Time Complexity Analysis Report

Programming Assignment #3

Uladzimir Kasacheuski

4/13/16

Objectives

This assignment required 3 things. First we created lists of random integers from 1-20000, with various numbers of integers inside them : 100, 500, 1000, 2000, 5000, 8000, 10000. After creating these n sized lists of these random integers, we were then asked to run them through 4 sorting methods and to analyze the time complexity. The purpose of analyzing the time complexities was to gain a better understanding of how the implementation of various sorting algorithms affects the time complexities, and how this time complexity varies based on the input - if its sorted or not sorted.

Algorithms

Insertion Sort

Insertion sort works in a very simple manner. You transverse down the array one element at a time. For each element, you compare it to the ones behind it. If this element is bigger than the ones behind it, you’re done. If it is smaller, then you swap the positions and test the new element behind it. In this manner you make sure that the element before each element is smaller than itself, and vice versa. The time complexity of insertion sort is on average and at worst of O(n^2).

Quick Sort

Quicksort is a more complex and more interesting algorithm. It is based on divide and conquer. All of the sorting in this algorithm is done in the dividing stage of the algorithm. The principle is that you pick the last element of the array, then to divide the array into elements smaller than this element and bigger than this element. After they have been divided, you place this element in between the two new subarrays, and repeat the quicksort process on each subarray. In this method you sort each part of this array. The worst case of this algorithm is O(n^2), like insertion sort, however the average case ends up being O(nlogn)

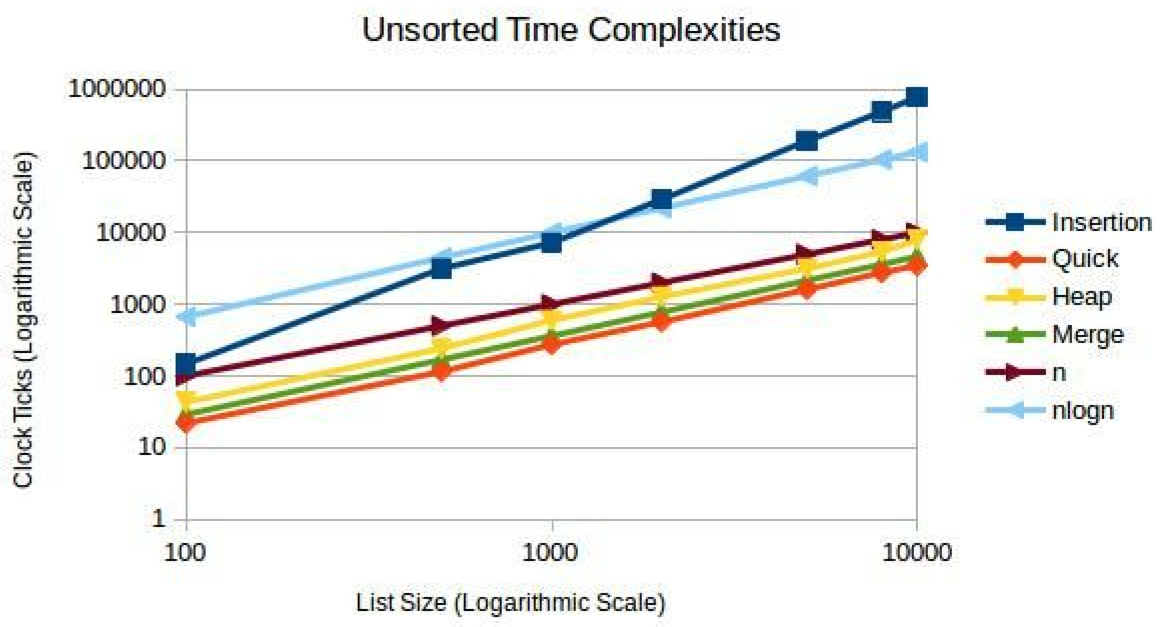
Heap Sort

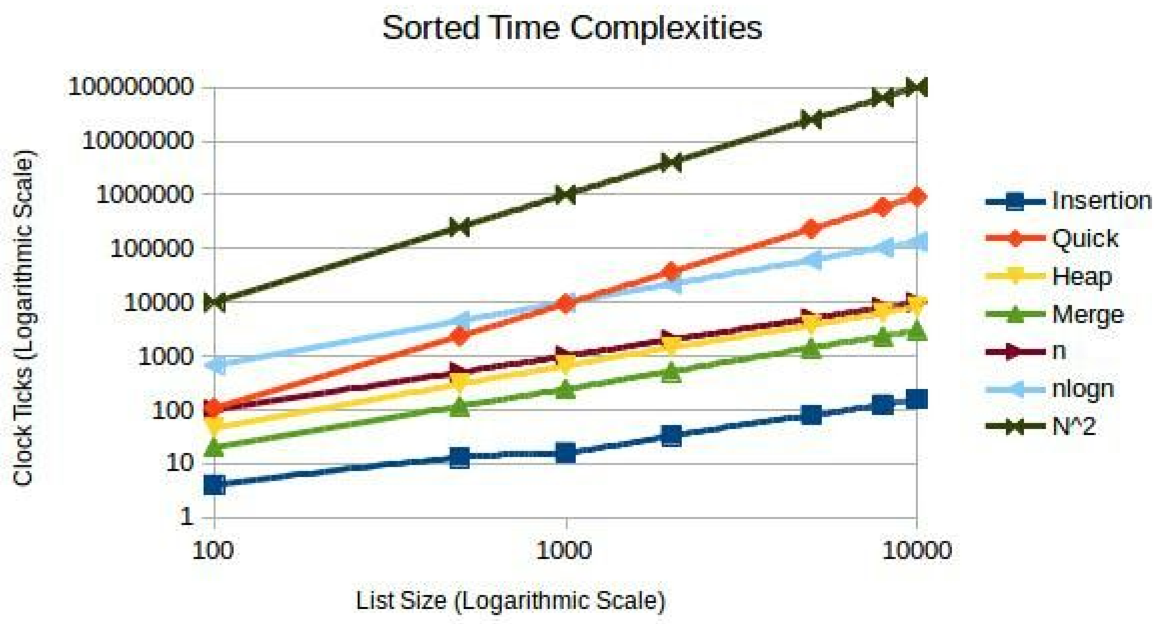
Heap sort is a very interesting algorithm. It employs the the fundamentals of a heap data structure in order to sort the array. A heap is based on the principle that every parent is either larger or smaller than its child elements. By setting up a heap where the parent is larger than every child, meaning the root is the largest character in the array, we are able to remove the root and place it as the last element of the sorted array repeatedly. In this method we reduce the size of the heap and create a sorted list. Implementations often require percolating down the heap. Average and worst case are both O(nlogn).

Merge Sort

Merge sort is another form of a divide and conquer algorithm. In this method, all of the analysis occurs at the conquering part, or more explicitly the merging part. You break down the array into subarrays (starting with individual elements), and then on merging two elements you create a new array and knowing that the next value you wish to put into the ordered array (the smallest value) is the first element in one of the subarrays. Thus by comparing the two first elements in the subarrays, the smaller one will be the first element. You then move the index up from the subarray you chose from and repeat the process. After you create a larger array from the combined subarrays, you repeat the process again with the new subarrays created (the large arrays). Average and worst case are both O(nlogn).

Graphs


Unsorted Time Complexities
Clock Ticks (Logarithmic Scale)
1000000
100000
10000
1000
100
10
1
100
1000
10000
List Size (Logarithmic Scale)
Insertion
Quick
Heap
Merge
n
nlogn


Sorted Time Complexities
Clock Ticks (Logarithmic Scale)
100000000
10000000
1000000
100000
10000
1000
100
10
1
100
1000
10000
List Size (Logarithmic Scale)
Insertion
Quick
Heap
Merge
n
nlogn
N^2

Comments

Insertion sort surprisingly performs better than expected, staying below O(n^2) for the unsorted and was the best at double checking the sorted data, well below even O(n). Quick sort was exceptionally efficient at sorting the completely random array, performing better than O(nlogn), however its worst case scenario of O(n^2) showed its colors with an input of a sorted array as it actually is the most complex algorithm in that case, even greater than O(nlogn). Heap sort displays itself as the most stable algorithm, aligning itself closely with O(n) with both the unsorted and the sorted input. Merge sort, equally as stable, displays itself as an even more efficient algorithm - in the graph staying below both O(n) and heap sort, and underperforming quicksort only partially with the unsorted array.

Conclusion

This programming project was an exceptionally useful experience. Implementing four different fundamental sorting algorithms and analysing their complexities as they take in random inputs and sorted inputs - seeing as they degrade or optimize with the different outputs. Based on this analysis, if you're double checking input, use insertion sort. If you know that the inputs are likely to be random and exceptionally large, quicksort would be a better fit. Yet, if you are not sure which it maybe, heap sort and merge sort would be the best way to go through with it. Their time complexities are stable regardless of the input put in it.