Sorting Algorithms

M. Hassell

January 23, 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.

References

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