Sorting Algorithms and Run-Time Complexity

Leanne R. Hinrichs

May 2015

Abstract

In combinatorics, sometimes simple questions require involved answers. For instance, we often want to compare multiple algorithms engineered to perform the same task to determine which is functioning most efficiently. Here, we introduce the bubble sort and merge sort algorithms for arranging objects in a row, and discuss the run-time complexity of both.

1 Introduction

Given a list of six integers, it would hardly be a challenge to arrange the values from smallest to largest. Some would begin by selecting the largest integer from the list, correctly positioning it at the rightmost position, and continuing to place the second and third largest digits and so on to the left of the largest until the integers are fully sorted. Others might perform the task in a slightly different way, perhaps making pairwise comparisons of adjacent integers, but it's safe to say that the time required to complete the sorting would be relatively analogous no matter the method chosen.

If we consider, instead, a list of sixty thousand distinct integers requiring sorting, the strategy would have to be much more precise. For instance, the previous technique of finding the largest integer and moving it to the end of the list would prove ridiculously laborious. Fortunately, computers are of great assistance on such tasks. Programming algorithms for seemingly simple tasks such as sorting is common practice, but the efficiency of the design must be considered. If we implement one sorting algorithm that must make many comparisons to exhaustively sort the list, when a different algorithm could have accomplished the task in fewer comparisons, time and computing capacity of the machine are lost.

Here, we introduce two sorting algorithms and discuss the process of each. Pseudocode is given for each method, and run-time complexity is examined. We consider best, average, and worst case scenarios for each algorithm.

2 Bubble Sort Algorithm

Bubble Sort functions by creating a sequence of pairwise comparisons of adjacent elements from an array. Beginning with the leftmost two integers from an array of size n of distinct integers, the algorithm compares the two numbers. If the left entry is larger than the right entry, the integers swap positions. If the left entry is smaller than the right entry, the two remain in position, and the algorithm continues. Next, Bubble Sort compares the (possibly just changed) second entry and the third entry from the array and makes the comparison step again. Once all adjacent elements have been compared pairwise and necessary swaps have been completed, a full pass of the algorithm is complete. In general, the algorithm will require a maximum of $n-1$ passes to entirely sort the array. We discuss possible improvements of the algorithm using pseudocode after an example.

2.1 Example of Bubble Sort

We first examine a single pass of the Bubble Sort algorithm on an array of size 5.

4	**3**	1	6	2

Comparing the leftmost digits, the 4 is larger than the 3, so the two will swap position.

3	**4**	**1**	6	2

In the second pairwise comparison, the 4 is larger than the 1, so the two will again swap position.

3	1	**4**	**6**	2

In the next comparison, the 4 is less than the 6, so the two remain in position, and the algorithm continues. This is still counted analogous to a “swap” by the Bubble Sort algorithm, even though no action takes place since a comparison must still be made.

3	1	4	**6**	**2**

In the final pairwise comparison of this pass, the 6 is larger than the 2, so the two swap position.

3	1	4	2	6

This displays the final state of our algorithm after the first pass is complete. Clearly, however, the list is not entirely sorted, so the algorithm must continue to make subsequent passes.

Note, however, that the largest digit in the array (6) has assumed the correct, rightmost position. In general, we can guarantee that the largest digit in any array of unique positive integers will be placed correctly after the first pass since it wins all pairwise comparisons. In fact, we are guaranteed that we secure one more correct position after each pass of the algorithm; the second largest entry will be in correct position after the second pass of Bubble Sort, and so forth. This idea will be important in making improvements geared toward efficiency of the skeleton Bubble Sort pseudocode [4].

2.2 Bubble Sort Pseudocode

The most simple form for a single pass of the Bubble Sort algorithm is shown below:

```
for i from 0 to n-2

    if (A[i] > A[i + 1])

        swap(A[i],A[i+1])
```

The algorithm simply compares each adjacent pair of digits as the swaps are taking place. Note that we only run i from 0 to n-2 because the final digit in the array is not compared with anything beyond.

As discussed previously, a single pass does not typically sort the algorithm entirely and successive passes are necessary. We introduce an improved version of the Bubble Sort full algorithm and then discuss the changes made.

```
for k from 0 to n - 1

    set fullsort = 0

    for i from 0 to n - k - 1

        if (A[i] > A[i + 1])

            swap(A[i],A[i+1])

            fullsort = 1

    if fullsort == 0, break
```

The first improvement is that the variable “fullsort” is initialized. In the simple version of the Bubble Sort code, we rely solely on the fact that we are guaranteed one more correctly sorted digit with each pass as previously discussed. This is an opportunity for great efficiency improvement. If we are able to recognize that the array is entirely sorted after, say, the second pass, it is unproductive to continue making the full $n-1$ passes. Using the variable "fullsort" we can recognize if a full pass is completed without any swaps taking place. If this is the case, the list is entirely sorted and we fall out of the algorithm.

The other change we introduce is the variable k, used to track subsequent passes, and this improvement actually refines what we do inside of a single pass. Since we know that after one pass, the largest integer is in correct rightmost position, its inefficient to compare anything to this digit on the second pass; it is already known to be larger. Thus, the $n-k-1$ portion of the i counting variable enables the pass to complete before making these repetitive comparisons for the portion of the array we already recognize to be sorted.

2.3 Bubble Sort Run Time

When discussing run-time, the concept of "Big O" is critical. Big O, also referred to as Landau's behavior, is used to described a rate of growth or behavior of a function. With many implications in computer science, we are able to strip coefficients and lower order terms to view the highest order term of a function.

For example,

$$f(n) = 3n^2 + 9n + 2 \implies f = \mathcal{O}(n^2)$$

From this simplified rate of growth, we are more easily able to compare functions side by side to determine traits of efficiency. For instance, if we were comparing two sorting algorithms, one polynomial degree three and another degree two, the algorithm with run-time $\mathcal{O}(n^2)$ would be more efficient for large arrays. An algorithm running n^3 is better than $1000n^2$ for small n, but eventually as n increases $1000n^2$ is better.

Considering Bubble Sort in particular,

$$\begin{aligned}\sum_{k=0}^{n-1} n-k-1 &= (n-1)+(n-2)+\ldots+0 \\ &= \frac{n(n-1)}{2} \\ &= \mathcal{O}(n^2)\end{aligned}$$

Algorithms whose behavior is polynomial are actually considered quite inefficient. So, while Bubble Sort is simple and a nice beginning example, other sorting algorithms are much more efficient in general. Bubble Sort does, however, perform quite well on certain arrays. Let's consider Bubble Sort for specific arrangements.

2.4 Bubble Sort Case Scenarios

2.4.1 Best Case

The best case scenario is somewhat trivial. If the array is already sorted before we begin, all the algorithm must do is verify this to be the case. As such, we compare each of the n values to only one other entry, and stop one short so we do not compare the final entry to anything beyond. The run-time for this configuration is shown below.

$$\begin{aligned} n-1 &\leq n \\ &\implies \mathcal{O}(n) \end{aligned}$$

Note, this uses the "`fullsort`" variable from (2.2).

2.4.2 Worst Case

The worst case scenario for the Bubble Sort algorithm is that the list is written in exactly reverse order. As a by-product of the second largest digit being compared only to the largest digit in the first pass of the algorithm, it is placed in the rightmost position after the completion of the pass. On the second pass, the third largest number is positioned in the rightmost place, so all of the possible $n-1$ passes of the algorithm must be executed before completion. In this case, we must compare every digit to every other digit, instead of comparing every digit to only one other digit like in the best case. For this configuration,

$$\begin{aligned} \binom{n}{2} &= \frac{n!}{2!(n-2)!} \\ &= \frac{n(n-1)}{2} \\ &= \frac{n^2-n}{2} \\ &\implies \mathcal{O}(n^2) \end{aligned}$$

Let's consider another sorting algorithm used to accomplish the same task in a different fashion, and compare the efficiency.

3 Merge Sort Algorithm

Merge Sort is a recursively-defined algorithm which continues to break an array down into smaller and smaller portions and then sorts those smaller bits before reassembling the sorted array. The number of comparisons required in the reassembly process is less than those needed in Bubble Sort, which increases efficiency. Let's examine an example.

3.1 Example of Merge Sort

$$n = 5$$

$$\begin{array}{|c|c|c|c|c|}\hline 4&3&1&6&2\\\hline\end{array}$$

Beginning with this array of size 5, we break into as evenly sized pieces as possible. In this example, we always go "left heavy," so if the array is of odd size, the additional entry will fall on the left side.

$$\begin{array}{|c|c|c|}\hline 4&3&1\\\hline\end{array} \quad \begin{array}{|c|c|}\hline 6&2\\\hline\end{array}$$

Again, we break these smaller arrays into two as evenly as possible.

$$\begin{array}{|c|c|}\hline 4&3\\\hline\end{array} \quad \begin{array}{|c|}\hline 1\\\hline\end{array} \quad \begin{array}{|c|c|}\hline 6&2\\\hline\end{array}$$

Continuing the process yet another time, we arrive at the base case.

$$\boxed{4} \quad \boxed{3} \quad \boxed{1} \quad \boxed{6} \quad \boxed{2}$$

This array of singletons cannot be broken down further, so we begin the process of reassembly. Remembering that the 4 and the 3 entries were grouped together at the previous step, we make this comparison first and swap that positioning, then consider the 6 and the 2 on the rightmost part of the array. The 1 entry does not get compared to anything in this step.

$$\begin{array}{|c|c|}\hline \mathbf{3}&\mathbf{4}\\\hline\end{array} \quad \begin{array}{|c|}\hline 1\\\hline\end{array} \quad \begin{array}{|c|c|}\hline \mathbf{2}&\mathbf{6}\\\hline\end{array}$$

Now, since the 1 was initially part of the left array, we will recombine it into that portion of the array. This is the portion that creates the opportunity for efficiency beyond that of the Bubble Sort algorithm. Since the 3 and 4 in the array are already known to be sorted, once we recognize that the 1 is less than the 3, no further comparisons are necessary.

$$\begin{array}{|c|c|c|}\hline \mathbf{1}&\mathbf{3}&\mathbf{4}\\\hline\end{array} \quad \begin{array}{|c|c|}\hline \mathbf{2}&\mathbf{6}\\\hline\end{array}$$

The same idea as the last step is now abstracted into this final merge. Now we compare the leftmost element of the left array, 1, to the leftmost element of the right array, the 2. Once we see that 1 is less than 2, we can definitively say that 1 is the smallest entry in the final array. Now we compare the 2 to the second from left position in the left array. Since 3 is greater than 2, place 2 to the right of 1 (the first entry) - no further comparison of this entry is needed. Thus, we compared the 1 only to the 2 in this step, and the 2 only to the 3, so every comparison guarantees that we discover final placement for one element. Due to this we find one pass with a maximum of $n - 1$ comparisons is required to reassemble the list. Since we are able to accomplish sorting in only one pass of the Merge Sort algorithm, we expect that run-time will be an improvement to Bubble Sort. To segue into that discussion, we first introduce Merge Sort Pseudocode.

3.2 Merge Sort Pseudocode

Let's examine just one merge process to understand how the algorithm functions in pseudocode. So, let's call the "parent" array from which the two partial lists are derived A, and the two partial lists L and R for right and left, respectively [3].

```
Merge(L,R,A)
    nL = length(L)
    nR = length(R)
    i = j = k = 0
    while (i < nL &&  j < nr)
        if (L[i] <= R[j])
            A[k] = L[i]
            k++
            i++
        else
            A[k] = R[j]
            k++
            j++
    while (i < nL)
        A[k] = L[i]
        i++
        k++
    while (j < nR)
        A[k] = R[j]
        j++
        k++
```

The first while loop is the typical execution of the algorithm. Moving through the partial lists, we compare elements to place them into the larger "parent" array. The two subsequent while loops only come into play in the case that the original while loops becomes false, that is $i \geq nL$ or $j \geq nR$. This situation occurs when one partial array is exhausted prematurely. For instance, if we were comparing an already sorted list, the algorithm would place all of the elements of L, the left list correctly before placing any elements of the right list. The while loops just say to place all the "leftover" elements of the array that is not exhausted in order since they are known to be sorted. Again, this code only describes one reassembly, called "Merge"; the entire Merge Sort algorithm requires the development of the recursive process. Let's now view the recursive portion which will call on the "Merge" process we just built.

```
MergeSort(A)
    n = length(A)
    if (n < 2)
        return
    mid = n/2
    left = array of left half
    right = array of right half
    for i = 0 to mid - 1
        left[i] = A[i]
    for i = mid to n - 1
        right[i - mid] = A[i]
    MergeSort(left)
    MergeSort(right)
    Merge(left,right,A)
```

Let's assume for the sake of simplicity that we have an array with the order of a power of 2. That way when we are continuing to do splits, the breaks will remain even throughout. In the case that it's not, we would introduce a ceiling function. In practice, however, when considering the large lists where this algorithm would be necessary, the number of extra comparisons needed would be all but negligible and would not give information about the fundamental behavior of the algorithm. The first condition of this algorithm is checking to see if we have hit the base case where we have a singleton entry. If $n \geq 2$ we have work to do, so the next few lines of code are introduced to split the "parent" array into right and left components. Once this has been completed, we recursively call on Merge Sort on the two smaller lists. Then, finally, we call upon the Merge method that we previously built to reassemble into a sorted "parent" algorithm.

3.3 Merge Sort Run Time

Let $M(n)$ be the number of comparisons required to sort a list of size n. Then $M(1) = 0$ and $M(2) = 1$ since a singleton is trivially sorted, and sorting an array of two entries only requires one comparison.

If we consider the case that the array is of even size, that is $n = 2k$, then $M(2k) = 2M(k) + 2k - 1$. Breaking this equation down, we are essentially claiming that the number of comparisons necessary to sort the full list will be the number of comparisons required to sort each half of the list, $M(k)+M(k) = 2M(k)$, and then also the additional comparisons required to reassemble the list $2k - 1$. This is the same as the Bubble Sort best case $n - 1$ where each entry is only compared to one other entry, and comes as the result of knowing that each smaller portion of the array is necessarily sorted before reassembly.

To further analyze the number of comparisons necessary to run the Merge Sort algorithm, we will examine again the case that $n = 2^m$. If this were the case, the array would be able to split exactly evenly at each step of the breakdown, so this case is nice to examine [1].

Let $f(m)$ be the number of comparisons required to sort an array of size 2^m. This size is a special case but the result will be valid asymptotically.

$$\begin{aligned}
f(m) &= 2f(m-1) + 2^m - 1 && (1)\\
&= 2(2f(m-2) + 2^{m-1} - 1) + 2^m - 1\\
&= 2^2 f(m-2) + 2 \cdot 2^m - 2 - 1\\
&= 2^3 f(m-3) + 3 \cdot 2^m - 2^2 - 2 - 1\\
&\vdots\\
&= 2^{m-2}(2f(m-m) + 2 - 1) + (m-1)2^m - 2^{m-2} - 2^{m-3} - \ldots - 1\\
&= m2^m - 2^{m-1} - 2^{m-2} - \ldots - 1 && (2)\\
&= m2^m - (2^{m-1} + 2^{m-2} + \ldots + 1)\\
&= m2^m - (2^m - 1)\\
&= (m-1)2^m - 1
\end{aligned}$$

In (1), we detail that the total number of comparisons will be the sum of the comparisons required to sort each half, left and right, and then the number of comparisons needed to reassemble the list previously discussed to be at most $n - 1$ since $n = 2^m$. Now, we continue to call upon this recursive definition repeatedly hoping that a pattern emerges to help us generate closed form.

By (2), we have found the pattern and do some algebra to make it seem a bit more manageable.

Since $n = 2^m$, we have $m = \log_2 n$. Therefore $f(m)$ is roughly $n \log_2 n$. To be fully rigorous, we should use this technique to suggest the form $f(m) = (m-1)2^m - 1$ and then prove it by induction but from this, we can gather an approximate run-time of $\mathcal{O}(n \log n)$. In what scenarios does this algorithm execute with this efficiency? Let us examine scenarios of Merge Sort.

3.4 Merge Sort Case Scenarios

Using the Merge Sort algorithm, the best and worst case scenarios both have run-time $\mathcal{O}(n \log n)$ [2]. Why is this? If you have the same number of elements in the array, it will take the same number of comparisons to split into subarrays all the way down to the base case. This process is required even if you have a fully sorted array to begin with. The only change in how the algorithm functions comes in the number of comparisons required to reassemble into the sorted "parent array." If we are working with a presorted array, the left side of the subarray would exhaust by placing all elements into the "parent array" before the right subarray has placed a single element. Since the left subarray will contain no more elements to be placed, the pseudocode will place all the elements of the right array into the parent array without making comparisons within, since the subarray is known to be sorted, and this indicates an improvement. The number of comparisons, however, required for reassembly runs in constant time and does not affect overall run-time since we are only concerned with overall growth behavior. Thus, the big-O run time remains the same even though the best and worst case scenarios are not identical.

4 Conclusion

How much better is a run-time of $nlogn$ compared to n^2? In an array of size 10, running these algorithms results in $10log(10) = 23.026$ with Merge Sort and $10^2 = 100$ using Bubble Sort. In an array of size 1,000, however, we are comparing 6908 to 1,000,000 computational units. Even worse, in an actual application, the lists could be longer yet. We have introduced two very different algorithms for accomplishing the same task, but many, many more exist. Clearly, with vast gains to be made in efficiency improvements, the effort required to select an algorithm design and properly code the process pays off in the long run, even if it is a little laborious in the developmental stages.

References

[1] M. Bona, *A Walk Through Combinatorics*, 3rd Ed., World Scientific, 2013.

[2] *Sorting Algorithms Animations*, 2015, `http://www.sorting-algorithms.com`.

[3] A. Nayan, *Merge Sort Algorithm*, 2014, `https://www.youtube.com/watch?v=TzeBrDU-JaY`.

[4] A. Nayan, *Bubble Sort Algorithm*, 2013, `https://www.youtube.com/watch?v=Jdtq5uKz-w4`.