CSCI 2720

Data Structures

Project 3

Sorting Algorithms

Fall 2016

**Team Members**

1. Name Andrew Durden UGA ID 81----------- CRN 25654

2. Name Carter Hart UGA ID 810158526 CRN 25654

3. Name Connor Kirk UGA ID 81----------- CRN 25654

1. **Introduction**

The objective of this Project was to empirically measure the complexity of sorting algorithms described by math in the field of Computer Science. By using these sorting algorithms on sufficiently large or small data sets, we can understand Big-O notation as a real phenomenon as well as observe the strengths and weaknesses of each algorithm with respect to time and space under various conditions.

The Experiment involves 6 sorting algorithms:

1. Bubble Sort
2. Insertion Sort
3. Selection Sort
4. Quick Sort
5. Merge Sort
6. Heap Sort

In the next section we will explore the theoretical pros and cons of these sorting algorithms. For the experiment, we run 3 different data sets (sorted, random, or reversed) on each algorithm and calculate the elapsed time with 2 time stamps. This is repeated for different sample sizes (10, 100, 500, 1000, 10000, 20000, 100000, and 200000) to observe how these different algorithms scale with input size.

1. **Theoretical Analysis of Sorting algorithms**

Sorting algorithms are described by their "Big-O" complexity, which is a measure of the rate of grown in complexity on input of size n. For example, a simple linear search of a list to find if an element is a member of the list is considered O(n). You might think that best case, we find that the element is the first member, which would only require 1 check, not n checks. However, O(n) is the upper bound of complexity, meaning we have to consider the worst case in order to understand an algorithms Big-O complexity.

Bubble Sort, Insertion Sort, and Selection Sort are all considered to be O(n2). This is because their sorting strategies do not take advantage of higher level data abstraction, but instead simply compare each element to every other element in order to figure out their order. For each element from 1 to n, we have to compare with each element 1 to n, and so their complexity is described as O(n2). However, this analysis is simplified because these algorithms still use small shortcuts to become more efficient under certain special cases. For example, Bubble Sort is O(n2) in its worst case, but best case (already sorted), no swaps have taken place and so fall under O(n), much better than Quick Sort, who's best case is O(n log n) complexity.

Quick Sort, Merge Sort, and Heap Sort all have average case O(n log n) complexity, which makes them more desirable in most cases. This efficiency comes at a cost, however, because they exploit data structures or logical abstraction to preform quickly, however this typically takes more memory. For example, Merge Sort involves splitting up an array into bits and then merging them together in order. However, in the best case scenario, this takes an addition n places in memory. In contrast, Heap Sort does not use more space, but the list must already be a heap data structure which comes with computational overhead.

1. **Experimental Setup**

In this section, you should provide a description of your experimental setup, which includes but is not limited to:

* 1. Machine specification
  2. How did you generate the test inputs? Why did you use different input files and What input sizes did you test?
  3. How do you measure time? Do you take an average of many execution? How many?
  4. Did you use extra memory space or other data structures other than an array? If so, explain when and why?

1. **Experimental Results**

In this section, for each algorithm, you should compare its theoretical performance to the actual performance (Execution time). Then you will compare them to one another in terms of actual performance and the number of comparisons done.

**4.1 O(n2) Sorting Algorithms**

Compare O(n2) sorting algorithms (Selection, insertion and Bubble) in terms of execution time then number of comparisons on the three different data sets. Discuss based on the results of your experiment.

Support your discussion with “*at least*” two plots:

1. A plot of Running time (y-axis) and number of elements (x-axis).
2. A plot of number of comparisons (y-axis) and number of elements (x-axis).

You might want to add extra notes on one of the sorting algorithms by inserting its specific details (see insertion chart on sheet2 in the excel file)

*Note: see the excel file plots.xlsx (The data in the file is unreal, just any numbers, SO, replace with your experiment results) you can create any chart by selecting the desired data.*

**4.2 O (n log n) Sorting Algorithms**

Compare O (n log n) sorting algorithms (Heap, merge and quick) in terms of execution time and number of comparisons. Show a table of your results and 2 plots similar to previous subsection 4.1.

1. **Concluding Remarks**

Provide a discussion of your results, which includes but is not limited to:

* + - 1. Compare the theoretical analysis with the experimental analysis. To what extent does the theoretical analysis agree with the experimental results? Attempt to understand and explain any inconsistencies you note.
      2. Comparing algorithms to each other which one is the fastest or the best? Does it depend on the order of the data to be sorted or number of elements? Which algorithms was most impressive with respect to amount of work done (number of comparisons)

Show a table and a plot of the running time for each algorithm on the three data sets, using different number of elements.

See table 1 below and the excel file.

Table 1: Running time vs. number of elements

(AVERAGE OF 3 TRIALS EACH, ROUNDED TO THE 1000th)

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **Number of elements** | | |  |  |  |  |  |
| **Sort Type** | **data type** | **10** | **100** | **500** | **1000** | **10000** | **20000** | **100000** | **200000** |
| Insertion | Inorder | 0.004 | 0.007 | 0.016 | 0.026 | 0.077 | 0.133 | 0.686 | 1.424 |
| Insertion | Random | 0.004 | 0.031 | 0.577 | 1.565 | 109.169 | 428.916 | 10664 | 44613.1 |
| Insertion | Reverse | 0.005 | 0.059 | 1.21 | 3.224 | 230.231 | 878.774 | 21472.9 | 90534.6 |
| Selection | Inorder | 0.004 | 0.028 | 0.569 | 1.587 | 107.406 | 419.753 | 11028.6 | 43261.9 |
| Selection | Random | 0.003 | 0.034 | 0.542 | 1.535 | 107.299 | 431.106 | 10829 | 43336.5 |
| Selection | Reverse | 0.004 | 0.031 | 0.405 | 1.469 | 108.719 | 481.84 | 11580.6 | 46098 |
| Merge | Inorder | 0.007 | 0.04 | 0.129 | 0.206 | 1.446 | 2.884 | 50.507 | 91.958 |
| Merge | Random | 0.008 | 0.047 | 0.132 | 0.255 | 1.964 | 4.079 | 51.049 | 111.821 |
| Merge | Reverse | 0.007 | 0.042 | 0.102 | 0.169 | 1.477 | 3.222 | 44.039 | 89.249 |
| Quick | Inorder | 0.004 | 0.009 | 0.029 | 0.038 | 0.374 | 0.699 | 4.009 | 8.413 |
| Quick | Random | 0.004 | 0.019 | 0.066 | 0.126 | 1.148 | 2.299 | 13.595 | 28.937 |
| Quick | Reverse | 0.004 | 0.009 | 0.02 | 0.035 | 0.382 | 0.733 | 4.442 | 8.267 |

// NOTES:

The number of comparisons insertion sort took on in order was always equal to n, meaning once it realized it was in order it's BIG-O became O(n), However in reverse insertion was terrible.

However, 2000002 is a huge number and it took a long time to get it to compute.

Also I thought Quick Sort was supposed to be bad in in order data sets...

**References:**

[1] Please insert references: books, papers, and web references

[2]