Computer Science 50
Fall 2009
Scribe Notes

Week 3 Wednesday: September 23, 2009
Andrew Sellergren

Contents

Computer Science 50
Fall 2009
Scribe Notes

Week 3 Wednesday: September 23, 2009
Andrew Sellergren

1 Announcements

- This is CS 50.
- If you haven't already, please turn in your sensor boards from Problem Set 0.
- Problem Set 2 has been released.
- For questions which don't require sharing a large portion of code or generally giving away too much, feel free to post them to the Bulletin Board. By default, when you login, you will be assigned the name "student" to keep you anonymous, but you can change this if you like.
- Check the website for the Office Hours schedule. Before you come to Office Hours with a question like "Where do I begin?" on the problem set, be sure to watch Marta's walkthrough. We do move quickly in this course, but once you start really plugging away at it (learning by *doing*), we think you'll be surprised at how capable you are.

2 Selection Sort and Bubble Sort (0:00–25:00)

- For this demonstration, we ask 8 volunteers to come on stage and hold pieces of paper with the numbers 1 through 8 in a somewhat jumbled order. Then Adam will be asked to sort them.
- Adam's approach, as described by another student, was to find the smallest number and move it to the leftmost position. Then repeat the process with the remaining 7 numbers. Adam makes one addition: he checked to see if a number was already in the correct position.
- Let's talk about this from a computer science perspective. What data structure might we use to represent these 8 numbers? An array would be good. Of course, while Adam knew almost immediately that the smallest number was in index location 4, he only knew this because he had already scanned the other numbers. In the same way, a computer will only know where the smallest number is after iterating through all the numbers and comparing them. In other words, it will take n steps to find the smallest number.
- Once we know where the smallest number is, how do we put it in its proper place? We need to make room for it. If we switch it to the leftmost position, we could shift all the other elements down one. But this will actually require multiple steps. Why not just swap positions with the leftmost element? Then we only need a single step to put it in position.
- How many steps total is this algorithm, which we formally call *selection sort*? As we said, the first iteration will take n steps. Once the first element is in position, though, we need only iterate over the remaining

$n-1$ elements to find the next-smallest number. So the second iteration takes $n-1$ steps, the third iteration takes $n-2$ steps, and so on. What's the sum of n and $n-1$ and $n-2$, etc.? If you remember from high school math, it's $\frac{n(n+1)}{2}$, or approximately n^2. The reason we can approximate it as n^2 is that for very large numbers, the n in the above formula tends not to add much compared to n^2.

- Recall that our counting-by-two algorithm technically doubled the speed of our counting-by-one algorithm. However, we tend to throw out constants like (1/2) in discussing big O because, again, they don't really make much of a difference as n gets very large.

- Big O notation allows us to discuss the interesting aspects of algorithms independent of things like processor speed, which might change every six months.

- What about the best-case scenario? Best-case scenario would be if Adam arrived on stage and the 8 numbers were already sorted. But using our current algorithm, we're still going to take n^2 steps to sort it because we have no checks in place to see if the array is already sorted.

- Let's talk about another algoritm called *bubble sort*. In this algorithm, we'll iterate down the array and if we find two numbers next to each other that are out of order, we'll swap them. This is a decent approach, but it turns out it's also in $O(n^2)$. In the worst case, if the largest number in the array happens to be all the way to the left, then we'll have to swap it n times to get it in its proper place. If the second-largest number is second from the left, we'll have to swap it $n-1$ times to place it properly. And so on.

- Check out this Java applet which demonstrates the sorting algorithms we've been talking about.

- Here we write out bubble sort in pseudocode:

```
Repeat n times:
  For each element i:
    If element i and its neighbor are out of order:
    Swap them.
```

We can improve on the best-case running time by checking for the number of swaps that are made on a given iteration. If no swaps were made, then we don't need to continue the algorithm. With this improvement, bubble sort goes from $\Omega(n^2)$ to $\Omega(n)$.

3 Merge Sort (25:00–45:00)

- It turns out that we can employ recursion to come up with a sorting algorithm that improves on the running time of bubble sort and selection sort. Take a look at the pseudocode for *merge sort*:

```
On input of n elements:
  If n < 2, return.
  Else
    Sort left half of elements.
    Sort right half of elements.
    Merge sorted halves.
```

  What do we mean when we say "sort" each half of the elements? Well, this is a recursive algorithm, so what we really mean is to call merge sort again but on only half of the original list. We keep repeating this until our list size is just 1, in which case the single element is already "sorted" in a manner of speaking. The magic, then, must be in our how we merge the sorted halves.

- Consider the following array: 4, 2, 6, 8, 1, 3, 7, 5. Using merge sort, we're going to recursively chop it in half until we're left with two lists of size one: 4 and 2. Each of these lists is already "sorted," so now we need to merge them. How do we merge? Well, 2 is less than 4, so we want to put the 2 list before the 4 list. Where will we store these? In reality, we're going to need more memory to store the sorted total list.

- Now that 2 and 4 are in order, we step back for a second. 2 and 4 comprise the left half of a list of four: 4, 2, 6, 8. So now that we've sorted the left half of this list, we need to sort the right half, namely 6, 8. Again, we divide this list of length two into two lists of length one: 6 and 8. Then, we merge them, putting 6 before 8.

- Now, again we're at the merging step. The left half is 2, 4 and the right half is 6, 8. Let's point our left hand at the first element of the left half and our right hand at the first element of the right half. $2 < 4$, so we put 2 in first. Now we'll advance our left hand one step so that it's pointing at 4. Our right hand is still pointing at 6 because we haven't merged that number yet. We compare 4 and 6 and then put 4 into our sorted array. Then we put 6 and 8 in because there's nothing to compare them against.

- At this point, we've completed the very step in our first call to merge sort: we've sorted the left half of the entire original list. Now we'll sort the right half: 1, 3, 7, 5.

- Cutting out a few of the intermediate steps, we end up with two sorted lists of size two: 1, 3 and 5, 7. When we merge them we end up with 1, 3, 5, 7. Finally, we'll merge this right half with the original left half, 2,

4, 6, 8. Obviously, we'll end up with 1, 2, 3, 4, 5, 6, 7, 8. We'll do so by iterating over both lists comparing the current left number to the current right number, selecting the smaller, and advancing to the next number in the list we just took a number from.

- The merging step will take n steps because we must iterate over both lists of size $n/2$. But what about the division steps? We only executed $\log n$ divisions. So our total merge sort algorithm is in $O(n \log n)$. This is an improvement on $O(n^2)$!
- Let's try to represent merge sort's running time, $T(n)$, formulaically:
  - Let $T(n)$ = running time if list size is n.
  - $T(n) = 0$ if $n < 2$
  - $T(n) = T(n/2) + T(n/2) + O(n)$ if $n > 1$

  That is, we have to sort the left half, which takes $T(n/2)$, sort the right half, which takes $T(n/2)$, and merge, which takes $O(n)$!
- Ex: Suppose we want to find $T(16)$:
  - $T(16) = 2T(8) + 16$
  - $T(8) = 2T(4) + 8$
  - $T(4) = 2T(2) + 4$
  - $T(2) = 2T(1) + 2$
  - $T(1) = 0$
  - $T(16) = 2(2(2(2(0 + 2) + 4) + 8) + 16) = 64$

  Eventually we boil down to $T(1)$, which is 0 because a list of size one is already sorted. Does our final result, 64, agree with our original determination of $O(n \log n)$. Well, $16 \times \log 16 = 64$, so yes. Compare this to $O(n^2)$, which would take $16^2 = 256$ steps. Already we're reaping the benefits.
- To visualize this, take a look at this animation. Specifically, try comparing bubble sort and selection sort with merge sort by running them side by side.