first pivot as the largest element, 2nd pivot as the 2nd largest element, and so on. Thus, it would also make 99 + 98 + 97 + ... comparisons. In fact, if you refer back to the table, the number of comparisons for both Quick Sort and Insertion Sort on a reversed array equals 4950, which is exactly 1 + 2 + ... + 98 + 99.

Shell Sort performs better on a reversed array than a random array, but worse than a sorted array. Looking at the code, it's not clear why shell sort would perform well on a reversed array while insertion sort would perform poorly. To understand why would require a more thorough analysis of the gap generation function.

Finally, we see that Heap Sort performs slightly better on a reversed array than a random array. The reason for this is similar to why Heap Sort performs slightly worse on a sorted array. The reversed array is already a valid heap, so Heap Sort spends less time building the initial heap.

4 Conclusion

In this paper we analyzed different sorting methods based on two metrics: the number of moves and the number of comparisons it took to sort an array of random integers. From this, we were able to find a unique trait for each sorting algorithm. Insertion Sort performed very slowly, Quick Sort had a high variance, and Heap Sort only beats Shell Sort at array sizes of over 70,000. Next, we looked at two special cases, a sorted array and a reversed array. From this, we were able to gain a better understanding of how these algorithms worked. For example, if the performance was different for a sorted or reversed array, we looked through the code and identified why this was the case. The data suggests that Quick Sort is the best while the other three sorts have uses for edge cases. Insertion Sort is the fastest to verify if an array is sorted. Quick Sort may be too slow if an array is mostly in sorted or reverse order, so Shell Sort and Heap Sort are more reliable than Quick Sort. Finally, Shell Sort would be better for an array size of under 70,000, while Heap Sort would be better on an array size over 70,000.