Assignment 3 WRITEUP.pdf

Roman Luo

February 5, 2023

1 Different sorting algorithms.

1.1 What I have learned:

- **Shell Sort:** Shell sort is an algorithm that sorts elements in a partially sorted array by gradually reducing the gap between elements to be compared. I used the given python code in this assignment to produce a gap sequence for shell sort. The gap value is used to partition the array into smaller sub-arrays, which are then sorted using insertion sort. It's a variation of the insertion sort algorithm sinces insertion sort is essentially shell sort with 0 gaps and is considered one of the fastest algorithms for sorting small arrays. The time complexity of shell sort is between $O(n^2)$ and $O(n \log n)$ depending on the gap sequence used.

- **Batcher Sort:** Batcher's Odd-Even Merge Sort, also known as Bitonic sort, is a parallel sorting algorithm that sorts elements in a partially sorted array using the divide-and-conquer technique. This algorithm is based on the idea of merging adjacent elements in an array in a specific order as it sorts the array by first dividing it by two and then merging the quotient in a specific order. The merging process is performed in a parallel manner, which makes the algorithm highly efficient for sorting large data sets. The time complexity of Batcher's Odd-Even Merge Sort is $O(n \log^2 n)$. This is because the algorithm uses a divide-and-conquer technique with a merging process that takes $O(\log n)$ time, resulting in a total time complexity of $O(n \log^2 n)$.

- **Heap Sort:** Heap sort is a comparison-based sorting algorithm that sorts elements in an array by transforming it into a binary heap data structure. Repeatedly removing the maximum element from the heap(adding it to the sorted array) and replacing it with the minimum. The time complexity of Heap Sort is $O(n \log n)$. This is because each step of the algorithm takes $O(\log n)$ time, and the algorithm performs n steps, resulting in a total time complexity of $O(n \log n)$.

- **Quick Sort:** Quick sort is a comparison-based sorting algorithm that sorts elements in an array by selecting a pivot element and partitioning the other elements around it. The average time complexity of Quick Sort is $O(n \log n)$. For the average case, the pivot element is chosen such that the two sub-arrays are roughly equal in size, resulting in a balanced partition. In the worst case(which I will talk about later), however, the pivot element may be chosen poorly, resulting in one sub-array with all elements and the other sub-array with only one element. In this case, the time complexity is $O(n^2)$. Quick sort is considered one of the fastest algorithms for sorting large data sets due to its good average-case performance and efficient use of cache memory.

1.2 Under what conditions do sorts perform well or poorly?

- **Shell Sort:** The algorithm performs well in certain conditions where the array is partially sorted or contains some known structure that the algorithm can exploit, and vice versa. However, Shell Sort is considered less efficient than other sorting algorithms, and its exact time complexity is poorly understood due to the gap sequence. (If a gap is equal to 0, it performs the same as insertion sort with a time complexity of $O(n^2)$)

- **Batcher Sort:** Batcher's Odd-Even Merge Sort is also efficient for arrays with long runs of sorted or nearly sorted elements. In general, however, the algorithm is considered less efficient than other sorting algorithms such as Quick Sort and Heap Sort.

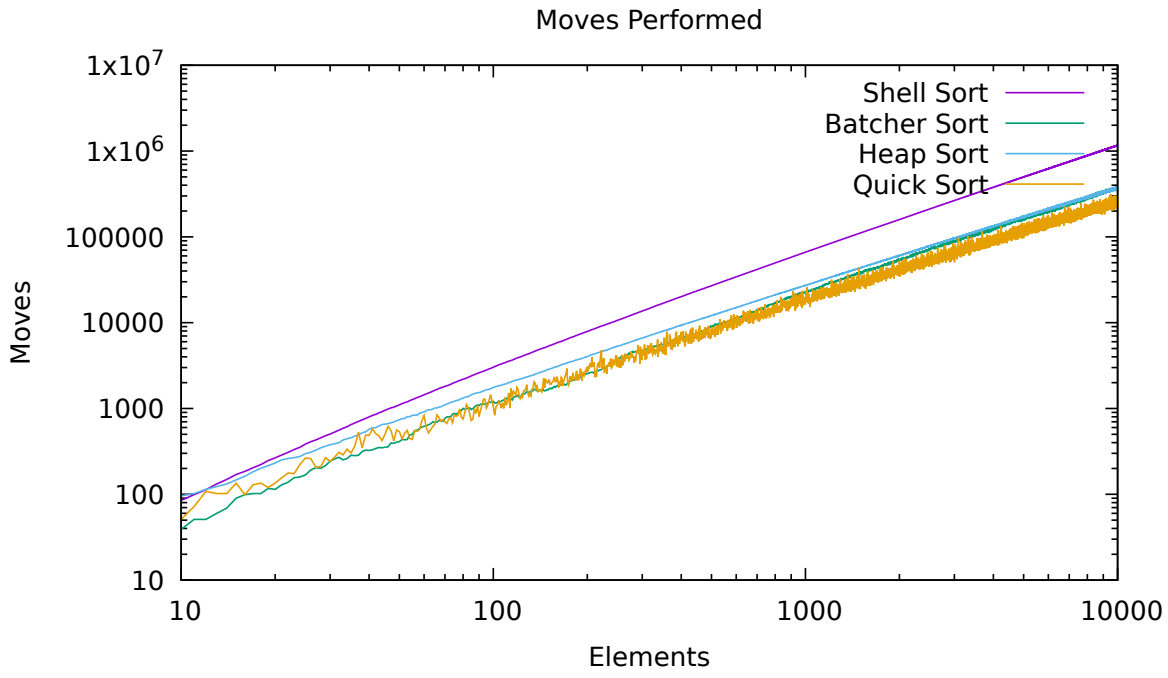

Figure 1: Using different sorts to sort the array filled with random number with number of elements from 10 to 10000

- **Heap Sort:** The algorithm performs well in conditions where the elements in the array are randomly ordered, or the array is partially sorted. Heap Sort is also efficient for arrays with a large number of elements, making it a good choice for sorting large data sets. Heap Sort is generally considered more efficient than other sorting algorithms, such as Shell Sort and Batcher's Odd-Even Merge Sort.
- **Quick Sort:** Quick Sort performs well in conditions where the elements in the array are randomly ordered or the array is partially sorted. The average time complexity of Quick Sort is $O(n \log n)$, making it one of the fastest algorithms for sorting large data sets. Although heap sort has the same time complexity of $O(n \log n)$, Quick Sort is generally considered more efficient because it has greater constant than other sorting algorithms, such as Shell Sort and Batcher's Odd-Even Merge Sort.

2 Performance Graphs

2.1 Figure 1 Analysis

Figure 1 is generated by going through different arrays with 1 - 10000 elements with default seed. As we can tell from the graph, shell sort becomes relatively ineffective as we increase the number of elements in an array, while other sorting functions perform relatively well. On the other hand, Quick sorting can handle an extensive database better than Heap sort and Batcher sort, and Heap sort is slightly worse than Batch sort when handling smaller arrays; however, both Heap sort and Batcher sort are effective for handling larger arrays as their lines tend to merge as the number of elements increases. Shell sort's line is relatively smooth since its moves increase linearly as the number of elements increase. However, the number of moves in other sorts does not increase linearly as the number of elements in the array increases because, in each iteration of the algorithm, the number of elements to be sorted is reduced by half on average.

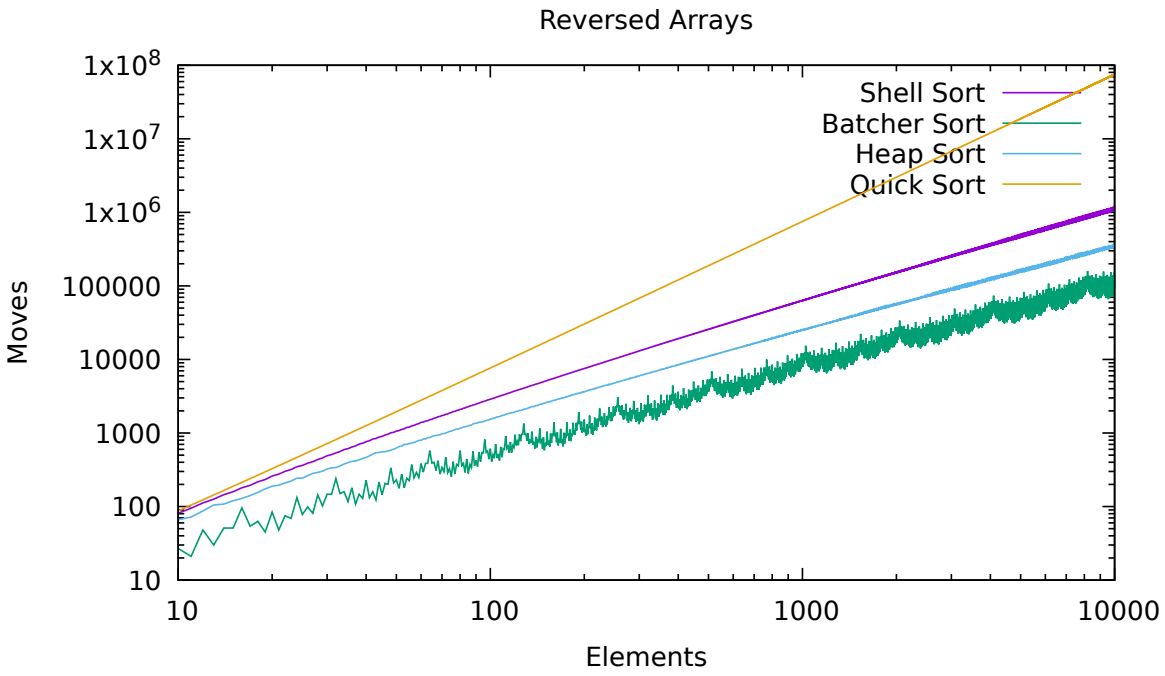

Figure 2: Using different sorts to sort the array in reversed order with number of elements from 10 to 10000

2.2 Figure 2 Analysis

Figure 2 is generated by sorting through different reversed arrays with 1 - 10000 elements with default seed. As we can tell from the graph, Shell sort, Batcher sort, and Heap sort have similar traits, where Shell sort is relatively ineffective, then Heap sort, and Batcher. However, Quick sort becomes the worse sorting algorithm for this situation. When the array is reversed, the pivot element will always be the largest element. In each iteration, the partition will always be unbalanced, with one sub-array having only one element and the other sub-array having all the remaining elements. This results in a degenerate case where the time complexity of Quick Sort becomes $O(n^2)$ (The same as insertion sort), making it less efficient than other sorting algorithms such as Heap Sort and Shell Sort.