Tucker Dickson

Professor James Cremer

CS:1210

11/5/2021

Time Complexity Comparisons for Six Sorting Methods

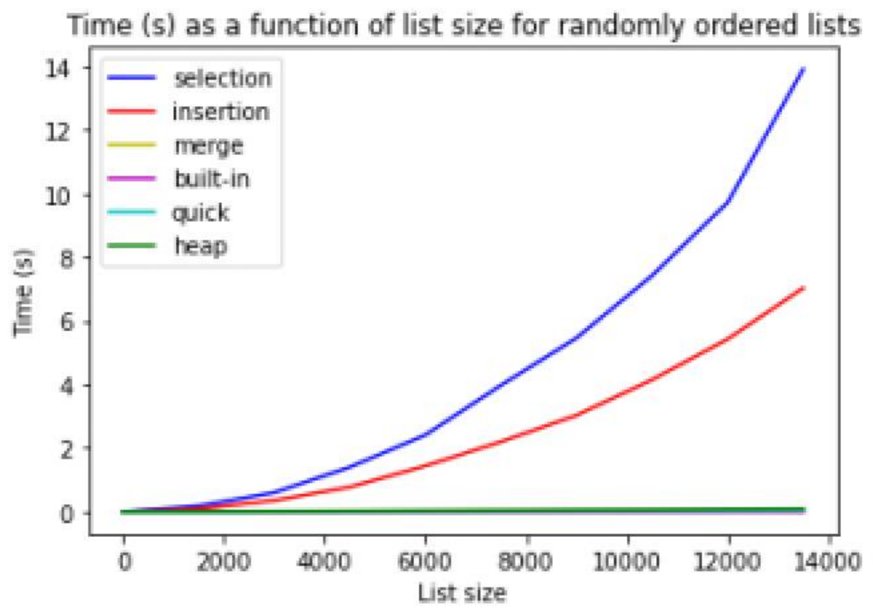
This report will attempt to give insight into the running times of six different sorting algorithms: namely selection sort, insertion sort, merge sort, python's built-in sort function, quick sort, and heap sort. Each of these sorting algorithms will be tested on a variety of sample data sets ranging in size and structure. Specifically, we will be considering data sets that are either randomly ordered, already sorted, or reverse-sorted to see how the runtimes of each of these algorithms are affected.

Part I. Randomly Ordered Data

The visual below depicts the running times for each of the six sorting methods listed above on data sets containing randomly ordered data. The size of these data sets ranges from 0 items to 15,000 items.

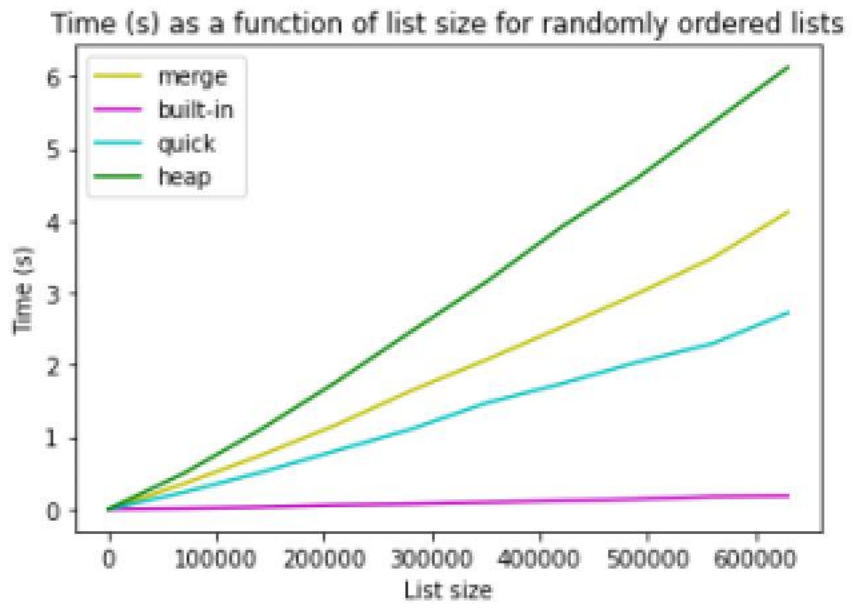
The first thing to notice about this graph is that the rate of change for both selection sort and insertion sort increases noticeably over time. Because we know that selection sort always follows n^2 complexity and insertion sort follow n^2 complexity on average, this behavior is expected.

The second thing to notice about this graph is that since the other four sorting methods are so fast compared to selection and insertion sort, they all overlap along the x-axis, and any differences between them are indistinguishable. For this reason, a second graph has been provided, which excludes selection and insertion sort.



The visual below depicts the running time of the four "fast" sorting algorithms for randomly ordered data: merge sort, python's built-in sort, quick sort, and heap sort. Excluding selection sort and insertion sort in this graph allowed for the use of much larger datasets (ranging in size from 0 to 700,000 items), further highlighting the differences in performance between algorithms.

One thing to notice about this graph is that all the algorithms display similar almost-linear trends. We know that merge sort, and heap sort always follow nlog(n) complexity, and that on average Tim sort and quick sort follow nlog(n) complexity. With all that in mind, the commonality between all the algorithms makes sense. This also explains the huge difference in performance between these four algorithms and selection and insertion sort.

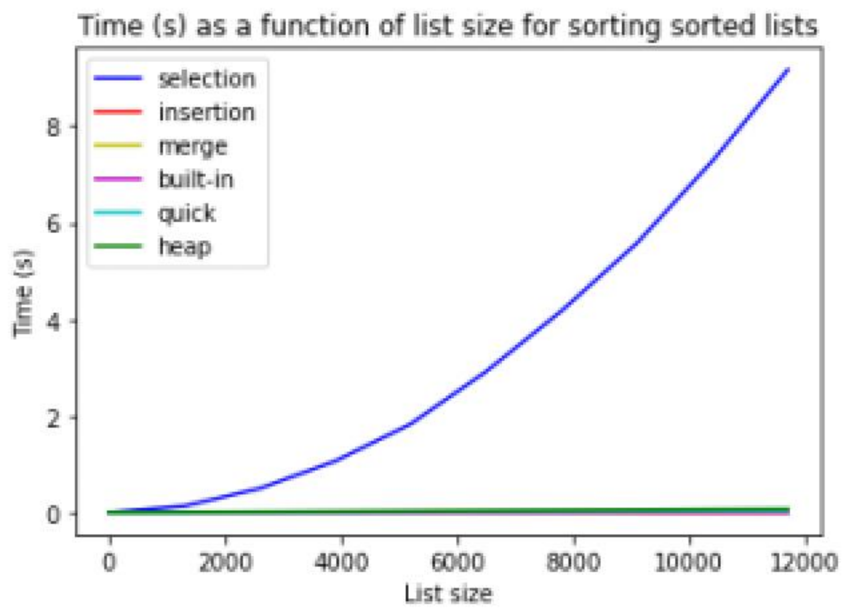


Part II. Sorted Data

The visual below depicts the time complexity for all six of the sorting algorithms when used on datasets containing data that is already sorted. The sizes of these data sets ranges from 0 items to 13,000 items.

The first thing to notice about this graph is that selection sort is far slower than the five other algorithms. Since selection sort always follows n^2 complexity, regardless of the ordering of the data, this behavior makes sense.

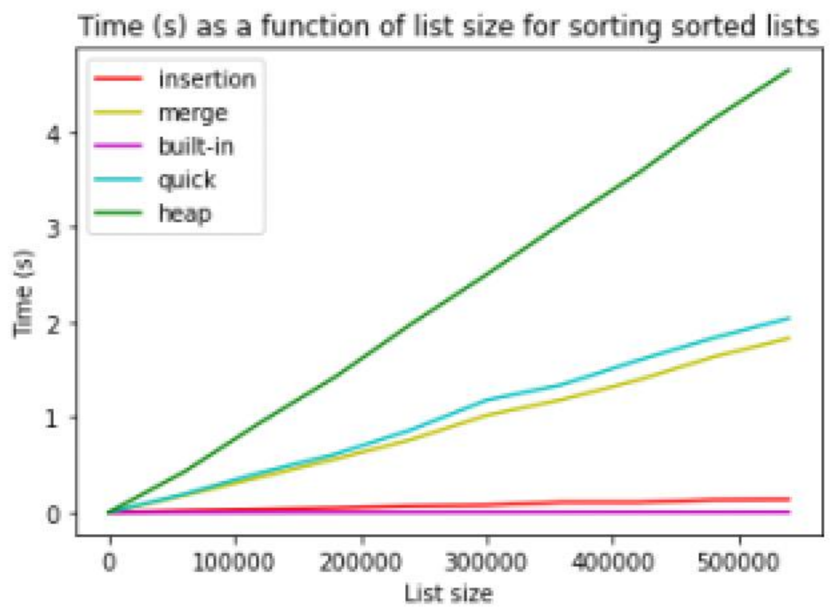
The second thing to notice about this graph is that the five sorting algorithms other than selection sort are all so much faster than selection sort that they're essentially indistinguishable on this graphic. For this reason, a second graphic has been included that leaves out selection sort.



The following graphic depicts the time complexity for the five “fast” algorithms on sorted data.

The first thing to notice about this graphic is that both insertion sort as well as python’s built-in sort both preform very well on sorted data. This behavior makes sense since we know that the best-case complexity for both insertion sort and Tim sort is n, and the best-case scenario is sorted data.

The second thing to notice about this graphic is that heap sort, quick sort, and merge sort all fit a seemingly linear trend; however, they don’t perform as well as insertion or Tim sort. Because merge sort and heap sort always follow nlog(n) complexity, and on average quick sort follows nlog(n) complexity, this behavior makes sense.

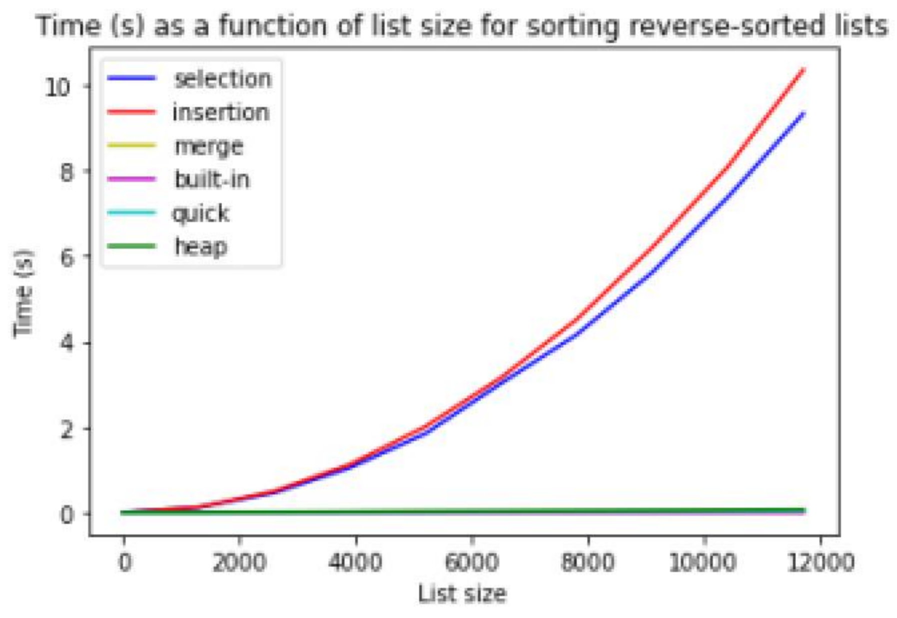


Part III. Reverse-Sorted Data

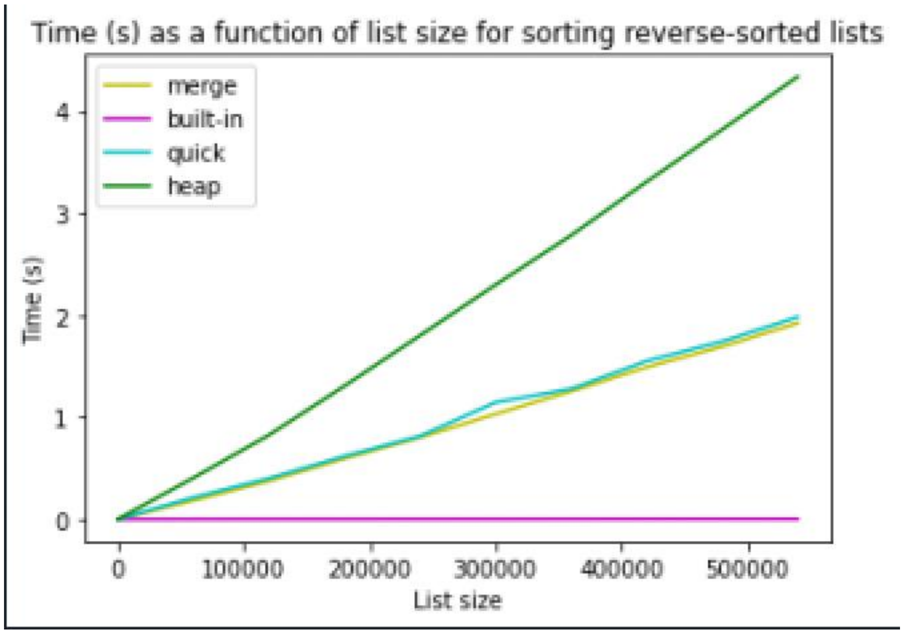
The following graphic depicts the time complexity for all six sorting algorithms on datasets containing reverse-ordered data.

The first thing to notice about this graph is that the rates of change for both insertion sort and selection sort increase over time. This makes sense because selection sort always follows n^2 complexity, and insertion sort follows n^2 complexity when sorting lists in the reverse order.

The second thing to notice about this graph is that again, the other four algorithms are so much faster than selection and insertion sort that they all overlap at the x-axis and can't be distinguished from one another. For this reason, another graph has been included which leaves out insertion sort and selection sort.



The following graphic depicts the time complexity for the four "fast" algorithms for reverse-sorted data. All four of these algorithms follow nlog(n) time complexity on average, so it makes sense that they are all so much faster than insertion sort and selection sort.


Time (s) as a function of list size for sorting reverse-sorted lists
merge
built-in
quick
heap
Time (s)
List size