Sorting Algorithm Analysis

Justin Edwards

CSCI 36200

4/19/2018

Overview

The purpose of this report and the program created for said report is to analyze the run times of various sorting algorithms given different input sizes. The input data is also broken into two types, pre-sorted arrays and unsorted arrays. These arrays are created using a pseudo random number generator using the srand() function seeded with NULL. The user can select which method they would like to do an initial sorting. By allowing the user to choose, the sorted data set is generated and the user can verify that all algorithms implemented are working correctly. The verification of the sorted data and the generated data can be carried out by selecting the print sorted or print unsorted menu options. The user can then chose to run an analysis which will output the execution time of four common sorting algorithms. These algorithms are quick, insertion, merge, and heap sort. In the following report these algorithms will be explained and their time complexities explored using graphs of the data collected from running the program, Sort. The graphs were created by generating a .csv file on the runtime of the program. As each data set was sorted the run times were stored within a file called time.csv. This file can easily be imported into software such as excel or Matlab. For the purposes of this project all data was examined using Microsoft Excel.

Quick Sort

Quick sort is an algorithm that divides a data set into a series of subsets then sorts the subset and recurses back until the entire array is sorted. This can be thought of as tree split at the pivot that is initially the midway point in the dataset. The pivot is selected and all values less than the pivot are placed to the left and all values greater to the right. This give the advantage on run time if the array is already partial sorted. It has advantages over other sorting algorithms such as merge sort because it is an in-place sorting algorithm and does not require any additional memory allocation to sort the array. The average runtime is O(nlogn). The worst case is if a poor pivot is chosen and then the quicksort needs to be called n times. If this is the case the time complexity is O(n­2). Below is a graph of the run times given various data sets.

As you can see from the graphs the execution time decreases with a sorted input array as expected. The time increases nlog(n) as predicted by the algorithm analysis.

Insertion Sort

Insertion sort is another in place sorting algorithm. It works by successively comparing the values in a sub-array (contained within the array itself) with the next item in the array. If the item is in the wrong position the sub-array is shifted so that the element is in the correct position. This process repeats until the entire list is sorted. Below are the graphs of its runtime. The run time is O(n2) for an average scenario. The best case is if the array is already sorted in which case the runtime is O(n).

The above graphs show the expected behavior of this sorting function. The sorted list has a very low runtime and the execution time corresponds to the O(n). The unsorted case has a much higher runtime and clearly displays the poor time complexity of O(n2).

Heap Sort

Heap sort works by placing the data in a heap structure. This structure is in the form of a tree. The nodes and their children are then compared and swapped in order to place the tree in numerical order. The tree is then deconstructed and the nodes are place back into the array in numerical order. The average time complexity based on algorithm analysis is O(nLog(n)) time must be taken initially to place the data on the heap. This eats up some execution time. Below are the graphs of execution times. The algorithm behaves as expected below.

Merge Sort

Merge sort is a divide and conquer style algorithm. To accomplish the sort the algorithm divides the input array into halves. It then calls itself and is divided in half until there are just individual pieces. The halves are then sorted and merged back together into the final sorted array. The overall run time for all cases is O(nlog(n)). This can be a pitfall for when merge sort encounters already sorted lists. The it also takes time to merge the pieces this takes O(n). Below are graphs for sorted and unsorted data sets.

As you can see from the graphs the run time remains consistent between the unsorted and sorted data sets. This confirms the expected time complexity.

Conclusion and Commentary

As can be seen from the data above, there is a trade-off between in-place and memory consuming sorting algorithms. Generally, the in-place algorithms sacrifice time complexity for better space complexity. This was a key advantage back in the early days of computing when memory was limited and expensive. Now however, large memory sizes are commonplace so this trade-off is much less of an issue. Some of the algorithms do have some downtimes when it comes to already sorted data sets, such an example is merge sort. Picking the right algorithm for the right data set is key to achieving a high level of optimization for a program.

Notes

One side note, this code was compiled and run on a machine running Ubuntu Linux 16.04 LTS. Some classmates were having issues with collecting execution times on other distributions. This code was compiled and run on saber.science.iupui.edu. If there is trouble running the program please feel free to contact me at justedwa@iu.edu.