CSC206 Algorithms and Paradigms
CSC140 Advanced Algorithms
Spring 2022

Assignment 3 Report

Student Name: Melvin Evans Student ID: 219494692

Part1: Algorithm Comparison

Report the results of your sorting experiments in the tables below. All times should be in milliseconds.

		10000	100000	1000000
Insertion Sort	Sorted	0.567042ms	4.918833ms	26.527253ms
	Nearly Sorted	0.908954ms	1.709282ms	5.796211ms
	Reversely Sorted	35.862453ms	1507.959838ms	493216.501683ms
	Random	15.88687ms	734.815841ms	242580.920983ms

		10000	100000	1000000
Merge Sort	Sorted	5.925564ms	29.011943ms	115.881238ms
	Nearly Sorted	1.697652ms	9.788092ms	59.622595ms
	Reversely Sorted	1.592448ms	9.926492ms	48.73653ms
	Random	2.751262ms	11.444853ms	138.968797ms

		10000	100000	1000000
Quick Sort	Sorted	35.70975ms	Stack Overflow	Stack Overflow
	Nearly Sorted	48.950638ms	Stack Overflow	Stack Overflow
	Reversely Sorted	32.70643ms	Stack Overflow	Stack Overflow
	Random	0.732519ms	Stack Overflow	Stack Overflow

		10000	100000	1000000
LibSort	Sorted	1.136354ms	3.957259ms	11.989826ms
	Nearly Sorted	1.93149ms	17.649961ms	38.1101ms
	Reversely Sorted	0.789966ms	2.276873ms	4.306323ms
	Random	3.857571ms	13.282841ms	165.289414ms

Part2: Quicksort Focused Study

Report the results of your focused study of Quicksort in the table below.

	Recursion Depth	Execution Time (ms)
Simple Random	328950	216.32555ms
Median of Three	329168	237.057208ms
with InsertionSort	23743	145.23091ms
Your Best Result (Extra Work)		

Discussion

Discuss the following points. This discussion is limited to the algorithms studied in class. So, library sort is excluded, because its internals are unknown.

1. For each input type (sorted, nearly sorted, reversely sorted, random), which algorithm has the best performance? Explain why?

   Sorted: **Insertion sort**

   The reason why this was the fastest algorithm is due to the fact that when you present sorted data to insertion sort it really just always breaks on the first call of the inner loop. Due to there never being a time were the data[i-1] is greater than the value at data[i]. Therefore, it will run that in constant time. Making the algorithm linear, the best case for insertion sort, in comparison to running at nlogn which is the best runtime for merge sort or quicksort.

   Nearly Sorted: **Insertion sort**

   Insertion sort is the fastest for a nearly sorted array as there are small number of comparisons done for a nearly sorted array. As since it is nearly sorted it will not be doing n amount of comparison. Instead, it will be doing less than that. Making it faster than merge sort as that will have to do n comparisons just like Quicksort will have to do when using the implementation of the pivot at the last index. Therefore, making insertion sort faster than the other two algorithms.

   Reversely Sorted: **Merge sort**

   In a reversely sorted input Merge sort is the fastest. Due to it running at a time complexity of nlogn. In comparison to insertion sort which will be ats worse case of n^2 as it will need to do the maximum number of comparisons every single iteration. While for Quicksort we are also at its worst case due to our implementation that we chose for quicksort which ends up being n^2 as well. Therefore, merge sort with its runtime of nlogn is faster than the other two.

   Random: **Quicksort / Merge sort**

   For random the fastest algorithm was Quicksort for the input size of 10000 due to the algorithm having a very low chance of the algorithm hitting its worst case. Due to the input seemingly working as a random pivot selection. Making the runtime nlogn. This changes as we run the other two test as with our implementation of quicksort it will start to get stackoverflow errors. Therefore, making it not comparable and leading merge sort to be faster on these sizes due to its runtime of nlogn and insertion sort running at n^2 due to the number of comparisons they both need to do in order to sort the array. As insertion sort will have the chance of having to check every element in the array every time.

2. How does the performance of each algorithm change when we change the input type? Explain why?

Insertion Sort:

When the input type is completely sorted the performance of insertion sort is at its best as it performance is only dictated on the size of the given data running at a time of n. With a nearly sorted array it will be a tad slower as it has to run its inner loop now, but it will still be close to linear as it will only do a small number of comparisons before it breaks and exits the loop making it the second fastest. Then it sees a drastic increase in runtime as the input type changes to be reversely sorted. This is due to it now handling its worst case which is n^2 due to you now having to do the maximum number of comparisons in order to properly sort the array. This runtime does get better when the input type is random as while it will still be n^2 runtime it will not be running at its absolute worst case. Due to the fact that it will not be doing the maximum number of comparisons due to data[i-1] not always being greater than data[i].

MergeSort:

Although all of these will run at the same time complexity, they all differ due to the number of comparisons that are needed to be done to sort the array. With a nearly sorted array taking the least then followed by reversely sorted, random and sorted in that order. The reasons the comparisons differ is due to the splitting of the array as we are always comparing the current element in the left array to the right array. Leading to a difference in runtime.

QuickSort:

Quicksort was at its fastest when the data was random. The reason being due to our implementation. As due to our implementation when data is random will lead to a small number of swaps due to data[j] being less than pivot. This only gets when you start sorting the data. With reversely sorted being the second fastest. As a result, that it will just change the element at the end with the starting value every single time. Then followed by a sorted array and lastly the nearly sorted array.

3. How does the performance of each algorithm change when we change the input size? Explain why?

Insertion Sort:

On all input types as the input size went up the runtime went up with it. As with more data coming into the algorithm the software had to do more comparisons the whole way up leading to an increase in runtime. The worse offense being for reversely sorted and randomly sorted arrays. Where the gap between them and the other two, sorted and nearly sorted, grew. Which focusing more on sorted and nearly sorted as the input size went up it begun completing faster on the nearly sorted data over the sorted data.

Merge Sort:

For both the 10000 and 100000 input size both data sets followed the same ranking of reversely sorted, nearly sorted, sorted, and lastly random. This changed for the last input size of 1000000 as by this point the number of computations for the random data overtook the amount needed for sorted the same way as nearly sorted worked for reversely sorted.

Quick Sort:

Quicksort was the most affected by the change in input as it only ran properly for the input size of 10000 while it crashed for all of the other data sizes through StackOverFlow Errors. The reason being is that there is a limited amount of space available on the stack which has all of the parameters and function calls of all of the methods in the program. So, when we got to any of the data sets over a size 10000 the amount of function calls needed along with storage of parameters, local variables and more we basically overflowed the built-in stack maximum. Causing the code not to work when we used the implementation picking the pivot all of the time at the last index which could lead to the absolute worst case of quicksort in terms of space due to the depth that the quicksort would have to go.

4. Do you have any other observations or insights?