Analyzing Algorithms

Isha Chury

February 5, 2023

Description of program:

Within computer science, one of the most prominent uses for functions is sorting through various sets of data. Defined as the act of systematically arranging sets of information, sorting is implemented across various platforms, from organizing internal information to implementing various user-experience features. While sorting may be implemented easily for smaller sets of input numbers, larger sets of data inputs require a considerable amount of time to compare and process the results. To navigate through this problem, there are four different algorithms centered around sorting: batcher sort, heap sort, quick sort, and shell sort. Each algorithm uses a different mathematical approach to parse through large sets of information, and arrange them accordingly. Though it can be noted that there are different types of algorithms that may be implemented in the place of these four, we are observing the implementation and results of these four algorithms specifically. We will be observing how many moves—referring to the algorithm's number of data shifts—each algorithm requires, and analyzing them under various conditions.

Complexity

Before we address the technique each algorithm utilizes, we must address the topic of space and time complexity. Though we won't be examining this topic in depth, it is a crucial aspect of these four algorithms, and contributes significantly to their respective performance. Space complexity refers to the total amount of space required—inclusive of the input and outputs generated—within memory. Memory must be allocated with respect to the program. However, as the memory allocation increases, the time for the program's completion increases. This is referred to as time complexity, the total amount of time the program requires to complete. These two complexities are intertwined through a positive, increasing relationship.

Shell Sort:

Shell sort employs the use of gaps, a factor by which the algorithm compares elements at a distance. Instead of comparing items in close vicinity of each other, shell sort essentially takes the factor, adding it from the zeroth element, to find the next element that needs to be compared. For example, if an array contained twelve elements, and the gap was initially placed at a factor of two, the zeroth element would be compared to the sixth element. If the zeroth element was greater, the numbers would be swapped. This would be a similar case if the sixth element was less than the zeroth element. This process would continue until the final element of the array is compared (and swapped if necessary). The gap would then be divided by the factor. This process would continue until the gap reaches a factor of one.

Batcher Sort:

Batcher sort, unlike shell sort, uses a system of sorting networks to compare various values. Sorting networks expressly take various inputs—defined within the function's parameters—to be compared. As these networks interact, the values contained within each network are interpreted and analyzed. If the value of one network is higher, the numbers are subsequently swapped. This process continues until the values are placed in ascending order. While there are usually express limits on the number of inputs and subsequently, the sorting networks, a slight modification to the code allows for arbitrary input values to be compared. Sorting networks by definition are required to use inputs that are powers of two. However, with the modifications present in the pseudo-code, this specific requirement no longer applies.

Quick Sort:

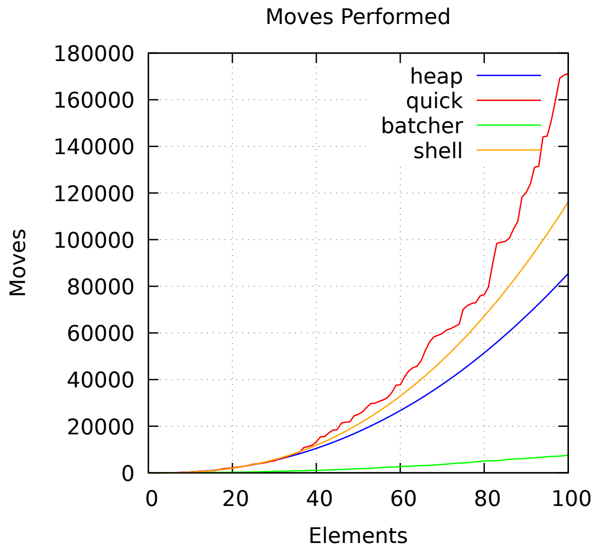
Quick sort uses a process similar to shell sort, using a factor to divide the data set. However, the factor within this algorithm is strictly set to two, partitioning the data into two sub-data sets based on a pivot point. The pivot point is chosen by the algorithm, and is used as a baseline to compare all other values within the data set. The two sub-data sets—referred to as the left and right subsets—are compared with the pivot point. If the data is greater than the value of the pivot point, then the value is transferred to the right subset. Similarly, if the data is less than the value of the pivot point, the value is transferred to the left subset. The process continues until the entire array is sorted.

Heap Sort:

Heap sort uses two different processes: building a heap and fixing a heap.

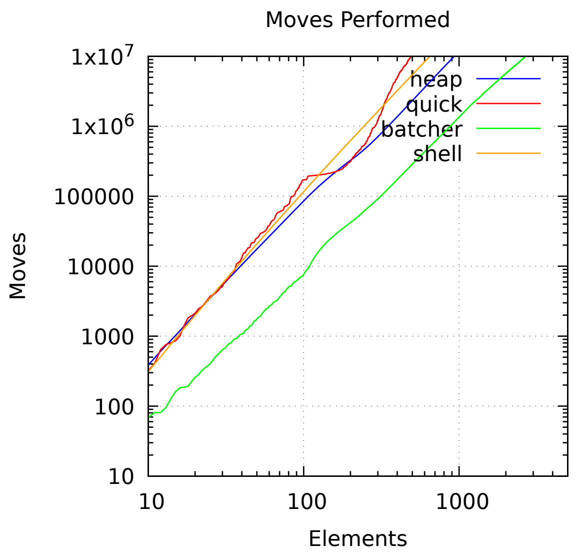
However, to understand this, we must first look at a max heap. A max heap refers to a binary tree in which the parent node's value is greater than that of its children. Each parent node must fulfill this requirement for the structure to be considered a maximum heap. Within this structure, the heap algorithm employs the first process, building the heap. This process arranges the elements within the structure of the maximum heap, obeying its overarching constraint regarding the parent node. Once the elements are arranged within the heap, the second process occurs. The second routine sorts the heap, using it's structure as a guideline to find numbers of higher orders.

Graphs



The graph depicts the moves utilized by various algorithms in response to sorting a randomly generated sequence of numbers. Observing each of the algorithm's respective set of moves, we can see that the Batcher algorithm is the most efficient in comparison to the others. While the other three algorithms used moves upwards of eighty-thousand, Batcher's highest set of moves for a hundred elements barely approached twenty-thousand. We can see a startlingly high disparity between the moves required by Batcher compared to the other three algorithms. While Batcher is the most efficient, Heap is the second most efficient in comparison to the other two algorithms. Heap did rise exponentially as the number of elements increased, but did not rise as significantly as Quick or Shell. While the disparity between Heap and

Shell was not as significant, Shell utilized more moves than Heap. However, when comparing each of these algorithms to Quick, there is a glaring difference in how Quick behaves. Quick rapidly increases and decreases, creating spikes in the graph. This does not occur to any other algorithm, though it may be due to the recursive nature of the function. Quick, perhaps as a result of utilizing recursion, required the most moves out of any of the algorithms. While Quick functioned similarly to Heap and Shell in its initial phases—when the number of iterations was low—it increased in moves significantly as the number of iterations increased.



Conclusion:

Shell sort is optimal for comparing values at a significant distance. Though it does eventually reach elements that are adjacent to each other, the process required to compare adjacent values takes time to reach. However, values across larger distances are easier for this algorithm to analyze and subsequently swap, if in a necessary case. However, this algorithm is predominantly based around the gap chosen. The size of the gap contributes to the space and time complexity of the algorithm. However, in the given instance that shell sort is used on a sorted array, the algorithm is incredibly efficient giving a time complexity of nlogn. This specific complexity utilizes the least amount of time in comparison to complexities of various powers.

Batcher sort is the most optimal algorithm for sorting large amounts of information. Despite the increasing amount of information processed by the algorithm, the number of moves remained relatively low in comparison to the other three algorithms. This suggests a complexity of n log n, which is optimal for analyzing and comparing data. While the other algorithms may follow a similar pattern—in their best cases—Batcher sort's time complexity suggests it is the most efficient. Though this particular algorithm may have a bad case, similar to its counterparts, it can be noted that it is currently one of the most efficient to utilize, regardless of the number of iterations required.

Quick sort is optimal for comparing values that are adjacent to each other. Though the algorithm can sort elements that are further away from each other, the algorithm is more efficient with elements that are significantly closer to the pivot point. This decreases the amount of moves required, creating a complexity of about n log n, at its very best case. However, there are instances where the pivot can be an extreme of being the smallest or the largest. This specific case generates an increasingly large number of moves required for the algorithm to process.

Heap sort, similar to quick sort, is an efficient algorithm that is optimal when comparing slightly sorted sets of data. This is largely due to the data structure it employs. If the heap is already "sorted" in terms of the maximum heap requirements, then the second process takes far less time and moves to proceed. However, this is strictly considering a good case, in which a maximum heap is already formed. Taking into consideration, a possibly normal case—in which the heap is not sorted—the time complexity increases significantly. The moves required to create the heap and furthermore, order the heap take much longer to create and analyze.