Comparing the Bubble-Sort and Merge-Sort Algorithms

Luke Hunsberger

April 10, 2013

1 Introduction

This paper addresses the problem of sorting a list of numbers. A sorting algorithm takes any list of numbers as its input and returns as its output a list containing the same numbers, but sorted into increasing order. Recently, two sorting algorithms have been presented in the literature: Bubble-Sort and Merge-Sort. This paper compares the performance of these two sorting algorithms on long lists of numbers.

1.1 Bubble-Sort

The Bubble-Sort algorithm works by passing through a list of numbers multiple times. In particular, if the list contains n elements, then Bubble-Sort will make at most $n - 1$ passes through the list. On each left-to-right pass through the list, it compares adjacent elements to see whether they are in the right order. If they are not in the right order, it swaps them; otherwise, it leaves them alone. After at most $n - 1$ passes through the list, the list is guaranteed to be sorted into increasing order.

For example, consider the list (4 3 1 2). On the first pass through the list, Bubble-Sort begins by looking at the first two items, 4 and 3. Since they are not in the right order, it swaps them, resulting in the list (3 4 1 2). Next, it compares the second and third items, 4 and 1. Since they are not in the right order, it swaps them, resulting in the list (3 1 4 2). Finally, it compares the third and fourth items. Again, since they are not in the right order, it swaps them, resulting in the list (3 1 2 4). Notice that after the first pass through the list, the largest number, 4, is in the correct position. This is an important characteristic of the Bubble-Sort algorithm.

The second pass through the list is illustrated below:

```
(3 1 2 4)  <-- Comparing 3 & 1:  Need to swap!
(1 3 2 4)  <-- Comparing 3 & 2:  Need to swap!
(1 2 3 4)
```

Notice that since 4 was guaranteed to be in its correct location after the first pass, there is no need to compare the last two elements during the second pass. This is another important characteristic of the Bubble-Sort algorithm. In particular, each successive pass through the list involves one fewer comparison.

The third time through this particular list, no swaps are needed; thus, the algorithm stops.

1.2 Merge-Sort

The Merge-Sort algorithm takes a "divide-and-conquer" approach. It begins by splitting the input list into two sub-lists, each having half of the items from the input list.[1] Next, it sorts each sub-list—using a recursive application of the very same Merge-Sort algorithm. Finally, it merges the two sorted sub-lists into a single sorted list.

For example, suppose the input list is `(4 2 8 3 6 7 1 4)`. First, the list is split into two sub-lists, `(4 2 8 3)` and `(6 7 1 4)`. Next, each sub-list is sorted, using a recursive application of the Merge-Sort algorithm, yielding: `(2 3 4 8)` and `(1 4 6 7)`. Finally, these two sorted lists are merged into a single sorted list, `(1 2 3 4 4 6 7 8)`.

Since each of the sub-lists, `(4 2 8 3)` and `(6 7 1 4)`, is sorted using a recursive application of the Merge-Sort algorithm, each sub-list is split into further sub-lists, which are then sorted, requiring further splitting and sorting. The recursive splitting of lists into sub-lists stops when the sub-lists have only one element. The reason is that a one-element list is necessarily sorted. Thus, the one-element list represents a *base case* for the recursive splitting process.

For the sample list, `(4 2 8 3 6 7 1 4)`, the recursive process of splitting and then merging is given in full detail below:

```
(4  2  8  3  6  7  1  4)   <--  split into two sub-lists
(4  2  8  3)(6  7  1  4)   <--  split each sub-list into two sub-lists
(4  2)(8  3)(6  7)(1  4)   <--  split each sub-list into two sub-lists
(4)(2)(8)(3)(6)(7)(1)(4)   <--  each sub-list has only one element:  SORTED!
------------------------
(2  4)(3  8)(6  7)(1  4)   <--  merge adjacent lists into sorted lists
(2  3  4  8)(1  4  6  7)   <--  merge adjacent lists into sorted lists
(1  2  3  4  4  6  7  8)   <--  merge adjacent lists into sorted lists
```

Notice that the original list in line 1 is split into two sub-lists in line 2. Those two sub-lists are recursively sorted in lines 3 through 5. Finally, those two sorted sub-lists are merged into a single sorted list.

[1] If the input list has an odd number of elements, then one of the sub-lists will have an extra element.

2 Empirical Evaluation

To compare the performance of the Bubble-Sort and Merge-Sort algorithms, each was implemented in Scheme using the DrScheme software. The Bubble-Sort algorithm was implemented by a function called `bubble-sort`. It uses the `bubble-one-pass` function to carry out a single pass of the algorithm through a list, and the `bubble-multi-pass` function to carry out all the needed passes through the list. The Merge-Sort algorithm was implemented by a function called `merge-sort`. The `merge-sort` function uses two helper functions, `splitty` and `merger`, that carry out the splitting and merging processes.

In addition, a testing function was implemented to generate random lists of numbers and apply each sorting algorithm to those lists. The testing function also reports the time taken by each algorithm to sort the list. Crucially, each sorting algorithm is applied to the same list.

The following Interactions from DrScheme show some typical results from comparing the Bubble-Sort and Merge-Sort algorithms. Times are given in milliseconds. For comparing performance, the most relevant times are those shown as `cpu time`.

```
> (sorting-test 1000 (list merge-sort bubble-sort))
SORTING FUNC: #<procedure:merge-sort>
cpu time: 16 real time: 15 gc time: 0

SORTING FUNC: #<procedure:bubble-sort>
cpu time: 532 real time: 534 gc time: 0

> (sorting-test 2000 (list merge-sort bubble-sort))
SORTING FUNC: #<procedure:merge-sort>
cpu time: 28 real time: 30 gc time: 0

SORTING FUNC: #<procedure:bubble-sort>
cpu time: 2092 real time: 2092 gc time: 32

> (sorting-test 4000 (list merge-sort bubble-sort))
SORTING FUNC: #<procedure:merge-sort>
cpu time: 64 real time: 65 gc time: 0

SORTING FUNC: #<procedure:bubble-sort>
cpu time: 8453 real time: 8575 gc time: 44
```

These results demonstrate that the Merge-Sort algorithm runs *much* more quickly than the Bubble-Sort algorithm, especially on longer lists. For example, the Merge-Sort algorithm takes only 65 milliseconds to sort a list containing 4000 numbers, while the Bubble-Sort algorithm takes over 8 seconds.

3 Conclusions

This paper presented an empirical evaluation of the Bubble-Sort and Merge-Sort algorithms. The data clearly demonstrate the superiority of the Merge-Sort algorithm, in terms of run-time, especially on lists containing many thousands of numbers.