CSCI 2210 - Project 2
Algorithm Analysis

Adam Hooven

September 25, 2024

Introduction

Algorithmic Analysis is a foundational aspect of Computer Science curricula as it enhances problem solving skills by reinforcing critical thinking and abstraction of complex challenges into solvable problems. There are real-world benefits as the performance of software doesn't rely solely on the hardware provided and peak performance will only be achievable through intelligent design/choice of algorithms, and this only becomes more relevant as technology advances so too do the problems we attempt to solve using it. Therefore the evaluation/understanding of algorithmic design and performance is necessary for well-rounded, skilled computer scientists. One such "problem" in algorithms is Sorting which forms the basis for the lab in which we set out to perform a comparative analysis of iterative and recursive sorting algorithms performed on both value and reference data types using data sets in a range of sizes to gain a better understanding of the computational complexity of these operations.

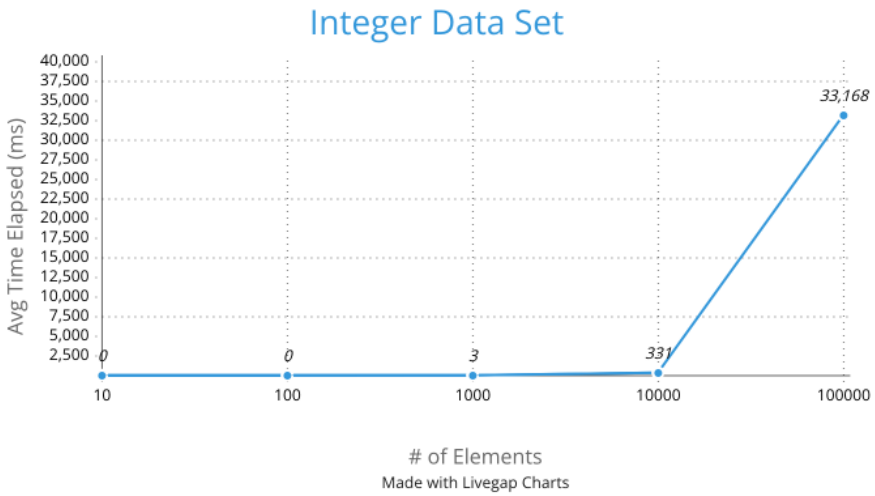
Iterative Sorting Algorithm - Bubble Sort

Iterative Sorting Algorithms are named so due to the fact that they will loop through a collection of data repeatedly in an incremental fashion (*iterating* as it were). For this lab I implemented BubbleSort as my Iterative choice. BubbleSort works by stepping through the collection and performing comparisons of adjacent values and if a comparison finds that the values should be swapped, the swap will be performed and the iteration continues until all values in the collection are sorted correctly.
BubbleSort is an effective algorithm for sorting very small data sets however the time complexity of BubbleSort is $O(n^2)$ (Quadratic Growth) this is due to the fact that BubbleSort will loop through the collection n^2 amount of times where n corresponds with the size of the input. See below:

Algorithm 1 BubbleSort(A, n) (Cormen et al., 2022 p. 46)

```
1: for i = 1 to n - 1 do
2:     for j = n down to i + 1 do
3:         if A[j] < A[j - 1] then
4:             exchange A[j] with A[j - 1]
```

From this pseudocode we can see that each loop will run an n number of times and since the loops are nested this leads to $n \cdot n$ aka n^2.



This was observed when running the Bubblesort algorithm on the data sets (both Books and Ints) the average runtime increased quadratically with given input (see above for Integer Data Set - In Sorted Order). The most surprising thing for the Iterative sorting algorithm was that its worst performance on average was with Randomly Sorted Integers and not Reverse Sort as I expected. I attribute this to possibly requiring more comparisons/swaps from the random list than the reverse order would necessitate. Comparatively BubbleSort did outperform the Recursive algorithm on the smaller data sets although the difference was minuscule enough that the speed had to be measured in Ticks. The smaller data sets (sub 1000 elements) are where BubbleSort shined, generally outperforming QuickSort for both data sets. In the case of Books which are reference types BubbleSort at all points took longer than Ints however I assume this comes from the need to perform potentially 4 times as many comparisons for the integer data (Last Name, First Name, Title, Release Date).

Recursive Sorting Algorithm - QuickSort

Recursive Algorithms differ from Iterative with respect to sorting as they implement a style of problem solving referred to as "Divide-and-Conquer" by which collections are broken into smaller sub-collections repeatedly, and then performing the sorting algorithm process on each sub-collection again. The benefits of sub-dividing, or partitioning, the collection is a significant increase in performance compared to Iterative algorithms at scale. The algorithm I chose to implement was QuickSort, which will select a value in the collection referred to as the pivot. The collection is then partitioned around the pivot and values that precede the pivot are placed into the "left" subcollection and values that should follow will be placed into the "right" subcollection. This ensures that the pivot is in the correct place and afterwards a new pivot is selected. This algorithm is considered better than Iterative algorithms because on average the time complexity of QuickSort is $O(n \log_2 n)$. The growth of the algorithm is reigned in when n begins to scale up due to that logarithm as compared to Iterative's n^2 factor.

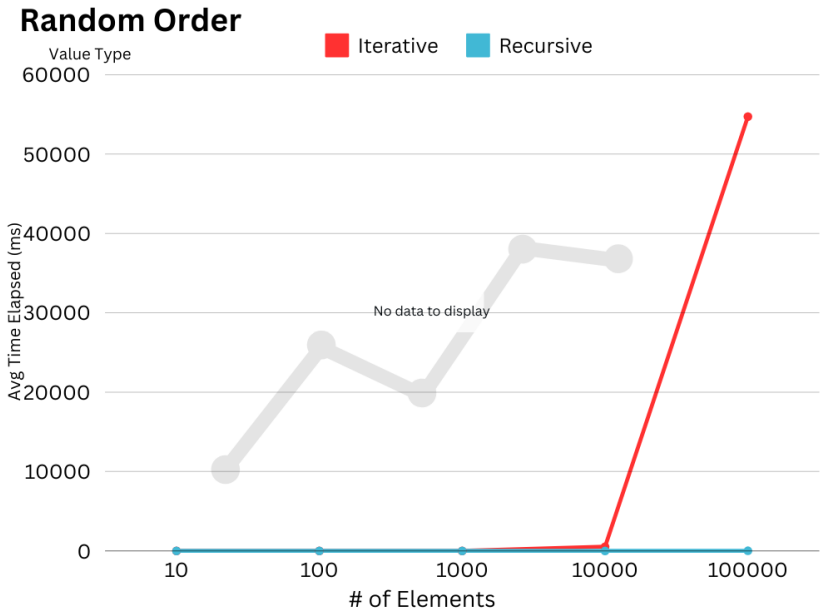
Algorithm 2 Quicksort(A, p, r) (Cormen et al., 2022 p. 183)

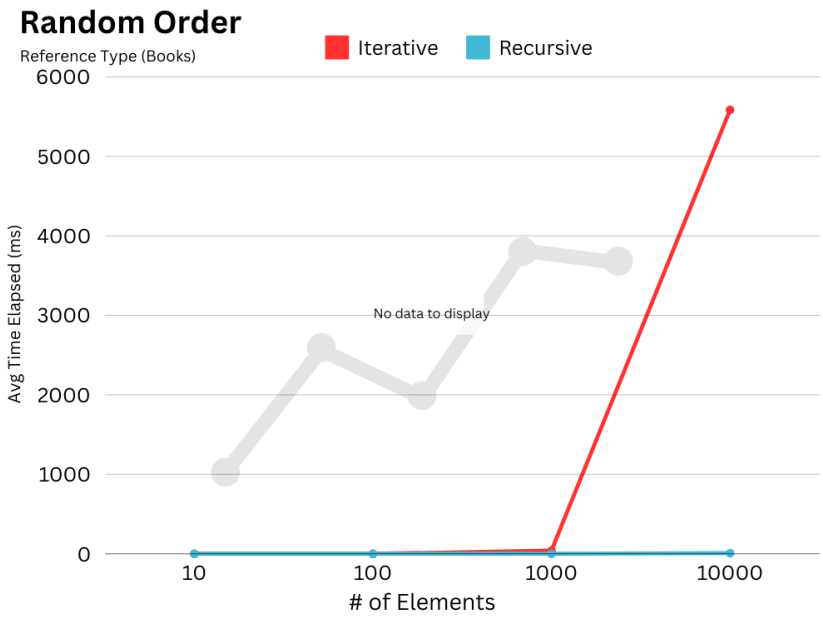
```
1: if p < r then
2:     q = Partition(A, p, r)
3:     Quicksort(A, p, q - 1)
4:     Quicksort(A, q + 1, r)
```

Algorithm 3 Partition(A, p, r) (citeCLRS p. 184)

```
1: x = A[r]
2: i = p - 1
3: for j = p to r - 1 do
4:     if A[j] ≤ x then
5:         i = i + 1
6:         exchange A[i] with A[j]
7: exchange A[i + 1] with A[r]
8: return i + 1
```

The Recursive algorithm demonstrated vastly superior performance for sorting the data sets as expected. In the tests run on both Integer and Book data the only 2 instances where QuickSort's average time was high was in the case of 100,000 integers in Reverse Order and In Order. This is expected as QuickSort's worst case runtime scenario occurs when partitioning the array creates one partition of size 0 and another of size n - 1. In my implementation of QuickSort I did not add any optimizations around pivot choice and opted for selecting the right-most item of the collection which in both cases would end up subdividing the array into one with 0 elements as either I have the greatest element or the least element causing "side" disparity. QuickSort performed the best in scenarios where the data set was large and in either random or "almost in" order as the partitioned collections would not be expected to be heavily one sided. As mentioned earlier QuickSort did experience higher average times for Books compared to Integers however I believe this is also a result of needing to perform more comparisons that may be necessary and more computationally intensive for comparing strings versus integers.





The above two graphics show the comparisons between the two algorithms on both data types using Random order data. As expected the recursive algorithm's growth was significantly lower than that of the Iterative once the number of elements began to grow quickly.

Conclusion

From the data provided above this experiment illuminated the performance benefits of implementing recursive algorithms in place of iterative algorithms .In essence, it is of paramount importance for not only computer science researchers to study the design of Algorithms but also developers in real-world scenarios. Without this knowledge we would not be able to make informed decisions that would lead to efficiently designed code causing us to waste time/resources on absurdly intensive computations which divert from solving actual problems or performing more important functions. These skills and practices can be extrapolated and applied to other problems in non-Computing domains.

References

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th). MIT Press.