Recursive Algorithms

Margaret M. Fleck

19 March 2010

This lecture continues coverage of sorting and search algorithms (sections 3.1 and 3.3 of Rosen).

1 Binary search

Last lecture, we saw a brief sketch of binary search. Let's look at this algorithm in more detail.

In many applications, it's sensible to assume that the input array is sorted. That is $a_1 \leq a_2 \leq \ldots \leq a_n$. We'll see that it would take $O(n \log n)$ time to sort an unsorted array. But sometimes an array of data comes pre-sorted. And sometimes an application will do many searches on an array of data that's constant or changes only very slowly, e.g. the registrar's big database of student records. In such cases, the cost of sorting the array may be insignificant compared to the cost of doing the searches.

When our input array is sorted, we can use a much more efficient algorithms: binary search. Let's assume that our array is sorted with the smallest element at the start. (The same idea works, with small code changes, if the array is sorted in reverse order.)

```
binarysearch(a_1,..., a_n: sorted array of reals, b real)
    i = 1
    j = n
    while (i < j)
        m = ⌊(i + j)/2⌋
```

6: if ($b > a_m$)

7: then i = m+1

8: else j = m

9: if (a_i = b) return i

10: else return 0

The two variables i and j define a range of array positions. At the start of the function, this range is all of the input array. In each iteration of the loop, we shrink the range in half, either by making i larger or by making j smaller. The loop terminates when the range contains only a single array position i.

To do the shrinking, we examine the value at the middle position in the current range. If it's smaller than b, we know that b must live in the upper half of the range (if it's in the array at all). Otherwise, b must live in the lower half of the range (again, if it's in the array).

2 Analysis of binary search

To do a big-O analysis of binary search's running time, notice that the code in lines 1-3 and 9-10 takes constant time, i.e. the same amount of time regardless of the input array size. So it's $O(1)$ time and doesn't matter to our big-O analysis. The code inside the loop (lines 5-8) also takes constant time. So the running time of the code (as a function of n) is determined by how many times the loop runs.

To figure out how many times the loop runs, first suppose that n is a power of two, e.g. $n = 2^k$. Since the range starts at length n and is cut in half in each iteration, it will take k iterations before the range has length 1 and the loop stops. If you have to do such an analysis really fast, e.g. on an exam, you might be uncertain about whether the loop really runs k times or perhaps $k + 1$ or $k - 1$ times. This difference doesn't matter: it's still $O(k)$ times. Since $k = \log_2 n$, this means that the loop runs $O(\log n)$ times.

If n isn't a power of two, we can get an upper bound on the running time. The code run on an array of size n will clearly take no more time than if it were run on an array of size 2^k where 2^k is the next power of two larger than n. So, the loop still runs $O(\log n)$ times.

So, the overall running time of binary search is $O(\log n)$, where n is the length of the input array. In general, if an algorithm works by reducing the size of its problem by a factor of two (or three or four ...), it probably takes $O(\log n)$ time.

3 Bubble sort

A sorting algorithm takes an input array of real numbers and puts them into sorted order, e.g. smallest number first. Sorting algorithms are not actually restricted to numbers. With some minor changes, they can operate on any sort of data where you can precisely define an ordering. For example, they can be used to sort strings (e.g. names) into dictionary order. We'll stick to real numbers for simplicity.

The bubble sort algorithm is an especially stupid sorting method. So stupid, in fact, that you're unlikely to see it outside of theory classes. But it is a useful starting point for looking at the running times of sorting algorithms.

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

  2: for j = 1 to $n - 1$

    3: for k = 1 to $n - j$

      4: if ($a_k > a_{k+1}$)

        · 5: swap a_k and a_{k+1}

This function doesn't return anything explicitly. Rather, it rearranges the values within its input array, so the input array will be nicely sorted after the function returns. (Rather like organizing someone's closet for them.)

The swap command in line 5 interchanges two values in the array. To implement this in most programming languages, you'd need to put one of the values into a temporary variable e.g. $t = a_k$, then move the other value into the first one's e.g. $a_k = a_{k+1}$, then put in the new second value e.g. $a_{k+1} = t$. Normally we hide this gory detail when writing pseudocode.

Bubblesort walks repeatedly through the input array, fixing local problems in the ordering. Some versions of bubblesort just keep sweeping through the entire array repeatedly until they do a complete sweep without finding any more local problems.

This version of bubblesort is a bit smarter. After the first time through the outer loop (line 2), we know that the largest value must have worked its

way all the way down to the last array position. So this last position has its final value and so we can ignore it in subsequent iterations. After the second time through the outer loop. the last two positions are ok and can be left alone. This is why the top index in the inner loop (line 3) keeps decreasing.

The running time of this algorithm is proportional to how many times the code inside both loops (lines 4-5) runs. This is

$$\sum_{i=1}^{n-1}(n-i) = (n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

When reversing a summation like this, it would be easy to get something off by one, e.g. the top or bottom bound. Writing out the terms of the summation is helpful for checking this.

4 The big picture

Bubble sort is 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. Let's now look at 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.

5 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.

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

7 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.