Recursive Algorithms

Margaret M. Fleck

21 October 2009

This was a short lecture due to the quiz. We saw insertion sort (in sections 3.1 and 3.3 of Rosen) and the ideas behind merge sort (in section 4.4 of Rosen).

1 Recap

Last lecture, we saw the linear and binary search algorithms and analyzed how long they take. We also saw bubble sort, a really dumb algorithm for sorting an array or list of numbers, which not only takes $O(n^2)$ time (where n is the length of the input array/list) but tends not to have good constants.

Today, we'll see two more reasonable ways to do this task: insertion sort and merge sort. Insertion sort is still $O(n^2)$ but tends to be faster than bubble sort. We'll also see merge sort, which has the best big-O running time: $O(n \log n)$. Merge sort has some extra overheads which make it only so-so for small sorting tasks, but it's the approach of choice for sorting really large amounts of data.

In many practical applications, the sorting algorithm of choice is often quicksort, another $O(n^2)$ algorithm that has very good constants and a fast average-case running time. However, we will leave it for another class because its running time is harder to analyze.

2 Insertion sort

[I did a little demo of insertion sort with moving number tiles. If you missed lecture and need to see a demo, I think there are some on the web. Or you can come to office hours.]

Remember that sorting algorithms take an input array of numbers and move the numbers around in the array until they are sorted, i.e. in numerical order. Bubble sort and insertion sort do not explicitly return a value. Rather, the caller is expected to know that his input array will have had its contents rearranged.

The idea behind insertion sort is to divide the input array (conceptually) into two pieces: a portion that has been sorted and a portion that isn't sorted yet. At the start, the sorted portion contains only a single number: the original first number in the array. We add the other numbers one by one, inserting each one into its proper place in the growing sorted list.

When we add some new number a_j to the sorted part of the array, we first remove it from the array. Then we first need to find its proper position in the sorted portion. Then we usually need to shove some of the sorted values to the right to make a space to insert a_j. The we actually insert a_j.

Our pseudo-code looks like

1: insertionsort(a_1,..., a_n: array of reals)

2: for j = 2 to n

{

3: t = a_j

4: i = 1

5: while ($a_j > a_i$)

6: i = i + 1

7: for k = j down to i + 1

8: $a_k = a_{k-1}$

9: $a_i = t$

}

The inside of the big loop moves the value a_j from the unsorted part of the array into the sorted part. Lines 4-6 locate the place where a_j belongs in the sorted portion, which will be right before the value a_i.

Lines 7-8 then shove the sorted values starting with a_i over to the right to make a blank space, into which line 9 puts a_j.

Notice that the loop index in line 7 moves downwards rather than upwards. In some programming languages, you can just do this directly. In

others, you (annoyingly) need to use an upward-moving loop variable and do some algebra on the index values to derive the downward-moving indices.

3 Running time of insertion sort

Suppose that $T(n)$ is the worst-case running time for insertion sort. That is, for each input array length n, $T(n)$ is the maximum time that insertion sort might take on any array of that length. Notice that we now have two closely-related functions floating around: insertionsort and its running time T.

Insertion sort has two nested loops. This should immediately suggest to you: $O(n^2)$ time. So let's check out the details of the loops to see if that's right.

The inner loop (line 5-6) runs no more than j times. If it hasn't already halted earlier, it definitely has to halt when $i = j$. To be really precise, the test for this loop (line 5) gets executed no more than j times and the material inside the loop (line 6) no more than $j - 1$ times.

So, the code inside the loop runs no more than $\sum_{j=2}^{n} j = \sum_{j=1}^{n-1}(j+1) = (n-1) + \frac{n(n-1)}{2} = O(n^2)$

Other parts of the code run less often. E.g. the loop index computation and testing in line 2 takes $O(n)$ time. And there's probably some constant-time work at the start and end of the function. But this work is dominated by the $O(n^2)$ work in the inner loop.

So, $T(n) = O(n^2)$.

4 Merge sort

[I did a little demo of mergesort. Again, see the web or come to office hours for a demo if you need one.]

Like bubble sort and insertion sort, merge sort takes an input list of numbers and rearranges them to produce a sorted output list. I have said "list" here, rather than "array," because merge sort implementations typically store their values in linked lists rather than in arrays. The pseudo-code doesn't actually distinguish lists from arrays and you could use either lists or arrays to implement any of these algorithms. (Also, don't worry if you haven't seen linked lists yet. Just use your intuitive understanding of what a list is.)

Merge sort is a so-called divide-and-conquer algorithm. That is, it takes a big problem (sorting a list of length n) and divides it into two smaller problems (sorting two lists of length $n/2$). We divide up the list repeatedly until we have a large number of very short lists, of length 1 or 2 (depending on the preferences of the code writer). A length-1 list is necessarily sorted. A length 2 list can be sorted in constant time.

Then, we take all these small sorted lists and merge them together in pairs, gradually building up longer and longer sorted lists until we have one sorted list containing all of our original input numbers.

The big trick here is merging two sorted lists, $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$. To do this, we make a new third list to contain our merged output. Then we examine the first elements of the two input lists and put the smaller value into the starting position of our output list. We keep looking at the first elements of both lists until one list is empty. We then copy over the rest of the non-empty list.

This merger process is very efficient, because each merger step only examines the two values at the start of the two lists.

Next lecture, we'll see the pseudocode for merge sort and analyze its running time. We'll see that it takes $O(n \log n)$ time. This is the best possible big-O running time for a sorting algorithm.