Computer Science 50
Fall 2011
Scribe Notes

Week 3 Wednesday: September 21, 2011
Andrew Sellergren

Contents

Computer Science 50 Week 3 Wednesday: September 21, 2011
Fall 2011 Andrew Sellergren
Scribe Notes

1 Big O Notation (0:00–16:00)

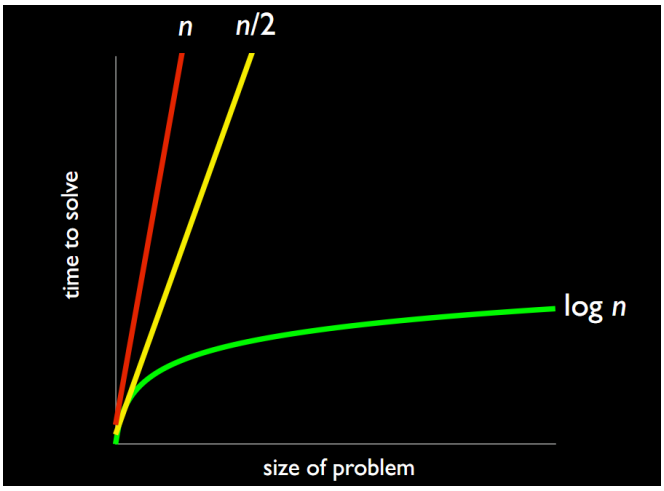
- Recall from Week 0 the phonebook demo. We learned that successively tearing the phonebook in half was much more efficient than turning page by page in order to find Mike Smith. This is the difference between a logarithmic and a linear approach.
- We also looked at an algorithm for counting the number of students in Sanders:

  1. stand up and think of the number 1
  2. pair off with someone standing, add your numbers together, and adopt the sum as your new number
  3. one of you should sit down; the other should go back to step 2

  Here again, the logarithmic approach, the divide-and-conquer approach, in which half of the students in Sanders sat down on each iteration, proved much faster.
- Let's try another algorithm that uses the logarithmic approach in order to find the tallest person in the orchestra section:

  1. stand up if in orchestra section
  2. pair off with someone standing; stay standing if you're taller, sit down if you're shorter; break tie randomly
  3. if still standing, go to step 2

  Renzo is the tallest person in the orchestra section today! Even if we had started with as many as 4 billion people, it would have only taken 32 steps to find the tallest person using this algorithm ($2^{32} \approx 4$ billion). The alternative, the linear approach, would take 4 billion steps.
- We can visualize this difference in runtime for these algorithms using the following graph:



On the y-axis, we show a generic count of the number of steps it takes to solve a problem. This number of steps is roughly proportional to time. On the x-axis, we show the size of the problem, which we call n. n is the number of things we're counting or sorting or comparing. In the case of finding the tallest person among 4 billion people, n would be 4 billion. As you can see, as n gets larger, the linear approach quickly diverges from the logarithmic approach. Even an approach that takes $n/2$ steps doesn't do much better. For example, counting every person in Sanders two at a time rather than one at a time isn't much faster.

- Interestingly, there are also algorithms that perform much worse than linear. Generally, we'll want to avoid anything that is exponential. In some cases, though, it actually provides the best solution. For example, the algorithm which would provide a solution that maximizes student happiness in sectioning is exponential in nature. For that reason, FAS doesn't use it because it simply takes too long. Thus, the algorithm they use is suboptimal.

- Computer scientists use what's called big O notation to discuss the worst-case runtime of algorithms. We would say that the algorithm for counting everyone in Sanders one at a time runs in $O(n)$. What this means is that in the worst case, this algorithm would take n steps to complete, given n inputs. The algorithm for counting everyone in Sanders two at a time runs in $O(n/2)$ and the algorithm for counting everyone in Sanders using the divide-and-conquer approach runs in $O(\log n)$. Actually, in the coming weeks, we'll simplify $O(n/2)$ to be just $O(n)$ since in the long run, constant coefficients have negligible effect.

2 Search (16:00–32:00)

- Last time, we talked about arrays as collections of related variables of the same type stored in contiguous memory. To access each element of the array, we used bracket notation. We also observed the dangers of iterating off the end of an array and touching memory that doesn't belong to us, causing a segmentation fault.
- On the board are two arrays of integers covered by pieces of paper. These are meant to simulate how a computer sees them. Whereas we as humans can glance at a series of integers and see all of them at once, a computer can only look at one integer at a time. Hence the pieces of paper covering the integers.
- Let's watch Sean as he tries to find the value 7 in the top array. Having no foreknowledge of the array, he looks behind pieces of paper moving left to right and finds that 7 is the last number in the row. Of course, this process isn't very efficient. In the worst case, for an array of length n, the very last number we examine will be the one we are searching for, so we'll have to walk through n steps to find it. With no foreknowledge of the array, this is the best we can do: to brute force examine every single element of the array. Looking under the pieces of paper two at a time is a marginal improvement, but with a large enough n, this improvement is negligible.
- Note that in the best case, the number we're searching for will be the first one we examine, so the algorithm only takes 1 step. We denote this as $\Omega(1)$, where omega refers to the best-case scenario runtime.
- Now let's ask Jeremy to search the bottom array for the number 50, this time knowing that the array is sorted in ascending order from left to right but still not knowing what numbers it contains. Jeremy jumps to the middle of the array and finds the number 61. Now we can disregard the right half of the array since our number is less than 61 and the array is sorted. With the remaining left half of the array, Jeremy again chooses a number in the middle and finds 50. Success! In this case, Jeremy found the number in 2 steps, but even in the worst case, it would only have taken 1 more step for a total of 3. Compare that to the 8 steps that Sean took.
- Dealing with a sorted array and using binary search, we were able to greatly reduce the number of steps it took to find a specific number. But how do we sort an array?

3 Sort (32:00–58:00)

3.1 Selection Sort

- Realize that when it comes time to implement search and sort algorithms, there might not be an exact middle to the array or list of items. You'll

need to make sure to round up or down after you divide the number of items in the array or list by two.

- 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. If we were to represent these 8 numbers in a computer program, we'd probably use an array rather than 8 separate variables. As a result, the computer itself can't see the values of all the variables at the same time. This is an important consideration for us as we design our sorting algorithms.

- Casey is our volunteer who will be sorting the other volunteers on stage. In her first attempt, she places the number 1 in the first position of the array, and shifts the other numbers down. Then she places the number 2 in the second position of the array, and shifts the other numbers down. And so on. How many steps does this take to sort the entire array of 8? Although it would seem that it took only 8 steps, one for each number that was placed in its correct position, this estimate glosses over the additional steps that were required to find the next smallest number and to shift the other numbers down the array each time a number was placed in its correct position.

- One optimization we can make is instead of shifting the other numbers down the array, simply swap the next smallest number with the number that is in its correct position. So, in the first case, we swap the number that's in index 0 of the array with the number 1. This only requires 1 step rather than the several it takes to shift numbers down the array.

- On the first iteration of our sorting algorithm, it actually took us n steps to find the smallest number since we have to traverse the entire array. On the second iteration of our sorting algorithm, however, we already know that index 0 correctly contains the smallest number, so we can skip it when looking for the next smallest number. Thus, searching for the smallest number takes n steps on the first iteration, $n-1$ on the second iteration, and so on. In addition to these searching steps, there is a swap step on each iteration of the algorithm, but we can ignore this as negligible when calculating big O notation for this algorithm. Our whole algorithm then takes $n+n-1+n-2+n-3\ldots$ which, if you remember some high school math, simplifies to $n(n+1)/2$. In short, this algorithm takes about n^2 steps, so we say that it is in $O(n^2)$. Formally, this algorithm is called *selection sort*.

3.2 Bubble Sort

- Our second sorting algorithm involves starting at the beginning of the array and examining the first two numbers. If the left number is greater than the right number, we know intuitively that they are out of place, so

we swap them. Then we iterate to the next two numbers and compare them in the same way.

- What's the worst-case scenario for this algorithm? If the array is in reverse order, it will take the maximum number of steps to sort using this algorithm. If the number 8 is at index 0, it takes 7 swaps to move it to its correct position on the first iteration of the algorithm. On the second iteration of the algorithm, it takes 6 swaps to move the number 7 from index 1 to its correct position. And so on. Looks like in the worst case, this algorithm also takes $n+n-1+n-2+n-3\ldots$ steps, so we're back at $O(n^2)$. Because of the way numbers bubble up from one end of the array to the other, this algorithm is called bubble sort.

3.3 Omega

- We've talked about the worst-case runtimes for selection sort and bubble sort, but what about their best-case runtimes? In the world of sorting, the best case is certainly that the array is already sorted.

- As it turns out, selection sort runs in n^2 even in the best-case scenario. Remember that the computer doesn't know at the time it sees 1 in index 0 of a sorted array that 1 is the smallest number in the array. It still has to traverse the entire array to make sure. Likewise with the second iteration, it still takes $n-1$ steps to find the smallest number. And so on. Bummer, selection sort is in $\Omega(n^2)$.

- With a certain optimization, bubble sort can be in $\Omega(n)$. If the array is already sorted and we iterate through every element of it on the first iteration, we won't need to make any swaps at all. If we made no swaps on the first iteration, then there's no need for a second iteration, since it too will make no swaps. Thus, assuming we check if any swaps were made, we need only one iteration of the algorithm in the best-case scenario. This means n steps.

3.4 Simulations

- To visualize these sorting algorithms, take a look at this demo courtesy of D. Thiebaut from Smith College.[1] Note that the shorter bars represent smaller numbers and the longer bars represent larger numbers. With this demo, you can really get a sense for just how slow selection sort and bubble sort really are!

- This simulation allows you to compare sorting algorithms side by side. As a teaser for next time, try watching selection sort, bubble sort, and a new contender, merge sort, duke it out.

[1] Unfortunately, this demo doesn't work on Safari or Chrome, it appears.