Jason Klamert

March 8th, 2016

Galina P.

Project 2 Analysis

Analysis of Sorting Algorithms

This analysis will cover selection sort, insertion sort, bubble sort with swaps counting, bubble sort without swaps counting, merge sort, and quick sort. Since you come from a computer science background, I need not explain what the algorithms do. However, I shall remind you of some definitions that will be used in this paper. When an algorithm is ‘stable’ it means that the algorithm will preserve the order of duplicates that occur in the input. Secondly, an algorithm is ‘in-place’ when input items are sorted within an original array with at most a constant number of items stored outside the array. Lastly, an algorithm is ‘adaptive’ if it performs the same sequence of steps for each type of input whether it is sorted, random, or reverse sorted. That wraps up the introduction and now we can move on to the theoretical efficiencies of our algorithms in question.

To begin with, we shall cover selection sort. I predict that selection sort will have an algorithmic efficiency of T(n) = Big Theta(n^2). Selection sort must go through n^2 elements to sort the list no matter what kind of input is given to it. This makes selection sort non-adaptive, which may come in handy later. In addition, selection sort does not preserve the order of duplicates which makes the algorithm unstable. However, the algorithm is in-place because it sorts using only the original array and some constant number of storage space not proportional to the size of the input. Next up we have insertion sort with a predicted efficiency of T(n) = Big Theta(n) in the best case and T(n) = Big Theta(n^2) in the worst case. The algorithm preserves the order of duplicates, changes how it reacts with variation of type of input, and it sorts using the original array and a constant number of storage spaces. This makes insertion sort stable, adaptive, and in-place. Bubble sort (without swaps counting) will be our next algorithm and it has a predicted efficiency of T(n) = Big Theta(n^2) for all cases. This is because bubble sort (without swaps counting) must go through n^2 operations to produce a sorted array. This algorithm is non-adaptive, stable, and in-place. Logically, we follow with bubble sort (with swaps counting). I predict that the efficiency of bubble sort (with swaps counting) will be T(n) = Big Theta(n) in the best case and T(n) = Big Theta(n^2) in the worst possible case. I say this because the best case features an array that is already sorted and the worst case must wade through n^2 operations to sort an array that is reverse sorted. Like bubble sort (without swaps counting) bubble sort (with swaps counting) is both in-place and stable. However, bubble sort with swaps counting is adaptive due to the algorithm having the capacity to detect when an array is sorted. Lastly, we will touch on merge sort and quick sort. These two algorithm have similar complexities and they both are predicted to have T(n) = Big Theta(nlg(n)). Quick sort is unstable, adaptive, and not in-place. Due to quick sort being adaptive it has a worst case complexity of T(n) = Big Theta(n^2). Merge sort is stable but it is not in-place or adaptive. Now we shall segue to the experimental data now that our predictions have been made.

Please find the charts included with this report and flip to the chart labeled “Random Array (100-1000) Sorting Times.” This chart contains the experimental data for our sorting algorithms which are ran with arrays that consist of completely random elements of size one hundred to one thousand elements. I should note that this type of input will cover the average case complexity for most of our algorithms such that best case < average case < worst case in terms of running time. I hope you notice the trend lines that are included on the charts. These denote the actual trends that the data follows. A linear trend line denotes that the algorithm follows a time complexity that is T(n) = O(n). Similarly, a polynomial trend line regards a curve with a complexity of T(n) = Big Omega(n) and T(n) = O(2^n). Luckily, bubble sort without swaps counting always runs at T(n) = Big Theta(n^2). Thus, we can use this algorithm’s running time to make experimental judgements of our other algorithms. Bubble sort without swaps counting using a one thousand element array takes 4,091,549,400 nanoseconds to be sorted. Please take a minute to look at the table used to generate the chart. Notice that selection sort is predicted to have a complexity of T(n) = Big Theta(n^2) but the running time is significantly shorter than bubble sort without swaps counting. I believe that this is due to the sheer amount of memory overhead due to the large random integer sizes of one to ten thousand. These are accentuated by the nature of how bubble sort swaps elements to sort the array. If we use 4,091,549,400 nanoseconds as our rough estimate for a running time of n^2 then we may apply this to our other algorithms run times and derive that our experimental results often fall in between the complexities of T(n) = Big Theta(n^2) and T(n) = Big Theta(n). This includes bubble sort (with/without) swaps counting, insertion sort, selection sort, merge sort, and quick sort. Additionally, we may choose to make some judgement off of merge sort’s time complexity. Due to the nature of merge sort it is extremely consistent in run times. It is non-adaptive and sticks to run times very close to nlg(n). This is how we will judge if results fall into the category of linear time complexities. Using the above logic I have determined that my experimental results will yield that bubble sort without swaps counting, bubble sort with swaps counting, insertion sort, and selection sort are T(n) = Big Theta(n^2) due to their running times being quite close. I should note that insertion sort and bubble sort with swaps counting are adaptive and they only end up in this category due to the nature of the input being completely random. I have made this judgement on the basis that 10,000^2 is approximately equal to the run times of these algorithms and I will credit any overage to computational overhead. Merge sort and quick sort are much harder to pin down. If we take ten thousand times binary logarithm of ten thousand then we should approximate a running time of both quick sort and merge sort. If we do this we end up with approximately 132877.1237954 nanoseconds. If we cross reference the run times, we will find that they are a little higher than this. First off I had to multiply these values by 60 to get them to register as non-zero on the graph and second of all they are recursive functions. This means that there is considerable overhead when maintaining the memory stack, especially for large numbers of recursive calls. However, we could attribute this to margin of error to system load as well. It is hard to say when running algorithms on a relatively widely used server. So, we will say that they are the same as predicted at T(n) = Big Theta(nlg(n)) for our sake.

Moreover, the same logic will be applied for the comparison of running times of our algorithms with sorted arrays of size one hundred to ten thousand. In an effort to shorten this analysis I will give my classification and briefly go over some interesting points about the classification. Bubble sort with swaps counting and insertion sort are both going to fall into the T(n) = Big Theta(n) category. This is due to the adaptive nature of the algorithms and they recognize when an array is already sorted. Meanwhile, merge sort and quick sort will remain slightly above that at T(n) = Big Theta(nlg(n)). Selection sort and bubble sort without swaps counting will remain at the classification of T(n) = Big Theta(n^2). It should be noted that selection sort was actually faster than anticipated for this trial.

Lastly, we will run the algorithms with partially sorted arrays of size one hundred to ten thousand. Interestingly enough, the same trends hold for this trial with one exception that I did not foresee. Bubble sort with swaps counting should have had a faster run time than what is listed. This is a significant contradiction to what was predicted. Similarly, selection sort had a rather quick run time as well. In classification, merge sort and quick sort are still T(n) = Big Theta(nlg(n)). Bubble sort without swaps counting still sits at T(n) = Big Theta(n^2). This leaves insertion sort at T(n) = Big Theta(n) and bubble sort with swaps counting and selection sort between T(n) = Big Theta(n^2) and T(n) = Big Theta(nlg(n)). In conclusion, most of the predictions have come to fruition. However, some key outliers have surfaced and this warrants some further experimental trials. Additionally, all final run times have been the average of three trial run times and for times significantly small have been multiplied by a sufficient amount to make them recognizable on the chart.