Caijun Qin
3545-8153
03/12/2020

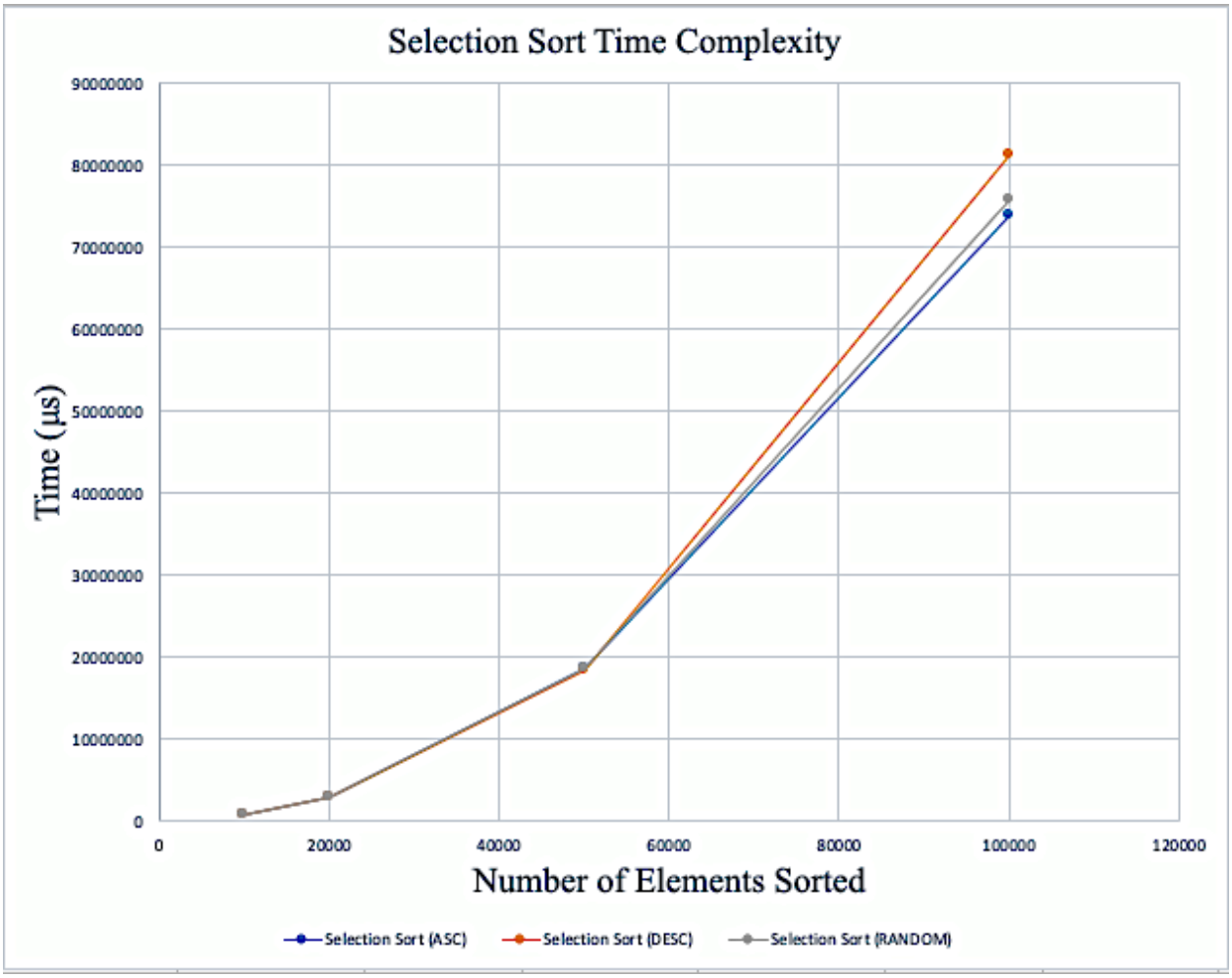
Sorting Analysis

Introduction and Performance

The big O time complexity of selection sort, heap sort, and counting sort are $O(n^2)$, $O(n \log n)$, and $O(n + k)$, respectively. For each sorting algorithm, the previously given time complexity is the same for best case, average case, and worst case. Altogether, the sorting algorithms ranked from slowest to fastest are selection sort, heap sort, and counting sort. To provide evidence for the time complexity of each sorting algorithm, vectors of size 10000, 20000, 50000, and 100000 integers were sorted using a function that implements each algorithm. Furthermore, each vector has three (3) versions: ascending, descending, and random. Each version is run and has its own line plotted under each sorting algorithm. In short, each sorting algorithm has its own graph that contains three line plots, one for each ordering. Each line plot tells the time taken to sort a particularly ordered vector of size 10000, 20000, 50000, 100000. The y-value (time taken) of each plotted point is actually the average of five (5) trials for the same algorithm, same ordering, and same vector size. Each vector is generated using a program called input_file_generator.cpp, which create a .txt file of integers representing a vector of specific ordering and size. The raw data for all the trials is in the Excel file.

Sorting Algorithms in Detail

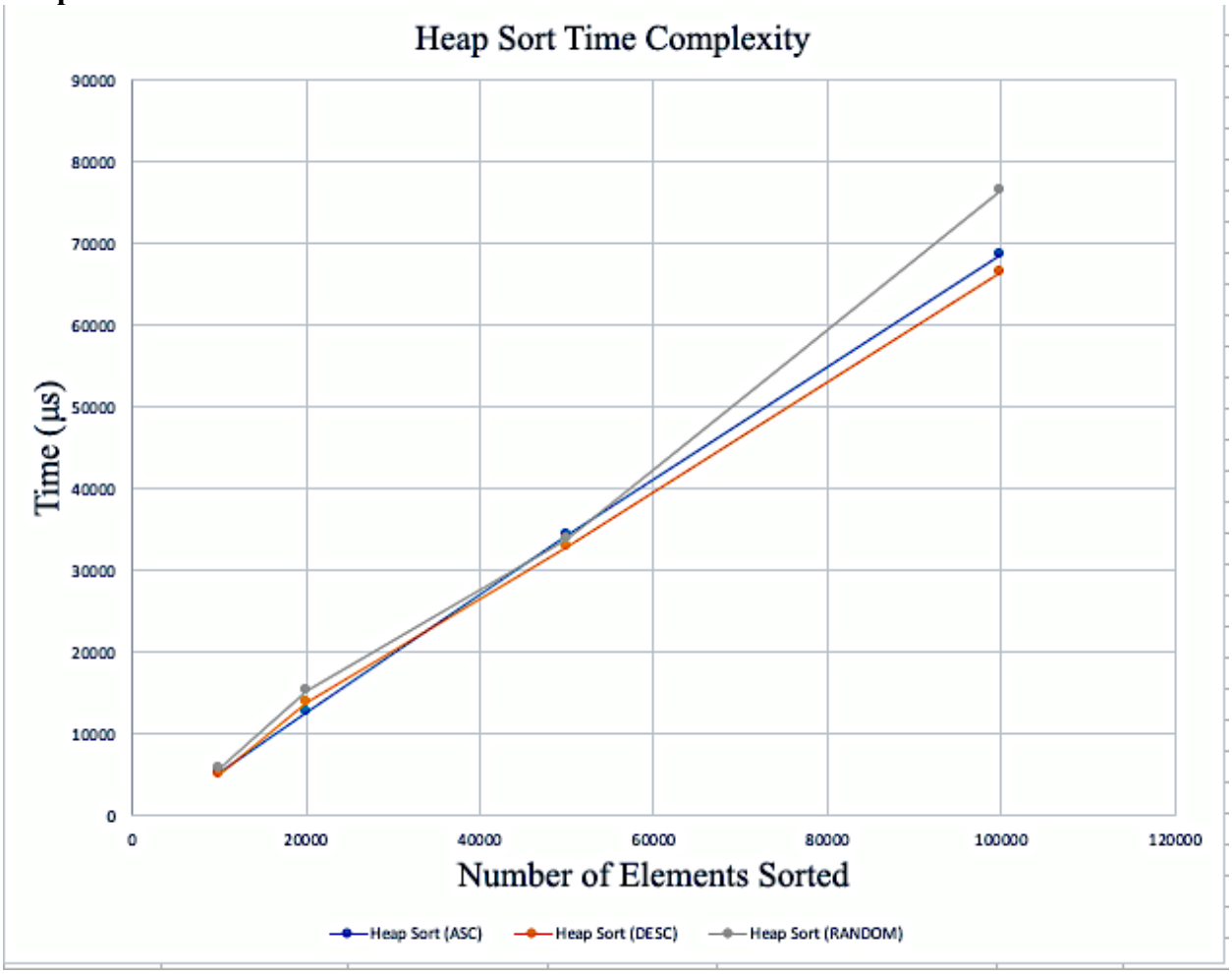
Selection Sort



Caijun Qin
3545-8153
03/12/2020

With a time complexity of $O(n^2)$, this sorting algorithm is quadratic in time complexity. As shown by the graph, all three (3) lines curve in a parabolic shape and stick close together. As x, the number of elements sorted, increases by a constant gap, the gap in y, the time taken to sort, increases at a growing pace. For example, as $x = 50000$ doubles to $x = 100000$, y approximately quadruples from 20 seconds to 80 seconds. This is quadratic behavior; $(2x)^2 = 4x^2$ indicates that doubling x quadruples y. The lines for ascending, descending, and random being packed closely means that the order does not alter the time complexity. The tiny gaps between each line when $x = 100000$ is due to the computer heating up after over a minute of sorting as well as scheduling other tasks in the background. On an interesting note, there is an unconventional way that can make the best case $O(n)$ time. For any sorting algorithm with a time complexity greater than $O(n)$, adding a preliminary scan across the vector to check that each element is less than or equal to the next element can confirm an already sorted vector. This is not part of selection sort by convention but is a time saving trick.

Heap Sort

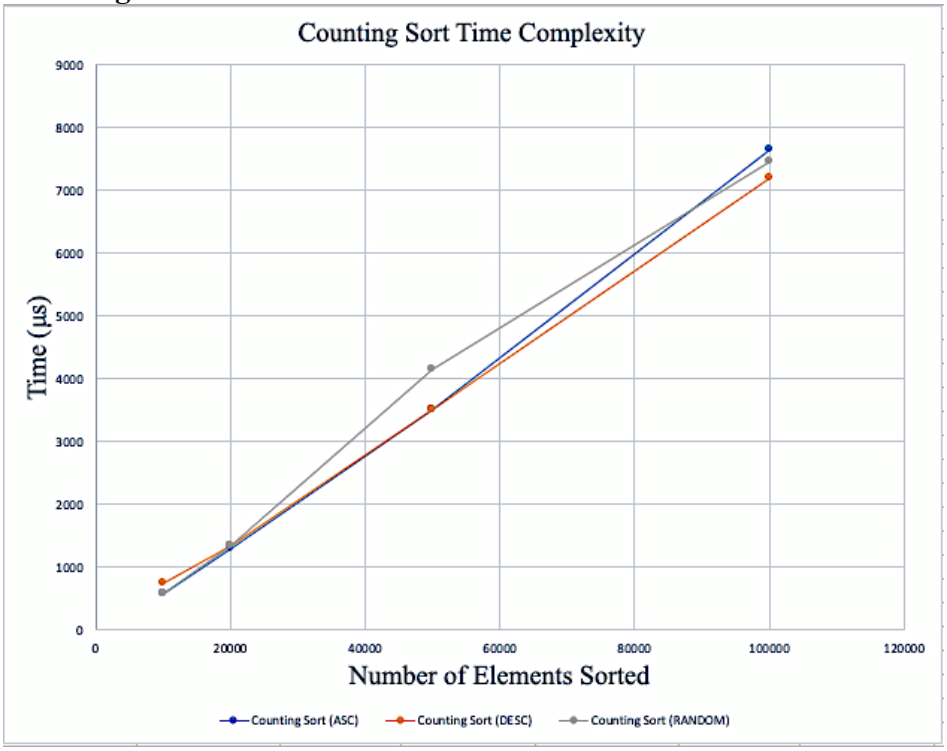


This sorting algorithm has a time complexity of $O(n \log n)$, also known as linearithmic. In the plot, the lines ever slightly curve up. This is most visible with the line for random order. As stated earlier,

Caijun Qin
3545-8153
03/12/2020

the order does not affect the time complexity in theory. However, the function $f(x) = x \log x$ will gradually curve above the function $f(x) = x$ when increasing x by large values. The lines for ascending and descending order seem to suggest slightly smaller time complexity than linearithmic, and this is due to environmental factors. For each sorting algorithm, running and capturing the duration is done first for ascending, then descending, and then random. Running ascending first had the advantage of the computer not as overheated as running the other two (2) orders. Descending order had the advantage of skipping the first step, which is building the heap from the initial input vector. In descending order, the maximum element is already first, and for any valid index i, any existing child at index $2i + 1$ or $2i + 2$ is already guaranteed to be smaller than the parent. However, descending order still does not escape $O(n \log n)$ time overall. Whenever the top of the max heap (front of vector) is swapped with the element at the next position in the sorted portion of the vector, the root (front of vector) changes value. To preserve a max heap, a heapify function is called at the root after each swap. This compares the root with both children, swaps with the largest child if that child is greater than the root, and recursively calls itself if a swap occurred until the unsorted portion of the vector is back to a max heap. This potentially needs the traversal of an entire path from root to a leaf, and such an action requires $O(\log n)$. Multiply this by n as heapify needs to be called each time a leaf replaces the previous root that has been sorted. Therefore, even starting with a max heap, descending order still requires $O(n \log n)$ time. Because the time complexity is slightly more than linear but under quadratic, it would take multiple plotted points including some with x-values greater than 100000 used in the above graph to more clearly show that sorting time for larger files gradually outgrows linear time complexity.

Counting Sort



The time complexity of this sort is $O(n + k)$, but amortized time complexity is $O(n)$ or linear time. This means that input file size is directly proportional to the sorting time. Assumption made about

amortized time complexity are in the later half of this paragraph. To summarize, counting sort sometimes requires large space complexity to trade for smaller time complexity. Essentially, a new vector, call it the *counter vector*, is created with size equal to the span of the input vector. The span, k, is the length of the gap between the maximum and minimum elements, inclusive (calculated as $k = \max v - \min v + 1, where\ v \equiv input\ vector$). Each element in the input vector has a corresponding index in the counting vector. For example, the minimum element has index 0, *minimum + 1* has index *1*, *minimum + i* has index *i*, and so on. Initialize the counting vector to all zeros (0). Iterate across the input vector, and at each element, increment the value in the counting vector at the corresponding index. This renders a counting vector that keeps the count of each element from the input vector in a sorted fashion, which takes $O(n)$ time. Clear the input vector but reserve the same capacity as its original size. Finally, for each valid index i from 0 to $k - 1$ in the counting vector, repeatedly append the value *minimum + i* to the input vector the same number of times as the count kept at index *i* in the counting vector. Inserting back into the input vector in sorted order takes $O(k)$ time. Altogether, the time complexity is $O(n + k)$. As shown in the graph, all three (3) plots appear quite straight, supporting linear time complexity regardless of order. Because the scale of time (y-axis) is much smaller than the other sorting algorithms, running the program is more prone to background interruptions from the computer scheduling other tasks. This is most evident in the small bump in the line for random order. On a final note, the time complexity is presented as amortized time complexity under certain conditions. The most important assumption is that each element in the input vector, when sorted, is close to both adjacent elements. In other words, this sorting algorithm wastes too much space if the input vector is sparse. When the span, k, is large compared to the number of elements, then the counting vector will allocate much memory while having most of its elements holding a zero (0). Any integer between the maximum and minimum elements but absent in the input vector corresponds to a zero (0) in the counting vector. As a preliminary measure, a ratio $R = \frac{unique(v).size()}{k} = \frac{number\ of\ unique\ elements\ in\ v}{\max v - \min v + 1}$ may be computed. If $R < 1$ and not close to 1, it is not advised to use counting sort. In general, if span is n^2 or larger, meaning $R \leq \frac{1}{n}$, there are faster alternatives to counting sort. This also means that if $k = cn \mid k < n^2\ and\ c\ is\ a\ constant$, meaning the span is a multiple of n but still less that n^2, then total time complexity is simplified to $O(n)$ linear time. This is the main requirement behind the amortized time complexity.