The City College of New York

Sorting Algorithms

Jonathan Tran

Algorithms

CSC 22000 Spring 2018

Professor Yuksel

4/9/18

Objective: Implement Insertion sort, Merge sort, Heapsort, Counting sort, Radix sort, and Quicksort using preferred programming language. Run each sorting algorithms using different problem sizes while keeping track of its completion time. Use this data to determine how these algorithms running time changes depending on problem size, and how each algorithms compare to each other.

Language: Using Java with IntelliJ IDEA Community Edition 2017.3.4 x64.

n-elements	Insertion Sort	Merge Sort	Heapsort	Counting Sort	Radix Sort
10	4217 ns	9638 ns	22588 ns	3313 ns	4518 ns
100	53006 ns	73486 ns	289729 ns	12650 ns	20480 ns
1000	1904620 ns	657462 ns	850212 ns	109928 ns	301173 ns
10000	29.7901 ms	2.2192 ms	4.2775 ms	1.0794 ms	4.6185 ms
100000	947.4658 ms	17.9265 ms	23.5774 ms	10.2796 ms	19.1839 ms
1000000	104857.0193 ms	123.3307 ms	146.3802 ms	52.7493 ms	90.8074 ms

n-elements	QuickSort (First-element)	(Middle element)	(Random element)
10	5722 ns	6626 ns	312317 ns
100	33430 ns	35840 ns	1048685 ns
1000	341531 ns	593011 ns	5452744 ns
10000	1.5282 ms	1.8872 ms	12.4303 ms
100000	9.8408 ms	10.5122 ms	111.0426 ms
1000000	81.7406 ms	83.4729 ms	954.6367 ms

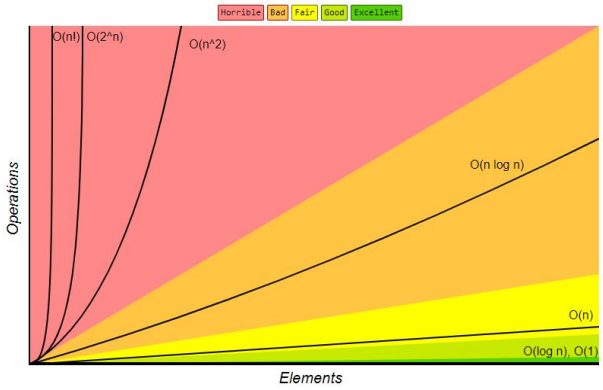

Image taken from http://bigocheatsheet.com/

Insertion Sort:

Insertion sort time complexity is θ(n^2), and is an in place algorithm. Assuming the array is in reverse order. This is due to it having an inner loop do i - 1 comparisons. For each ith element it must loop through the array and compare each value. If the array was is sorted order, the outer for loops would still have to iterate through the array making it θ(n). Since Insertion sort is not a divide and conquer algorithm. For smaller problem sizes it will have a faster completion time; however, if the problem size were to increase. The time it takes to completely sort the array would increase greatly. Looking at the chart above, It looks like as n increases so does that time it takes to finish the algorithms. Thus making it slower for larger problem sizes.

Merge Sort:

Merge sort is a divide and conquer algorithm. It has a time complexity of θ(nlogn), and is not an inplace algorithm. That is because Merge sort creates two new sub arrays for the merging portion. This algorithm divides the array to its base case which is 1. It then merges the left and right sub arrays into an ordered array. Drawing it out on a sheet of paper, gives an image of a upside down tree. It must complete the left portion of the sub trees before it can move on to the right. Since it has a time complexity of θ(nlogn), it should yeild faster results when n-size increases. As you can see in the chart insertion sort is faster when n is smaller however once n increases Merge sort becomes considerably faster.

Heapsort:

Heapsort has a time complexity of O(nlogn) and is an in place algorithm. The notation is big O because it gives the upper bound. The algorithm can stop anytime before nlogn. This algorithm uses the structure of a upside down tree. Max heapify, the main portion of the algorithm has a time complexity of O(logn). If we were to placed the mininume on the root, the minimum would have to travel down the height of the tree, which is why time complexity is O(logn) and each element does logn work. Heapsort has a time complexity of a divide and conquer algorithm but it doesn't behave like one. Comparing it with Merge Sort both have similar completion times when n >= 1000. When comparing it to insertion sort, Heapsort is better for larger problem sizes.

Counting and Radix Sort:

Counting and Radix are somewhat similar, due to Radix sorting using a modified counting sort code. Counting sort has a time complexity of θ(n+k) or θ(n). Its sorting algorithm is simple. It sorts the array based on the number of occurrences of a number. One big problem with Counting sort algorithm is the memory usages. In order to determine the range of the counted array it needs a max value. If the max value was 700, that means it requires a counted array of size 700, most of it will be filled with zeros. Depending on the max value, counting sort may not be the best algorithm.

Radix sort uses Counting sort as a subroutine. Its time complexity is θ(d*(n+b)). Radix sort determines the number of loops needed to sort the array based on the max value. If the max value was 700, it would need to loop through the array three times for each decimal places; in addition, the memory usages for the counted array would be of size 10. So memory usages for Radix sort is better than counting sort.. Radix sort is merely sorting the numbers based on decimal places for each loop. One problem with Radix sort is that if max value is 700 but all the other numbers are in ranges of 0 - 9, then radix sort is doing more loops just to sort the rest of the data. Both Counting and Radix sort is not inplace. Based on what is seen in the chart, depending on the ranges Counting sort seem to have better times then Radix sort

if memory usage is ignored. Counting sort also seems to be faster than the other algorithms even being insertion sort for smaller problems. Radix sort times seem comparable to the other algorithms but falls short of Quicksort.

QuickSort:

Quicksort has an average case of θ(nlogn) and is an in place algorithm. And since pivot can be modified the worst case of n^2 rarely happens. Quicksort is a divide and conquer algorithm in which a pivot index is chosen and numbers less then pivot will be sorted to the left and larger will be sorted to the right. Looking at the charts, it would seem that Quicksort does slightly better than Merge and Heap sort. Quicksort even looks comparable to insertion sort when n-size is small. Counting looks to be close to Quicksort, if we ignore counting sort's memory problem. When the first or middle element is picked as pivot. The completion times are comparable. However, randomized pivot seems to have large effect on the algorithm's running time. The random numbers being generated may have impacted the performance of the Randomized Quicksort.

Conclusion:

All these sorting algorithms have efficiently in one area while sacrificing efficiently in other areas. There is no best sorting algorithms, each one has its own purposes. Based on the charts alone, counting sort seems to be better in all areas of problem sizes based on speed. However, counting sort uses a lot of wasted memory based on range. Quicksort seems to be one of the more faster algorithms. Based on its times alone. It seems to be the fastest, if we ignore Randomized Quicksort. But since the data is randomly generated, I can't make a generalization like that. All these sorting algorithms have their place, and it would all depend on the situation.