Justin Gnatiuk
Data Structures
Project 3 Report

Summary

For our third data structures project we were to implement different sorting algorithms(Heap,Merge,Quick,Insertion) onto a varying size vector of integers(10,000:100,000:1,000,000) in order to analyze and compare the runtimes of the algorithms given different input sizes.

File Manifest

1. InsertionSort.h : Header file containing InsertionSort Algorithm
2. MergeSort.h : Header file containing MergeSort Algorithm
3. HeapSort.h : Header file containing HeapSort Algorithm
4. QuickSort.g : Header file containing QuickSort Algorithm
5. Sorting.cpp : Driver program for sorting of vectors with different algorithms
6. Project3.PDF : Project Report

Questions

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 the heap sort, runtime of the different input sizes was quick for randomized integers, ascending integers, and decreasing integers. A heap sort utilizes a heap, which is a type of tree, and runtimes for the traversal of trees are optimal, specifically with the heap sort algorithm, which yields a theoretical runtime of O(nlogn) for the three different orders.

The merge sort also achieved quick runtimes for all three orders of integers. Like Heapsort, MergeSort also has a worst case runtime of O(nlogn), with the worst case being a dataset in descending order. Unlike HeapSort, MergeSort utilizes recursion within it's internal functions to achieve it's optimal runtimes.

The insertion sort algorithm proved to be the worst algorithm out of the 4. Yielding a 2415.88 second runtime on a randomized integer vector of 1,000,000. Obviously it's increasing order runtime stayed low like the rest, though it's worst case, decreasing order runtime ended up taking too long for me to be able to wait. This is because the insertion sort is the simplest of the four algorithms, and with it's nested for loop structure, produces a runtime of O(n^2).

The quick sort algorithm with no cutoff provided the best results, proving to be faster than all the other algorithms. Unfortunately, quick sort has a worst case running time of O(n^2) though interestingly enough, it's runtime stayed low for the decreasing order vector which I believe should have been it's worst case. Regarding it's average case runtime of O(nlogn), it achieves this optimal theoretical runtime with a highly optimized inner loop.

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

> There were vast differences between the insertion sort algorithm and the quick sort algorithm in both observed runtime and theoretical runtime. The quick sort algorithm proved much faster than the insertion sort algorithm. Perhaps in a situation where you know for a fact that your dataset sizes will always be small, insertion sort could prove to be a simple solution. Though in the vast majority of cases it is clear that the quick sort algorithm is one of the best to use.

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 quick sort algorithm produced the lowest runtimes of the 3 algorithms, shown in the table below. All yielding the same runtimes of O(nlogn) except in worst case for the quick sort, the merge sort utilizes recursion, the heap sort utilizes a tree structure, and the quick sort utilizes a highly optimized inner loop within it's inner function (3 parameters). Given the quick sort's worst case runtime of O(n^2), it is clear that there are potential situations where a merge sort or a heap sort would be a safer option.

d. Include the table below with you're the runtime for each algorithm and number of elements and the Big-Oh runtime as well. (runtimes are in seconds)

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
	10K	100K	1MIL	10K	100K	1MIL	10K	100K	1MIL			
heap sort	0.003	0.043	0.532	0.003	0.038	0.43	0.003	0.035	0.42	O(nlogn)	O(nlogn)	O(nlogn)
merge sort	0.004	0.044	0.507	0.002	0.031	0.362	0.003	0.031	0.36	O(nlogn)	O(nlogn)	O(nlogn)
quick sort (no cutoff)	0.002	0.021	0.242	0.001	0.007	0.092	0.001	0.014	0.159	O(nlogn)	O(nlogn)	O(n^2)
insertion sort	0.24	24.151	2415.88	0	0.001	0.015	0.488	48.146	Too Long	O(n^2)	O(n)	O(n^2)