Efficiency Analysis: Various Sorting Algorithms with Real-World Application

Cadet Third Class Gavin Delphia

Cadet Third Class Brandon Daley

Cadet Third Class Alex Thomson

Math 340 (T4A, LtCol Werner)

04 May 2012

**INTRODUCTION**

The four sorting algorithms analyzed in this paper are the bubble sort, merge sort, insertion sort, and quick sort. Each sort manages data in a different manner while obtaining the same results, a sorted list of integers. In this analysis, there were four data sets used, each containing subsets of numbers in random, reversed, almost sorted, and sorted orders. Data Set 1 contained 1,000 integers, Data Set 2 contained 10,000 integers, Data Set 3 contained 100,000 integers, and Data Set 4 contained 1,000,000 integers.

Before sorting these data sets with a program created by the project team that implements every sorting method, the team was asked to determine possibilities for the Big-O of each sort. The Big-O of a sort is a generic interpretation of the runtime based on the number of elements to sort and the implementation of the algorithm. Table 1 contains the predictions for each sorting algorithm.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Bubble Sort** | **Merge Sort** | **Insertion Sort** | **Quick Sort** |
| **Big-O Prediction** | O(n2) | O(n log(n)) | O(n2) | O(n log(n)) |

**Table 1. Predictions for Big-O for Each Sort**

**JUSTIFICATION OF BIG-O DETERMINATIONS**

For the bubble sort, the algorithm contains two nested **for()** loops that each cycle through every number in the data set. If the outside loop cycles *n* times and the inside loop cycles *n* times, the Big-O for bubble sort should be O(n\*n) or O(n2). When examining the merge sort, every recursive call splits the data set into two different subsets; this part of the algorithm is O(log(n)). The merge sort also splits these sublists until each list contains only one element of the main data set; for this reason, that part of the merge sort algorithm is O(n). Therefore, the entire Big-O for merge sort should be O(n log(n)). The insertion sort also contains two nested loops that go through every element, a **for()** loop and a **while()** loop. Therefore, the Big-O for insertion sort should be the same as the bubble sort, O(n2). The last sort, quick sort, takes the first element in a list, makes two sublists of greater and lesser values to the element, then the algorithm recursively concatenates these results. Because it takes every first value and recurses until each sublist has one value, thereby going through every value and splitting them in half, the Big-O should be the same as merge sort, O(n log(n)).

**PERFORMANCE ANALYSIS**

Each algorithm ran through four data sets, each containing four lists of numbers. The time performance for Data Set 1, Data Set 2, Data Set 3, and Data Set 4 are located in Table 2, Table 3, Table 4, and Table 5, respectively.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Data Set 1 (1,000 integers)** | | | |
|  | **Random** | **Reversed** | **Almost Sorted** | **Sorted** |
| **Bubble Sort** | 21.0 ms | 6.0 ms | 2.0 ms | 3.0 ms |
| **Merge Sort** | 7.0 ms | 2.0 ms | 1.0 ms | 0.0 ms |
| **Insertion Sort** | 17.0 ms | 2.0 ms | 1.0 ms | 1.0 ms |
| **Quick Sort** | 52.0 ms | 39.0 ms | 32.0 ms | 31.0 ms |

**Table 2. Time Performance for Data Set 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Data Set 2 (10,000 integers)** | | | |
|  | **Random** | **Reversed** | **Almost Sorted** | **Sorted** |
| **Bubble Sort** | 643.0 ms | 691.0 ms | 276.0 ms | 280.0 ms |
| **Merge Sort** | 18.0 ms | 17.0 ms | 13.0 ms | 11.0 ms |
| **Insertion Sort** | 127.0 ms | 87.0 ms | 86.0 ms | 83.0 ms |
| **Quick Sort** | 970.0 ms | 956.0 ms | 935.0 ms | 942.0 ms |

**Table 3. Time Performance for Data Set 2**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Data Set 3 (100,000 integers)** | | | |
|  | **Random** | **Reversed** | **Almost Sorted** | **Sorted** |
| **Bubble Sort** | 105,724.0 ms | 86,629.0 ms | 35,388.0 ms | 34,870.0 ms |
| **Merge Sort** | 1248.0 ms | 609.0 ms | 952.0 ms | 609.0 ms |
| **Insertion Sort** | 9,069.0 ms | 9,235.0 ms | 9,685.0 ms | 9,152.0 ms |
| **Quick Sort** | N/A | N/A | N/A | N/A |

**Table 4. Time Performance for Data Set 3**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Data Set 4 (1,000,000 integers)** | | | |
|  | **Random** | **Reversed** | **Almost Sorted** | **Sorted** |
| **Bubble Sort** | N/A | N/A | N/A | N/A |
| **Merge Sort** | 223,519.0 ms | 79,857.0 ms | 172,733.0 ms | 76,661.0 ms |
| **Insertion Sort** | 1.22 \* 107 ms | N/A | N/A | N/A |
| **Quick Sort** | N/A | N/A | N/A | N/A |

**Table 5. Time Performance for Data Set 4**

N/A denotes data that the team could not collect for a couple of reasons. Runtime for Quick sort on Data Set 3 and Data Set for is not available because the computer used for sorting the data could not allocate enough memory to accommodate the large data sets. Quick sort is recursive, thereby using a stack to keep track of what it has already calculated; stack overflow occurred even when allocating all 4GB of RAM for the sort. The bubble sort and insertion sort runtime not collected for Data Set 4 attributes itself to time constraints. Because of the size of the data set, runtime was far too long, assuming O(n2) for the average calculation time. For these reasons, some runtime data is not available.

From the data that does exist, the project team interpreted many key points. When analyzing the best situation for each sort (the sublist of each data set that gave the fastest runtime on a consistent basis), the team found:

* Best Case Bubble Sort: Almost Sorted
* Best Case Merge Sort: Sorted
* Best Case Insertion Sort: Sorted
* Best Case Quick Sort: Sorted

Overall, each sorting algorithm performed best when the data was either sorted or almost sorted. Ironically, the bubble sort performed better when it sorted the ‘almost sorted’ sublists, as if it took its time when everything was sorted but was more efficient when it had work to do. On the opposite side of the spectrum, the project team achieved worst case for each algorithm in the following manner:

* Worst Case Bubble Sort: Random
* Worst Case Merge Sort: Random
* Worst Case Insertion Sort: Random
* Worst Case Quick Sort: Random

For the bubble sort, it is important to note that worst-time is also average time, O(n2). Only if it is fully sorted will it achieve O(n) because it will make no swaps and complete. For the quick sort, this worst-time scenario makes sense because it grabs the first element in the list and recurses. Some could believe worst-time would occur when the data is in reversed order, but the algorithm would only need to perform half of the work. Merge sort has worst-time performance with a random data set for the same reason as quick sort, and insertion works somewhat like a bubble sort; if the elements were in reversed order, only half of the data would require movement.

According to the project team’s predictions, the sorting algorithms roughly ran as planned in Big-O time. Obviously, some minor variations in runtime attribute themselves to computer performance due to managing other tasks as well. The time for bubble sort, on average, squared after every increase in data set size. Insertion sort followed the same pattern, and the merge and quick sorts relatively followed the predictions of O(n log(n)). These predictions held because the team analyzed how the algorithms would handle each element in the data sets.

**REAL-LIFE APPLICATIONS**

Many technologies today have built-in sorting functions that help an end-user when dealing with data sets. Microsoft Excel contains a sorting function for numbers, words, or other specifications. Java also has built in code that it uses to sort such things as arrays, ArrayLists, and other data sets. The focus of the project team was an online sorter[[1]](#footnote-1) with an unknown sorting algorithm. The team inputted the data sets, and only Data Set 4 failed due to its size of 1,000,000 values. Table 6 displays the best runtime[[2]](#footnote-2) (in seconds) out of ten tests for each of the subsets in Data Sets 1, 2, and 3.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Random** | **Reversed** | **Almost Sorted** | **Sorted** |
| **Data Set 1** | 1.06 s | 1.27 s | 0.99 s | 0.80 s |
| **Data Set 2** | 2.00 s | 2.25 s | 1.88 s | 2.01 s |
| **Data Set 3** | 8.08 s | 10.74 s | 15.44 s | 9.01 s |

**Table 6. Runtime for Web-Based Number Sorter**

When observing the runtime of the web-based number sorter, a distinct pattern emerges. As the data sets increase by a factor of ten (from 1,000 to 10,000 to 100,000 integers), the runtimes actually square. For example, using the Sorted sublist of each data set, the runtime went from 0.80 seconds to 2.01 seconds to 9.01 seconds. This increase in time is roughly proportional to the square of the previous runtime (by a factor of 2). Therefore, the team made a conclusion that the web-based number sorter was implementing either a bubble sort or an insertion sort. Because bubble and insertion sorts both have average, best, and worst case times of O(n2), O(n), and O(n2), respectively, the team was not able to determine exactly which of the two algorithms the sorter used. It is difficult to confirm the conclusion made because the source code for the web-based sorter is not accessible to end-users.

**CONCLUSIONS**

When observing sorting algorithms, it is important to determine their possible runtimes according to the number of elements being sorted. Big-O notation defines such times. The project team predicted Big-O runtimes for each of four sorting algorithms, bubble, merge, insertion, and quick sorts. After collecting data from each implemented algorithm, the project team affirmed the predictions of Big-O runtime. Using a real-world sorting application, the team tested the data sets against the web-based sorter. After collecting runtime data, the group concluded that the online sorter was implementing either the bubble sort or the insertion sort. The exact algorithm cannot be confirmed because bubble and insertion sorts both have the same best, worst and average-case Big-O times. Overall, the performance analysis of various sorting algorithms provided data to determine the best algorithms for certain situations and data sets.

**Appendix A: Bubble Sort Java Code**

public static void BubbleSort(ArrayList<Integer> data) {

for (int i = 1; i < data.size(); i++) {

for (int j = 1; j < data.size() - i; j++) {

if (data.get(j) > data.get(j + 1)) {

int temp = data.get(j);

data.set(j, data.get(j + 1));

data.set(j + 1, temp);

}

}

}

}

**Appendix B: Merge Sort Java Code**

public static void MergeSort(ArrayList<Integer> data) {

data = MergeSortHelp(data);

}

private static ArrayList<Integer> MergeSortHelp(ArrayList<Integer> data) {

int n = data.size();

if (n >= 2) {

int m = (int) Math.floor(n / 2);

ArrayList<Integer> l1 = new ArrayList();

ArrayList<Integer> l2 = new ArrayList();

for (int i = 0; i <m; i++) {

l1.add(data.get(i));

}

for (int i = m; i < n; i++) {

l2.add(data.get(i));

}

return Merge(MergeSortHelp(l1), MergeSortHelp(l2));

}

return data;

}

**Appendix B (Continued)**

private static ArrayList<Integer> Merge(ArrayList<Integer> l1, ArrayList<Integer> l2) {

ArrayList<Integer> L = new ArrayList();

while (!l1.isEmpty() && !l2.isEmpty()) {

int smallest = Math.min(l1.get(0), l2.get(0));

if (l1.get(0) == smallest) {

l1.remove(0);

L.add(smallest);

if (l1.isEmpty()) {

L.addAll(l2);

}

} else {

l2.remove(0);

L.add(smallest);

if (l2.isEmpty()) {

L.addAll(l1);

}

}

}

return L;

}

**Appendix C: Insertion Sort Java Code**

public static void InsertionSort(ArrayList<Integer> data) {

for (int j = 1; j < data.size(); j++) {

int i = 0;

while (data.get(j) > data.get(i)) {

i++;

}

int m = data.get(j);

for (int k = 0; k <= j - i - 1; k++) {

data.set(j - k, data.get(j - k - 1));

}

data.set(i, m);

}

}

**Appendix D: Quick Sort Java Code**

public static void QuickSort(ArrayList<Integer> data) {

data = QuickSortHelper(data);

}

private static ArrayList<Integer> QuickSortHelper(ArrayList<Integer> data) {

if (data.size() <= 1) {

return data;

}

int firstElement = data.get(0);

data.remove(0);

ArrayList<Integer> lessThan = new ArrayList<>();

ArrayList<Integer> greaterThan = new ArrayList<>();

for (int i = 0; i < data.size(); i++) {

if (data.get(i) <= firstElement) {

lessThan.add(data.get(i));

} else {

greaterThan.add(data.get(i));

}

}

return concatenate(QuickSortHelper(lessThan), firstElement, QuickSortHelper(greaterThan));

}

**Appendix D (Continued)**

private static ArrayList<Integer> concatenate(ArrayList<Integer> less, int pivot, ArrayList<Integer> greater) {

ArrayList<Integer> concatenated = new ArrayList<>();

for (int i : less) {

concatenated.add(i);

}

concatenated.add(pivot);

for (int i : greater) {

concatenated.add(i);

}

return concatenated;

}

Documentation:

On 03 May 2012, our group met with LtCol Werner to discuss possible errors in our code for Merge Sort. In the first loop, he noticed that we were not checking the right condition for whether the current array was a specific size, thereby not properly sorting our array. We made the adjustment and the sort worked correctly.

1. The online sorter was located at <http://endmemo.com/math/numsort.php> and accessed on 03 May 2012. [↑](#footnote-ref-1)
2. The team took the *best* runtime because dealing with internet applications holds the potential for server communication to take longer at times. [↑](#footnote-ref-2)