Kashif Ahmadi

ntropy@uga.edu

An Experiment of Sorts

Description

This experiment will test the efficiency of four sorting algorithms: insertionSort(), bubbleSort(), happyHourSort() (aka, “cocktail” sort) and quickSort(). The data that will be used to test these algorithms is a census file that delineates the population, poverty and crime statistics of every county in the United States. The experiment will run a program that loads the census file and parses the relevant data, and then this data will be used to test each algorithm by calculating the time it takes for each algorithm to perform its task in milliseconds. I will redirect output to a text file for each test run to record the results. After ten test runs I will calculate the average time of each sorting algorithm for each corresponding data field (county, population and crime percentage).

Computer Specifications for experiment

CPU:	Intel i5 3570k (stock CPU clock)
RAM:	2x CMX4GX3M2B1600C9 (stock clockspeed and timings; dual channel)
OS:	Ubuntu 12.10 (non-Unity, up-to-date as of 03/18/2013)
Compiler:	Java version "1.7.0_21"
	OpenJDK Runtime Environment (IcedTea7 2.3.9) (7u21-2.3.9-0ubuntu1~12.10.1)

Hypothesis

The sorting algorithm that will perform the best will be quickSort(), because the time complexity of quickSort() is typically O(n*log(n)) whereas the time complexities of its counterparts are O(n^2). The algorithm following quickSort() in efficiency would be insertionSort(), because, ostensibly, insertionSort() method of shifting entire subsets of sorted lists is better than shifting individual elements on each pass. Also, happyHourSOrt() should be more efficient than bubbleSort() since the former scans in both directions whereas the latter only scans in one direction.

<u>Results</u>

Note: numerical values denote time in milliseconds (ms).

<u>Sorting by County</u>

	Average	1	2	3	4	5	6	7	8	9	10
Insert	**33.7 ms**	34	33	35	33	33	33	33	34	33	36
Bubble	**101.4 ms**	102	102	103	101	102	99	101	100	102	102
Happy	**122.1 ms**	123	123	126	121	122	120	120	121	122	123
Quick	**21.6 ms**	22	21	23	21	22	21	22	21	21	22

<u>Sorting by Population</u>

	Average	1	2	3	4	5	6	7	8	9	10
Insert	**17.1 ms**	17	17	19	16	17	17	17	17	17	17
Bubble	**16.6 ms**	16	17	18	16	16	16	17	17	17	16
Happy	**21.0 ms**	21	21	21	21	21	21	21	21	21	21
Quick	**32.0 ms**	32	31	34	32	32	31	32	32	32	32

<u>Sorting by Crime Percentage</u>

	Average	1	2	3	4	5	6	7	8	9	10
Insert	**11.1 ms**	11	11	12	11	11	11	11	11	11	11
Bubble	**34.5 ms**	35	34	36	35	35	34	34	34	34	34
Happy	**41.8 ms**	40	40	50	40	40	40	40	40	40	48
Quick	**28.5 ms**	28	29	30	28	28	29	28	28	28	29

Observations

It is quite unexpected to me how well insertionSort() performed compared particularly to quickSort() - more than doubling its efficiency when sorting crime percentage, for example. What must have aided in the performance of insertionSort() is the fact that the population and crime percentage data were already arranged favorably in the census file. But quicksort() did better when sorting counties than insertionSort(), possibly because the county field has a greater range of values.

Another interesting thing to note is the fact that bubbleSort() - contrary to what I expected – performed significantly better than happyHourSort() in all three statistics. It may very well be that even though happyHourSort() does scan in both directions, the additional time taken to scan the opposite direction has detriments to efficiency.

All in all both of these bubble sorts did decidedly worse than insertionSort() and quickSort(). Although insertionSort() came out on top in this experiment, I believe the favorable arrangement of the data contributed significantly to this result. This is an anomaly of sorts.