Sorting Algorithms

M. Hassell

February 1, 2017

1 Introduction

The goal of this document is to gather key sorting algorithms and some analysis and discussion so I can understand where they fit, how they work, their strengths, and their weaknesses. This is going to supplement implementing these algorithms in C++. The code will also be in this directory. I intend to follow [1]. Any questions, comments, or suggestions are welcome.

2 Insertion sort

2.1 The idea

Insertion sort is one way to sort a list of numbers. Suppose the list is denoted v. We start by picking the second element of the array and calling it "key." We then look backwards one place (at the first element of the array) and compare it to the value stored there. If it is greater than the first element of the array, we're done. If not, then we set the second value of the array to be the same as the first, and then insert in the first position the value of "key." We can continue this throughout the rest of the array. We look at the third element and compare it to the second. If we're greater than the second element, we're done. Otherwise, shift the second element to the third index in the array, and continue down the line until we either find where we need to place the current value, or we hit the end of the array (then the current key is the smallest value and goes in the first position in the array). Here's the pseudocode:

```
for j=2 to A.length
        key=A[j]
        i=j-1
        while i>0 and A[i]>key
                A[i+1]=A[i]
                i=i-1
        A[i+1]=key
```

To analyze the complexity of the algorithm, we count the number of instances each line is executed. I'm not going to go through the details of $\mathcal{O}$ notation and the constants and all. Here we go.

```
for j=2 to A.length                                  (n-1)
        key=A[j]                                     (n-1)
        i=j-1                                        (n-1)
        while i>0 and A[i]>key                       sum_{j=2}^{n} t_j
                A[i+1]=A[i]                          sum_{j=2}^{n} (t_j - 1)
                i=i-1                                sum_{j=2}^{n} (t_j - 1)
        A[i+1]=key                                   n-1
```

Note: the book says that the top level for loop iterator is executed n times, but the list 2 to n contains $n-1$ elements, so is this a typo, or am I missing something? I haven't been able to find if this is a typo. It doesn't impact the complexity of the algorithm, but I'm still quite curious. Anyway, we need to fill in the values t_j to complete the analysis. The best case performance occurs when the array is already sorted, so we will always find that A[i]>key is false. So $t_j = 1$ for all j (we have to check the condition once before moving on). Summing up 1 from 2 to n gives $\mathcal{O}(n)$ performance over all, so in the best case, we have linear complexity when sorting already sorted arrays. Notice that we sum from $j = 2$ to n because we're taking into account the outer loop. In my mind, it feels like there should be a multiplication here, but I need to make this more precise.

Now we consider the more interesting worst-case performance. In this case, we need to shift the entire array to the right before dropping the key in the right place. This corresponds $t_j = n$. We can conclude that the complexity in the worst case is thus $\mathcal{O}(n^2)$.

Here's a thought. Instead of the summation, which is bothering me, consider this argument: In the outer loop, we touch every element, for a linear complexity. Independently, we have that i can range from 1 up to $j - 1$. The outermost loop touches n elements, while the inner `while` loop touches up to $j - 1$ elements per iteration. We multiply and get that the complexity is $n(n-1) = \mathcal{O}(n^2)$. This isn't completely clear just yet, but I think it can get there.

3 Merge Sort

Merge sort uses a divide-and-conquer approach to sort an array. Beginning with an unsorted array, we recursively split it into smaller arrays until at the deepest level we have a number of singleton arrays. These are all trivially sorted. We then work back up the recursion by merging the sorted arrays at each level to produce a new sorted array. The key to merge sort is the merge operation. The rest of the algorithm falls out once we have the merge tool. We take as arguments A, an array containing two already sorted sequences, and three indices: p, q, and r. The elements of the first sorted sequence are stored in $A[p, q]$, and the second sorted sequence is stored in $A[q+1, r]$. Let's take a look.

```
merge(A,p,q,r)
        n1 = q-p+1
        n2 = r-q
```

```
for j=1 to n1
        L[j] = A[j]
for j= 1 to n2
        R[j] = A[q+j]
i=1 // index into L
j=1 // index into R
for k = p to r
        if i<n1 and j < n2  // L and R both contain entries
                if L[i]<R[j]
                        A[k]=L[i]
                        i++
                else
                        A[k]=R[j]
                        j++
        else if i==n1 and j<n2  // L is empty
                A[k]=R[j]
                j++
        else if i<n1 and j==n2 // R is empty
                A[k]=L[i]
                i++
```

The authors [1] provide a great analogy with merging two decks of already sorted cards. With this merge procedure in place, merge sort is trivial:

```
merge_sort(A,p,r)
        if p<r
                q = floor(p+r/2)
                merge-sort(A,p,q)
                merge-sort(A,q+1,r)
                merge(A,p,q,r)
```

We simply take the array we desire to sort, split it in two, and recursively sort each of those sub arrays. We include the simple check $p < q$ to ensure that we have a non-trivial array. Upon sorting each of the sub arrays recursively, we merge them according to the merge method.

3.1 Analysis of merge sort

To analyze merge sort, we can use a recurrence relation. We let $T(n)$ denote the running time of merge sort on an algorithm on a problem of size n. We assume that for a small enough problem, we can sort the array in constant time. Now when we split the problem in two, we have a running time of $2T(n/2)$. If we suppose that it takes $C(n)$ time to merge the problems generated after sorting sub-problems of size n, and $D(n)$ time to divide a problem, then we have as an overall running time for a problem of size n (for n large enough) $2T(n/2) + C(n) + D(n)$. As a side note, for different divide and conquer

algorithms, we can write the running time for a problem of size n as $aT(n/b)$, for positive integers a and b.

We now assume that the array to be sorted is a power of two, which allows the array to be cleanly split into two further cleanly divisible sub-arrays. This assumption doesn't hinder the analysis. Clearly, the splitting the array in twain takes constant time $\Theta(1)$ since we're only computing the midpoint of the array. Next, the merge operation takes linear time, so we have that $C(n) = \Theta(n)$. Thus, we have the following recurrent relation for the complexity of the merge-sort method:

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

By way of a "master theorem" that I need to see and prove (!) it can be show that the run time of merge sort is $T(n) = \Theta(n \lg n)$, where $\lg n$ is $\log_2 n$. Remark: I've wondered where the ln term comes from in the analysis of some algorithms. In this light, it becomes clear where this transcendental function comes in. Under the assumption that our input array contains a number of elements that is a power of two, we see that we need to recurse $\lg n$ levels to reach the leaves of the recursion tree. Since for any a and b we have $\log_b = \Theta(\log_a)$, so the asymptotic bound holds for the natural log as well.

An interesting exercise in the book is to coarsen the leaves of the recursion tree and apply insertion sort at the leaves to achieve a more favorable constant in the asymptotic bound for merge sort. This is done by fixing a number k such that for arrays of size k, we apply insertion sort to sort these small arrays, and then merging them back up the recursion tree. I've implemented this as `merge_sort_truncated`, which takes the usual merge sort parameters, plus the size of the arrays at the bottom level that get insertion sorted. I made a modified function for using insertion sort at the bottom level to avoid copying the values in the vector of interest into another vector of the needed side (since insertion sort works with the size of the array passed to it). Some thought shows that the complexity of the truncated merge sort method is $\Theta(n \lg n/k)$ where k is the size of the bottom level arrays that are merge sorted. You can use an argument similar to the recursion tree in figure 2.5 of [1] to get the result.

References

[1] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.