Computer Science 50
Fall 2009
Scribe Notes

Week 3 Monday: September 21, 2009
Andrew Sellergren

Contents

Computer Science 50
Fall 2009
Scribe Notes

Week 3 Monday: September 21, 2009
Andrew Sellergren

1 Announcements (0:00–11:00)

- This is CS 50.
- 1 new handout.
- 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.
- Happy birthday to Cansu and Mike T.!
- Don't make your password 12345.
- 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.

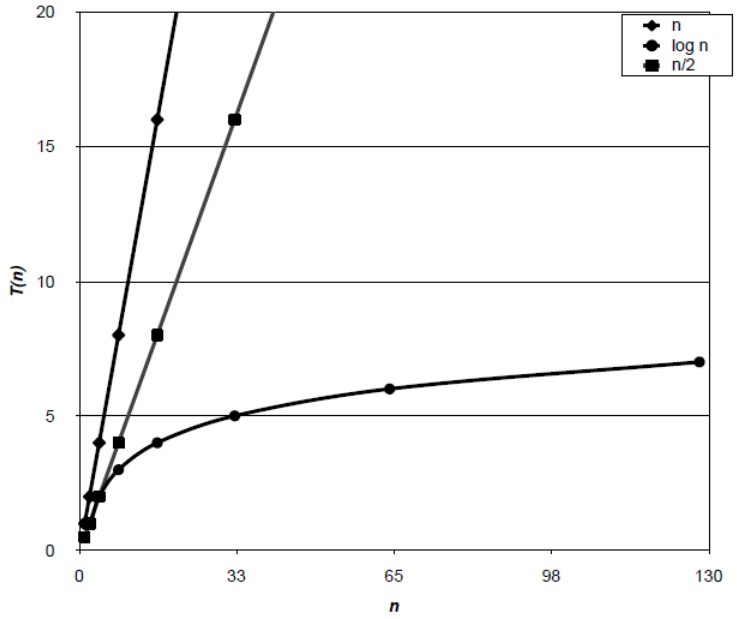
2 Cryptography (3:00–15:00)

- You'll find that even a field of computer science which appears complex is actually composed of simple steps.
- As a result, of course, some cryptographic algorithms, like the Caesar cipher, for example, are vulnerable to being cracked. How would you go about cracking a password encrypted with the Caesar cipher? You could do a frequency analysis, which would map the most commonly occurring letters in the ciphertext to the most commonly occurring letters in the alphabet. Of course, you could also just use brute force and try all keys 1 through 26.
- If we want to step it up a notch, then, we might implement the Vigenère cipher. Instead of rotating by a single number, we rotate each character in the message by a different number derived from a key word. If the key isn't as long as the plaintext message, then just repeat the key word as necessary.
- For example, if p = HELLO,WORLD and k = FOOBAR, then we get c by doing H + F = M, E + O = S, L + O = Z, etc., to come up with c = MSZMO,NTFZE. We can formalize this like so: $c_i = (p_i + k_i) \bmod 26$.

- Whereas the *keyspace*, the number of possible keys, for Caesar is only 26, the keyspace for Vigenère is 26^n, where n is the number of letters in our keyword.
- Of course, neither is Vigenère foolproof. To brute force crack it, we would need only try all possible 1-letter keys, followed by all possible 2-letter keys, and so on. With the power of today's CPUs, this is actually not as slow a process as we might hope.
- Another step up is the DES algorithm, which is still used to store passwords on many Linux systems. It offers 72 quadrillion possible keys via a complex formula which we won't go into here. Even so, it can be cracked!
- Nowadays, the RSA algorithm is widely used, from browsers to ATMs. Ultimately, it is based on a simple assumption: factoring a very large number into two very large primes is so computationally intensive as to be near-impossible. In other words, it couldn't be done in the span of a human lifetime, for example.

3 Asymptotic Notation (15:00–35:00)

- Recall from Week 0 the Phonebook Example, in which we searched for Mike Smith using binary search or "divide and conquer." Each time we divided the phonebook in half, we were cutting the problem in half, so what was once a 1024-step problem (if we had flipped through every single page) became a 10-step problem. And if the phonebook had been 4 billion pages long, the problem would've only required 32 steps to solve ($2^{32} \approx 4$ billion).
- In general, we'll use the variable n to denote the size of the problem, or the number of steps it requires to solve. Each step is roughly accomplished with a single CPU cycle (assuming it has a single core).
- So when we were counting the number of students in the lecture hall, how much faster was it to use the algorithm whereby you students paired off and added your totals versus David standing at the front counting one at a time? Well, if you think about it, each step that David executed was only taking care of $1/n$ of the problem whereas each step of the pairing algorithm was taking care of $1/2$ of the problem.
- Let's take a look at a graph of running time versus n for these algorithms:



The n vs. n line represents David's counting algorithm. The $n/2$ vs. n line represents a counting-by-two algorithm. Finally, the $\log n$ vs. n graph represents the pairing algorithm. Notice the dramatic difference in running times.

- If one more student comes into the room, it takes David one more step, but it takes you roughly the same number of steps. If 200 to 300 students entered the room, it will take David roughly twice as many steps. It will take you, on the other hand, only one more step.
- Just for reference, algorithms for factoring large prime numbers generally run in exponential time.
- To standardize our discussions of running times, let's introduce the following symbols: O, Θ, Ω. O, or big O, denotes the worst-case running time, Ω, or omega, denotes the best-case running time. If both happen to be the same for a given algorithm, then we bring in Θ, or theta.
- So, for example, David's counting algorithm takes both at best and at worst n steps. So we can say it's in $\Theta(n)$. If we talk about an algorithm for searching for a certain student named Mike and we search the students one by one as they walk in, then in the worst case Mike will be the last student through the door and in the best case Mike will be the first student through the door. So this search algorithm is in $O(n)$ and $\Omega(1)$. We call the latter "constant time."

- If we look back at the Phonebook Example, searching one page at a time is a linear algorithm. In the worst case, the entry we're looking for is on the last page of the phonebook, so this search algorithm is in $O(n)$. Comparatively, our "divide and conquer" approach, which boils down to binary search, is in $O(\log n)$.

4 Searching, Sorting, and Recursion (35:00–64:00)

- On the board are two arrays of integers covered by pieces of paper.
- Bring down a volunteer and ask him to find the value 3 in the top array.
- Edward looks behind pieces of paper "randomly" and finds 3 on the second try. His method is in $\Omega(1)$ and $O(n)$.
- How could we improve on this? Well, we could sort the array to begin with. In the Phonebook Example, this was already done for us. When we flipped to the middle and got to the M's, we knew that S for Smith would be to the right.
- The top array is clearly not sorted. But let's give Edward the assumption that the bottom array *is* sorted and ask him to find the number 50.
- Well because Edward apparently has supernatural powers, he found it on the second try once again. When we press him to find the number 52, we find that we can formalize his algorithm a little better. He checked the rightmost number and then the leftmost number to know whether the array was sorted highest to lowest or lowest to highest. Then he planned to iterate from left to right until he found the number. This algorithm is then still in $O(n)$.
- Let's write down linear search in pseudocode:

```
on input n:
  for each element i:
    if i == n:
      return true.
  return false.
```

  In plain English, just look at every element and see if it's the right one. If you never find it, return false.

- Binary search is a direct application of the "divide and conquer" approach. In pseudocode, it looks like this:

```
on input array[0], ... , array[n - 1] and k:
  Let first = 0.
  Let last = n - 1.
```

```
While first <= last:
  Let middle = (first + last) / 2
  If k < array[middle], then let last = middle - 1
  Else if k < array[middle] then let first = middle + 1
  Else return true.
Return false.
```

This is just a written out version of what we were doing with the phonebook: look in the middle and go left or right. So in our bottom array, if we're looking for 50, we look in the middle, find the number 101 and then throw away the right half of the array since all those numbers will be larger than 101 and thus larger than 50. Then we focus on the remaining left half of the array and repeat the process. This algorithm is a considerable improvement on Edward's since as n increases, binary search will not take anywhere near as long as Edward's search.

- Incidentally, the bottom array consists of all the CS courses you'll be qualified to take after or before CS 50: 1, 50, 51, 61, 105, 121, 124, 171.
- To walk through the pseudocode, think of `first` as David's left finger and `last` as David's right finger. At the beginning, his left points to the leftmost number and his right points to the rightmost number. Then we set `middle` to be the middle number in the array. If the number we're looking for is less than this number, then we set `last` to be `middle - 1`, or in other words David's right points now to the number to the left of the middle. Our new array, then, is just the left half of the original array.
- When we say "repeat" in our pseudocode, we might actually be referring to a programming concept known as *recursion*, whereby a function invokes itself albeit on a smaller piece of the puzzle.
- Let's take a look at `sigma1.c` for a summation program which doesn't employ recursion:

```
/**************************************************************************
 * sigma1.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Adds the numbers 1 through n.
 *
 * Demonstrates iteration.
 **************************************************************************/

#include <cs50.h>
#include <stdio.h>
```

```
// prototype
int sigma(int);


int
main(int argc, char *argv[])
{
    // ask user for a positive int
    int n;
    do
    {
        printf("Positive integer please: ");
        n = GetInt();
    }
    while (n < 1);

    // compute sum of 1 through n
    int answer = sigma(n);

    // report answer
    printf("%d\n", answer);
}


/*
 * int
 * sigma(int m)
 *
 * Returns sum of 1 through m; returns 0 if m is not positive.
 */

int
sigma(int m)
{
    // avoid risk of infinite loop
    if (m < 1)
        return 0;

    // return sum of 1 through m
    int sum = 0;
    for (int i = 1; i <= m; i++)
        sum += i;
    return sum;
}
```

This program simply implements the sum of numbers 0 through n, where n is a number input by the user. In mathematical notation: $\sum_{i=0}^{n} i$. Notice that we're returning 0 if the number provided to `sigma` is less than 1. Although we could print out a message to indicate this, the program itself can't understand a *side effect* like this. It needs a return value. In this case, the 0 represents a *sentinel* which our `main` method can respond to as necessary.

- Notice also at the top that our prototype declaration of the `sigma` function is necessary because we call `sigma` from `main` and yet define `sigma` only at the bottom of the program. We don't need to name the arguments provided to `sigma`, only to give their type—in this case, `int`.
- Notice that we declare `sum` outside the scope of the loop so that later on we can access its value and return it.
- If we compile and run `sigma1`, we see that it works perfectly correctly. But, interestingly, we can implement the same functionality in an entirely different way using recursion. Check out `sigma2.c`:

```
/**************************************************************************
 * sigma2.c
 *
 * Computer Science 50
 * David J. Malan
 *
 * Adds the numbers 1 through n.
 *
 * Demonstrates recursion.
 **************************************************************************/

#include <cs50.h>
#include <stdio.h>


// prototype
int sigma(int);


int
main(int argc, char *argv[])
{
    // ask user for a positive int
    int n;
    do
    {
        printf("Positive integer please: ");
```

```
        n = GetInt();
    }
    while (n < 1);

    // compute sum of 1 through n
    int answer = sigma(n);

    // report answer
    printf("%d\n", answer);
}


/*
 * int
 * sigma(int m)
 *
 * Returns sum of 1 through m; returns 0 if m is not positive.
 */

int
sigma(int m)
{
    // base case
    if (m <= 0)
        return 0;

    // recursive case
    else
        return (m + sigma(m-1));
}
```

Notice that our `main` method is identical to that of `sigma1.c`. Of course, we don't want to induce an infinite loop by implementing a function which calls itself over and over again indefinitely. That's what the *base case* is for—to provide an exit. The rest of the magic takes place in the *recursive case*, in which `sigma` is called again. Think about it: if we want the sum of m, we can reduce that to be the sum of m and all the numbers less than $m-1$. That sum, then, is $m-1$ plus all the numbers less than $m-2$. So each time we call `sigma`, we're passing it one number less than our current number. Only once the number we pass to `sigma` is less than or equal to 0 do the functions start returning and the answer starts bubbling up.

- What's the catch? Well, if we start putting in very large numbers as our input, `sigma1` will return just fine, but `sigma2` will quit unexpectedly with a *segmentation fault*. This is essentially a memory error and, no, it's not okay to turn in programs that do this! When a program of yours does this,

a file called `core` will be created in your directory. This file will contain the contents of RAM when your program failed.

- In `sigma2`, every time the `sigma` function is called, a new frame, or a new chunk of memory, is pushed onto the stack. If we do this too many times, we'll run out of memory. This is when we get the infamous segmentation fault. We'll talk Wednesday about how to work around it!