Project 1

CS 253

Lance Ding, Joshua, David Choi

1.1

Total Inversions:
Data Set 1: 2324
Data Set 2: 2502835631
Data Set 3: 250160253054
Data Set 4: 0
Data Set 5: 499999500000
Data Set 6: 249742317055

1.2

MergeSort Runtimes:
Data Set 1: 0 ms
Data Set 2: 340 ms
Data Set 3: 4626 ms
Data Set 4: 3046 ms
Data Set 5: 3527 ms
Data Set 6: 4188 ms

HeapSort Runtimes:
Data Set 1: 0 ms
Data Set 2: 1125 ms
Data Set 3: 94773 ms
Data Set 4: 1335 ms
Data Set 5: 181763 ms
Data Set 6: 91833 ms

Insertion Sort Runtimes:
Data Set 1: 0 ms
Data Set 2: 314200 ms
Data Set 3: DNF - python was probably too slow. Extrapolation suggested that it may take more than 10h to run
Data Set 4: 112 ms
Data Set 5: 26340352 ms
Data Set 6: 12763148 ms

* Dataset 5 and 6 were run overnight

Insertion Sort Runtimes (Java) for reference:
Data Set 1: 0 ms
Data Set 2: 3113 ms
Data Set 3: 372366 ms
Data Set 4: 2 ms
Data Set 5: 181763 ms

Data Set 6: 91833 ms

1.3

F3 unilaterally yielded the worst results in terms of runtime. No matter what sort we used, this was the worst result for all sorts. However, this wasn't the array with the most inversions. It is likely a representation of an "average" or slightly "below average" case for all sorts. We immediately see the difference between the two O(nlogn) sorts (merge and heap) and the O(n^2) sort (insertion), as expected.

F4 was a sorted ascending array. Was by far the fastest of the four datasets of size 1,000,000. Since the data set was already in order, insertion sort did not need to make any changes to the array; the time complexity was O(n). For heapsort there is no heapify process as the dataset is already in minheap order. This means that the root of the heap can be immediately taken out and the successor is immediately found and also taken out.

F5 was a sorted descending array. Since mergesort is stable and relatively insensitive to the ordering of the elements, it performed the best. One of insertion sort's worst case scenarios appear here, since it actually has to stop at the last elements for every insertion, not allowing it to end any interior loops early. For heapsort, since the array was in the reverse of the heap configuration that we want, the heapify process likely took a very long time before the actual sort could be done.

F6 contained a lot of duplicates being an array of 1,000,000 elements with only 3 digit numbers. Since mergesort is the best out of the three at dealing with many duplicates at a large scale (insertion sort would be technically quicker at dealing with duplicate elements but given the size of the problem it is slower), it performed the best. Additionally, since the array is not sorted in a way that is averse to heapsort, it performs better than F5. Insertion sort, since it is no longer in its worst case scenario and is quite good at dealing with duplicate elements, performs better than F3 and F5 despite having the same problem size.

*Making comparisons with real runtime is very difficult as there are many different factors that contribute to differences in run time. The different processes happening in the background of the computer running these algorithms can lead to different runtimes.

1.4

Given the same data set and two algorithms with the same time complexity. It is completely possible for one to be much faster than the other. But why is this? Let's observe a case with both mergesort and quicksort (both algorithms with O(nlogn)) and data set 4 which is an already sorted data set. Even though both algorithms have the same average time complexity, in this special case mergesort has a time complexity of O(nlogn) and quicksort has a time complexity of O(n^2).

This is due to the nature in which these algorithms work. While both are divide and conquer algorithms, mergesort recursively partitions the array in half and merges them back together while sorting. The partitioning takes O(logn) time (as the recursion depth is logn) and the iteration as it merges takes O(n) time leading to a total time of O(nlogn). Quicksort on the other hand takes a pivot and swaps elements into place before the pivot is placed and the array is partitioned into two arrays. The worst case of quicksort is either an inorder array or an inverse order array. Where the pivot chosen is always the highest or always the lowest leading to a recursion depth of n, as the array is partitioned into n-1 and 1 size arrays each time, and an iteration time of n leading to a total time complexity of n^2.

We see that it is completely possible to have two vastly different run times with the same datasets and the same average time complexity. So why wouldn't we just use mergesort for everything over quicksort? This is due to memory allocation. Since mergesort requires an auxiliary array to merge into, it would take up more space than quicksort which only takes up one array of size n. If memory is an issue, quicksort would be preferred over mergesort. Each algorithm has some benefits over others that lead to a nuanced argument for the use case of each.