Alycia Riese

Data Structures Project 3

11-14-18

Abstract

The purpose of this project was to analyze the runtime of four different sorting algorithms: heap sort, merge sort, quick sort, and insertion sort. The algorithms were tested and proven to work using a small test file. Then the algorithms were tested with various numbers of integers and the results were recorded.

File Manifest

- `.vscode` - Folder that contains VSCode information.
- `a.out` - Linux executable file of the program
- `heapsort.h` - Heap sort algorithm from the book with slight modifications
- `insertsort.h` - Insert sort algorithm from the book with slight modifications
- `mergesort.h` - Merge sort algorithm from the book with slight modifications
- `quicksort.h` - Quick sort algorithm from the book with slight modifications
- `runtimeTbl.xlsx` - Table containing the runtimes from each of these algorithms with different number of integers
- `sorting.cpp` - Main code file. All code written is contained in here.
- `sorting.dat` - Data file provided for testing

Analysis

a. For each sorting algorithm, explain the difference in runtimes for the different type of data. That is, why do you think the randomized, ascending, descending data yields different/similar runtimes for the merge sort algorithm; is there an inherent reason with the way the code works? Answer this question for each algorithm; i.e. for heap sort, merge sort, quick sort and insertion sort. Explain please.

   For heap sort, the run time for random, increasing, and decreasing integers was similar. This is because the heap sort still has to construct a heap and remove the largest/smallest element each time, so the run time would be the same no matter which order the integers were in. For merge sort, the run time for random, increasing, and decreasing were similar. This is because the algorithm has to go through the vector and continuously split the elements into separate elements. Then it switches them into the correct order. The runtimes are similar because the switching part does not take up that much time. The quicksort algorithm runtime is slower for random integers because of the

pivot point. In the code I used, the median element was used as the pivolt point. This makes it so that quicksort has to move the smaller elements to left of the pivot point. This causes the runtime for random integers to be slower than the runtime for increasing and decreasing because the code has to sort more values once it picks a pivot point. The insertion sort took the longest; the increasing integers was the fastest, the decreasing integers was the slowest, and random integers was somewhere in the middle. This is because the insertion sort must move a single element to the correct place after finding it. With the increasing integers, the algorithm does not have to move anything, so it runs the fastest. With decreasing integers, the algorithm must move each element all the way back to the front for each element, so it takes the longest. With random integers, the time is somewhere in the middle because it has to move the elements that aren't in order, which could be any number of elements.

b. Explain the difference in runtimes between the insertion sort algorithm and quick sort. Why do you think there are differences/similarities?

   The insertion sort must go through and find the first element that is less than the first element in the vector. This is similar to the quicksort algorithm, if we pick the pivot point to be the first element. The insertion sort takes a long time because if it finds an element that is less than the first element, it must move all the elements to the right by one in order to get the correct element to the first position. Then, this is repeated each time it finds a new smallest element. The quicksort chooses the median element as the pivot point, then it puts all elements smaller than this pivot point to the left of it. This way, quicksort goes through fewer elements; that is why it is faster than insertion sort.

c. Explain the differences/similarities in runtimes between the heap sort, merge sort and quick sort algorithms. Why do you think there are differences/similarities?

   The runtimes for heapsort and mergesort were similar. The runtime for quicksort was faster than the heap and merge sort. This is because the heap sort constructs heaps out of the data and moves the smaller integers to the left of the tree. Then it will re-heapify and repeat the process. This is similar to the mergesort because the mergesort splits the vector into individual elements and swaps them based on which one is bigger. This is why the merge and heap sort algorithms are similar in runtimes. The quicksort is fastest because once it chooses a pivot point, it then sorts the integers that are smaller to the left and the integers that are bigger to the right. This is repeated until the vector is sorted. Quicksort is faster than merge and heap because it goes through fewer elements overall.

number of integers N	runtime									theoretical Big-Oh runtime		
	randomized integers			presorted in increasing order			presorted in decreasing order			random order	increasing order	decrease order
	10,000	100,000	1,000,000	10,000	100,000	1,000,000	10,000	100,000	1,000,000			
heap sort	0.003802	0.045334	0.534499	0.003663	0.044108	0.513172	0.003423	0.042041	0.499638	O(NlogN)	O(NlogN)	O(NlogN)
merge sort	0.003798	0.045693	0.531982	0.002967	0.037022	0.424316	0.002921	0.036574	0.42263	O(NlogN)	O(NlogN)	O(NlogN)
quick sort (no cutoff)	0.002004	0.021112	0.247706	0.000779	0.008281	0.106079	0.001251	0.016465	0.180965	O(N^2)	O(NlogN)	O(NlogN)
insertion sort	0.286299	28.5993	2903.53	0.000195	0.001709	0.017284	0.600199	58.7056	5900.44	O(N^2)	O(N^2)	O(N^2)

Figure 1: Runtime Table