Alif Rahi
Project 1 analysis
Cs323 section 14

Please include the following information in your document:

Processor: 2 GHz Quad-Core Intel Core i5

RAM: 16gb

Cache sizes/type: 1 terabyte

Operating System: Mac

Language(s): Java

Analysis of part 1:

Here are the average outputs I got running this program 1000 times for a 10,000 element array.

Time Taken for mergesort = 0.863 milliseconds
Number of swaps = 133616
Temporary arrays created = 9999

Time Taken for quicksort = 0.64 milliseconds
Number of swaps = 24977

Time Taken for HeapSort = 0.901 milliseconds
Number of swaps = 106775

Time Taken for insertionSort = 16.902 milliseconds
Number of swaps = 25014365

d) We can see that out of all these algorithms, quickSort had the fewest average swaps. This surprised me because I thought HeapSort would have the fewest swaps. QuickSort does a good job of only swapping when it needs to. I wasn't surprised by the insertionSort results because its an exponential algorithm of O(n^2) and it makes sense to have that many steps if you swap everytime iterating through a nested loop and 10,000 elements.

Alif Rahi
Project 1 analysis
Cs323 section 14

e) Based on this analysis, we can see that mergeSort created a helper array 9999 times. This explains why the space complexity is O(n). It creates n numbers of helper arrays. This is probably why quicksort keeps winning by a few milliseconds everytime you try to time the two algorithms together.

Although the runtimes for QuickSort, MergeSort and HeapSort are all supposed to average nlogn, I realized that some are actually quicker than others. The nlogn algorithms are mostly close with quicksort being the fastest and heapSort and mergeSort being edge to edge, however insertionSort is much slower. Especially when I changed the array from 10,000 elements to 100,000 elements. Sometimes my compiler didn't even show insertionSort. It takes a few minutes to 5 minutes to output. And though Quicksort is the fastest, it is also the most likely to crash. Depending on the order of my randomized array, quicksort crashes a lot. It made me think something was wrong with my code.

For one run only, here are my results for quicksort and mergesort on a sorted array.

- Time taken for QuickSort in sorted array = 29 milliseconds
- Time taken for MergeSort in sorted array = 1 milliseconds

Quicksort changed drastically from .7 milliseconds to 29 milliseconds. This is due to the worst case runtime of O(n^2) which only happens on sorted arrays or arrays of the same numbers. Another thing is that quicksort keeps blowing up. I had to run the program multiple times to work. MergeSort remained the same because it does not depend on comparisons. It creates an array each and every time the recursive method is called.

Alif Rahi
Project 1 analysis
Cs323 section 14

Analysis of part 2:

Array of size [16]:

Time Taken for **MergeSort** = 415.967 nanoseconds
Number of swaps = 64

Time Taken for **HeapSort** = 238.027 nanoseconds
Number of swaps = 26

Time Taken for **insertionSort** = 150.879 nanoseconds
Number of swaps = 48

Array of size [32]:

Time Taken for **MergeSort** = 814.445 nanoseconds
Number of swaps = 160

Time Taken for **HeapSort** = 527.161 nanoseconds
Number of swaps = 83

Time Taken for **insertionSort** = 289.083 nanoseconds
Number of swaps = 221

Array of size [64]:

Time Taken for **MergeSort** = 1483.879 nanoseconds
Number of swaps = 384

Time Taken for **HeapSort** = 1078.84 nanoseconds
Number of swaps = 225

Time Taken for **insertionSort** = 834.391 nanoseconds
Number of swaps = 997

Alif Rahi
Project 1 analysis
Cs323 section 14

Array of size [128]:

Time Taken for **MergeSort** = 3475.277 nanoseconds
Number of swaps = 896

Time Taken for **HeapSort** = 2403.005 nanoseconds
Number of swaps = 559

Time Taken for **insertionSort** = 3029.833 nanoseconds
Number of swaps = 3849

Array of size [256]:

Time Taken for **MergeSort** = 6757.137 nanoseconds
Number of swaps = 2048

Time Taken for **HeapSort** = 5771.558 nanoseconds
Number of swaps = 1404

Time Taken for **insertionSort** = 12806.197 nanoseconds
Number of swaps = 16480

d) For smaller arrays of 16-256, I averaged out the nanoseconds of runtime. InsertionSort was the fastest algorithm in arrays 16, 32 and 64. It was beating the other three sorting algorithms by hundreds of nanoseconds. Even at the array size of 128, insertionSort was still beating mergeSort with a runtime of 2,886 nanoseconds in comparison to its 3029.833 This proves that it is efficient to use insertionSort on smaller arrays. I wrote the code for quicksort but my program kept crashing due to the size of my stack perhaps. So I commented quicksort out. I assume that it's near the same runtime as heapsort and mergeSort anyway. InsertionSort is definitely the best for array sizes 64 and smaller.