Sorting Algorithm Analysis

Brent Jang — CPSC 350 - Section 2

I. Algorithm Explanation

A. QuickSort

QuickSort is a popular divide and conquer sorting algorithm that can be broken down into three parts:

- Divide the array
- Use recursion
- Concatenate the array

By using a pivot, QuickSort divides the original array into two (or possibly three) arrays. Usually, the pivot is the last value in the unsorted array. Values smaller than the pivot are put into an array L and values larger than the pivot are placed into array R (values equal to the pivot are put aside in array M).

The algorithm uses recursion to continue to divide the arrays L and R until they cannot be split anymore, this signals that all the values are sorted.

Lastly, a simple concatenation is used to put the arrays in a singular sorted array. The order of concatenation is: L, M, R.

QuickSort loops through every element of the arrays to partition them, making runtime $O(n)$, but also uses the divide and conquer method over $O(log(n))$ loops. Therefore, QuickSort has a runtime of $O(n * log(n))$ in the average case.

B. Merge Sort

Merge Sort is much like QuickSort in that is has a runtime of $O(n * log(n))$ and it uses a divide and conquer algorithm.

- Divide the array
- Use recursion
- Concatenate the array

It differs from QuickSort as it does not use a pivot to determine a split into other, temporary arrays. Instead, Merge Sort continuously splits arrays until every element is a singular value. Then it resolves each sub-problem by comparison and merges the values back into sorted the sorted array.

The time complexity of this algorithm is derived from the iterative method over the number of times it has to do so. If there are n elements, the algorithm must iterate over all of them, but only $log(n)$ number of times because of the divide and conquer method. Therefore, the runtime of Merge Sort is $O(n * log(n))$.

C. Bubble Sort

Bubble Sort is much different than the past two sorting methods as it uses an iterative swapping method to sort through an array. The steps needed to apply this algorithm are:

- Iterate through the array
- Stop at the first pair of unsorted values and swap them
- Continue until the array is sorted

Starting at the beginning of the array and working down it, the algorithm must keep track of the value at index i and $i+1$. Once it finds a pair of unsorted values, it swaps them. It continues comparing the number with the proceeding number until the value is in the right spot.

This algorithm runs for each element n and does so n number of times. Therefore, the runtime of Bubble Sort is O(n^2).

D. Insertion Sort

Insertion Sort has the same time complexity as Bubble Sort, but outperforms it in most cases. This is because insertion sort requires less swaps to sort an array. The steps needed to apply this algorithm are:

- Iterate through the array
- Stop at the first unsorted value and place it in the correct position of the sorted part
- Continue until the array is sorted

Insertion Sort also iterates through the array until it comes to an unsorted value. It then takes the value n and compares it to the other values in the sorted part of the array. If n is less than the last value in the sorted part, the last value is shifted to the left. This process continues until the correct index for n is found. This algorithm continues until all values are sorted.

The difference between Bubble Sort and Insertion Sort are that Insertion Sort works quicker for almost sorted data and performs the same sorting, with (usually) less number of swaps. Insertion Sort also has a runtime of $O(n^2)$, therefore, it is not usually suitable for sorting large lists.

II. Personal Findings

After running each algorithm with a large data set, I was pleased to find that the algorithms performed how they are described in terms of functionality and time complexity. While my analysis is limited to using a built in package in C++, I could clearly see how each algorithm performed over a large data set. One major short coming that I had was that my computer was not able to process as large of a data set that I would have liked. I found that trying to sort over $!0^4$ values put a lot of strain on my computer and crash it multiple times.

The difference in the times that it took for each algorithm did impress me in seeing how large the disparity is between algorithms. While some algorithms are definitely more difficult to navigate and understand, it is worth the extra effort and research to understand which algorithms correctly fit your needs.