Chayton Hamric

Professor Dianne Foreback

Data Structures

18 November 2019

Project 4

This project comparing different sorting algorithms based on different data sets, and number of elements. The four data sets are a predefined file input, randomly arranged vector of integers with N elements, vector of N elements containing ascending values, and a vector of N Elements with descending values. N was tested for 10,000, 100,000, 1,000,000 elements in each vector.

Heapsort.h - Code for the heap sort function. (This was copied from the book as instructed) Mergesort.h - Code for the merge sort function. (This was copied from the book as instructed)

Quicksort.h – Code for quicksort, the code was modified from the books code, to do quicksort

throughout the algorithm without degrading into another algorithm.

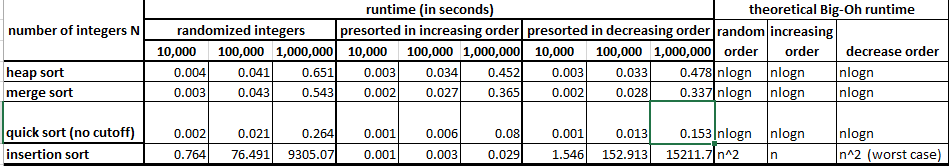
Insertion.h - Code for the insertion sort function. (This was copied from the book as instructed)

Sorting.cpp - Driver file for testing algorithms and analyzing run times. Contains Main

Function and function calls.

Project4.pdf - This file containing my report.

Comparison sheet:



Heap sort is sorting that is based on binary Maximum heap data structure. To build the max heap the runtime would be O(N). By using Delete max the array will be sorted. It returns the deleted max item to the last spot of the array and swaying the current value position. Then decreasing the last data point of the array pointer by one memory location of the array. It then puts the new array in the heap. This time takes O(logN) time and it does it N times for each of the N elements. This makes the Total runtime O(NlogN) in all situations, it does not matter if the arrays was presorted or not. Which explains why all the run times for Random, presorted ascending and presorted descending is close to the same time, for their specific element size. It also explains why when the size of the data set is increased the time is not heavily punished do to the growth rate only growing at a NlogN rate.

Merge sort is a divide in conquer algorithm that is at O(NlogN) runtime, the input it repeatedly cut in half, which takes O(logN) time and then this happens N times so it is O(nlogn) time, this explains why the times are close no matter the data set, whether random, presorted ascending or descending the time is the same for any N amount of elements. This is why my test times are very close.

Quicksort is a recursive algorithm that runs average O(NlogN)runtime. The worst case is O(N2) and this only occurs when a poor pivot is chosen. The worst pivots are the highest and lowest point of the data set. All the run times are so fast and appear to be close to O(NlogN) time of the heap and merge sort. With randomized being a tad slower than presorted arrays, the difference is next to nothing.

You can see the speed of the quicksort algorithm in how the time change is so mild as the size of the set increases, with none of the sets taking more than a quarter of a second at most. Because of this, the quicksort is the fastest comparison algorithm, the inner loop of the algorithm consists of an increment/decrement (by 1) a test and a jump. There isn’t and extra stuff you have to keep track of as there is in merge sort which contributes heavily to the speed.

Insertion sort is a simple sorting algorithm that consists of N-1 elements, for pass p =1 through N-1 it ensures that elements in positions 0 to p are sorted. This algorithm runs in average case O(N2)time. The best case runs in O(N) time and when the array is already in sorted ascending order. The worst case is when the array is sorted in descended order and every pass consists of an addition to the beginning resulting in maximum number of array movements, this causes the run time to be a O(N2). Massive run times in the set caused this issue. This is why presorted ascending order takes less times, but random and descended orders take a really long time. This is due to the fact that the N2 growth rate causes each factor of 10 to increase the run time.

The biggest difference in runtimes is between quick sort and insertion sort times. While insertion sort is faster in their best case of presorted ascending data, with its run time being O(N) and quicksort being O(NlogN). Also, with array that contains less than or equal to 20 elements Insertion sort performs better than quicksort. Because of quicksort being recursive will occur often.

For random and descending ordered sets when comparing the insertion sort is as extremely slow. For random integers the quick sort took 0.002s for N=10,000 and Insertion sort took 0.764 seconds, as Log10(N) increases by 1 the insertion sort takes 100x longer because of N2 growth, while Quick sort grows much slower at a NlogN. For 100,000 elements, Quicksort took 0.021s while insertion sort took a minute and 15 seconds. For 1 million elements, Quicksort took 0.264 seconds while insertion sort took over two and a half hours. And it is only worse for when insertion sort is at the worst possible case of reverse sorted datasets. For 1 million elements Quicksort took 0.153 seconds, Insertion Sort took insanely time of four hours.

While for smaller or ordered data sets insertion sort can keep up with or be faster than quicksort, but as the data set grows the runtime of Insertion sort grows so incredibly faster than quicksort. In my mind I would not use insertion because of the scalability compared to quicksort.

Heap, Merge and Quicksort all run on simplified O(NlogN) time. But the expanded runtime formulas are slightly different. For example, on page 304 it says that Heapsort will have an average case of 2NlogN – O(N) comparisons, which is really fast compared to Merge and Quicksorts equation of T(N) = cNlogN + N. That results in faster runtime and explains the slight time difference and slowness of heapsort.

This doesn’t explain the difference in times between quicksort and merge sort with quicksort being much faster. This time difference is explained by the fact that the c in the T(n) = cNlogN + N equation means different things in each algorithm. The “c” in merge sort refers to the time required to merge two arrays. While the “c” in quick sort refers to the partition time of two arrays, since that takes a lot less time than merging. This results in quicksort being the faster algorithm.