Computer Science 50 — Week 3 Wednesday: September 22, 2010
Fall 2010 — Andrew Sellergren
Scribe Notes

Contents

1 Announcements and Demos (0:00–4:00)

- This is CS50.
- 0 new handouts.
- If you still need to change sections, please do so today. Either e-mail help@cs50.net or follow the instructions in the e-mail the bot sent you.
- Zynga will be here for recruiting purposes today at 6:30 PM in Maxwell Dworkin 119. Bring your resumes. There will be free food and a chance to win an HP Mini Notebook.
- Brian Kernighan, a former professor of CS50 now on sabbatical from Princeton, will be here Thursday at 3:30 PM for ice cream on the 2nd floor of Maxwell Dworkin and to give a talk on "The Changing Face of Programming" at 4:00 PM in Maxwell Dworkin G125. His first CS50 lecture included a demonstration of how to shave a beard with hedge clippers!
- Facebook will be here Monday at 12 PM in Maxwell Dworkin 119. Thomas Carriero, former CS50 TF, will be recruiting. Lunch will be served.

2 From Last Time (4:00–6:00)

- Searching through an unsorted array for a single integer proved time-consuming because we could do no better than brute force. In the worst case, brute force search takes n steps to search through an array of length n.
- Thankfully, we did a little better with a sorted array. Using binary search, we significantly reduced the number of steps it took to find a single integer.
- How do we go about sorting an array of numbers? What about, in the context of Facebook, a list of friends? We'll dive more into this today.

3 Debugging (6:00–25:00)

- Thus far, you have probably only used `printf` to debug your programs. And, of course, if you have a syntax error in your program, GCC will point it out, albeit somewhat cryptically. As your programs get more complicated, this kind of debugging becomes unwieldy.
- GDB, or GNU Debugger, allows you to step through your program line by line while it's executing. In this way, you can examine the state of the program in realtime, printing out variables and peeking at the stack as needed. You can also set breakpoints in GDB which allow you to pause your program's execution at a specific line so that you don't have to step through all the previous ones to get to it.

- To demonstrate the use of GDB, we'll examine `buggy3.c`. Recall this is the program that aimed to swap the values of two variables but failed to do so because of issues with scope: although the values were actually swapped in the function `swap`, as soon as that function returned, the variables took on their original values.

- When we run `make buggy3`, we're actually running `gcc` with a number of flags. `-lm`, `-lcs50`, and `-lcrypt` link in the `math.h`, `cs50.h`, and `crypt.h` libraries, respectively. `-Werror` instructs the compiler to treat warnings as errors. We know this is nitpicky, but it will force you to correct your mistakes, however small. `-ggdb` includes some additional bits in your program's binary that help GDB follow along while it executes.

- If we run `buggy3`, we confirm that the values of `x` and `y` aren't actually swapped. However, if we add a `printf` statement at the bottom of `swap`, we see that the local variables `a` and `b` have been swapped.

- Now, from the command line, we run `gdb buggy3` to start GDB. After the warranty and copyright information is printed, we are presented with a prompt that looks like this:

  ```
  (gdb)
  ```

  Here, if we type the command `run`, our program will execute just as it would outside of GDB and the message "Program exited normally." will be printed. This message indicates that `main` returned 0.

- To set a breakpoint at the beginning of the `main` function, we execute the following command from the GDB prompt.

  ```
  (gdb) break main
  ```

  This gives output that looks something like the following:

  ```
  Breakpoint 1 at 0x804842d: file buggy3.c, line 21.
  ```

  The `0x804842d` is a number in hexadecimal, which is a base system like decimal or binary, and represents a memory address. Line 21 is where `main` begins in our source code.

- Now if we type `run` at the prompt, we get the following:

  ```
  Starting program: /home/malan/src2/buggy3

  Breakpoint 1, main () at buggy3.c: 21
  21          int x = 1;
  ```

Our program has paused execution right before line 21. Line 21 will execute if we type the command `next` or `n`, for short. Once we've done so, we can print out the value of `x` like so:

```
(gdb) print x
$1 = 1
```

The `$1` allows us to refer back to variables we've already printed later in the program's execution.

- At this point, line 22, in which `y` is initialized, has not been executed yet. Let's print out `y` anyway:

  ```
  $2 = 3223540
  ```

  This is a strong reminder to initialize your variables before you use them! If we don't explicitly assign a value to `y`, we have no way of knowing what it contains.

- Executing `next` a few more times gives us the program's output commingled with GDB's:

  ```
  (gdb) next
  24      printf("x is %d\n", x);
  (gdb) next
  x is 1
  25      printf("y is %d\n", y);
  (gdb) next
  y is 2
  26      printf("Swapping...\n");
  (gdb) next
  Swapping...
  27      swap(x, y);
  ```

  At line 27, we're about to call the function `swap`. If we `next` again, we go straight to line 28 where "Swapped!" is printed. Then if we try to print `x` and `y`, they'll have the values 1 and 2, respectively.

- This exercise wasn't all that useful because we didn't get to see what was going on inside `swap`. If we wanted to do that, we could have typed `step` when we reached line 27. This tells GDB to step inside any functions that are called on the next line.

- Stepping into `swap` and executing the `list` command gives us the following:

```
(gdb) step
swap (a=1, b=2) at buggy3.c:41
41      int tmp = a;
(gdb) list
36  */
37
38 void
39 swap(int a, int b)
40 {
41      int tmp = a;
42      a = b;
43      b = tmp;
44 }
```

`list` shows us the lines of source code both above and below the one we're currently paused on.

- As we did before with `y`, let's print `tmp` before we've initialized it:

  ```
  (gdb) print tmp
  $1 = 0
  ```

  See what we mean about not knowing what an uninitialized variable will contain?

- The next lines of code will clobber the value of `a` with that of `b`. Before we do so, let's examine `tmp`, `a` and `b`:

  ```
  (gdb) next
  42      a = b;
  (gdb) print tmp
  $7 = 1
  (gdb) print a
  $8 = 1
  (gdb) print b
  $9 = 2
  ```

- Once we clobber `a` with `b` we can see that they both equal 2:

  ```
  (gdb) next
  43      b = tmp;
  (gdb) print a
  $10 = 2
  (gdb) print b
  $11 = 2
  ```

  Although this example is somewhat elementary, hopefully you can see how useful GDB will be as your programs get more and more complex.

- If we type `run` in the middle of the program's execution, we will be asked if we want to start from the beginning.

- Let's say we're paused while in the `swap` function but we forget exactly how we got there. Use the `backtrace` command:

  ```
  swap (a=1, b=2) at buggy3.c:41
  41      int tmp = a;
  (gdb) backtrace
  #0  swap (a=1, b=2) at buggy3.c:41
  #1  0x08048487 in main () at buggy3.c:27
  (gdb)
  ```

  `backtrace` shows us the contents of the stack or RAM. As you can see, there are two stack frames, one for `swap` and one for `main`.

- Question: can you only set breakpoints at `main`? No, we could've typed `break swap` to set a breakpoint on the `swap` function or we could even have typed `break 23` to set a breakpoint on line 23.

- Question: can you `break` and then `continue` again? Yes. For example, if you set a breakpoint in the middle of a loop, `continue` will stop on the next iteration of the loop where the next breakpoint is according to the logic of your program.

- Question: can you start execution of your program at different points? No.

4 Sorting (25:00–65:00)

4.1 Bubble Sort

- Although we as humans may have some intuition as to how to sort a list of numbers, we need to be able to translate that intuition into instructions that the computer can understand.

- 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.

- Our first attempt at sorting 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.

- Iterating through the array obviously requires a loop. But because the array most likely won't be sorted after walking through it once, we need to have a second outer loop that tells us to keep walking through the array as many times as necessary until it is sorted. How many times will that outer loop execute? Intuitively, we can reason that it will execute 8 times (the length of the array) because if the lowest number is in the last position in the array and we only swap it once on each iteration of the loop, it will take 8 iterations to make the 8 swaps that are necessary to put it in the correct position at the beginning of the array.

- On the fourth iteration of our outer loop, we make a single swap and see that the entire array is sorted. However, we only know this because we can see all 8 numbers at once. Because the computer can't see all 8 numbers at once, it doesn't know that the array is sorted, so it must keep iterating. If we make even one swap while iterating through the array, the computer assumes that we're not done sorting. Only when we iterate through the array and make no swaps will the computer know that the array is sorted.[1]

- How many steps does this algorithm involve? In the best case, the array is already sorted, so we iterate through it once, make no swaps, and we're done. We'll count that as 8 steps, one for each number in the array. In the worst case, it's going to take 64 steps since the outer loop will execute 8 times and each iteration of the loop takes 8 steps to walk through the array.[2]

- To generalize, this algorithm takes n steps in the best case and n^2 steps in the worst case, where n is the length of the array we're sorting. Although this doesn't seem that bad, imagine if n is not 8, but 10000. In that case, this algorithm might consume a lot more resources than we'd like it to.

- Because of the way numbers bubble up from one end of the array to the other, this algorithm is called bubble sort.

4.2 Selection Sort

- Beginning again with an unsorted array, we start walking from left to right, this time looking for the smallest number in the array. When we find the smallest number so far, we store its location in a temporary variable. Whenever we find a number that's smaller, we update the temporary variable to store the new location.

- When we reach the end of the array, we make a single swap: the smallest number to index 0 in the array. On the second pass through the array,

[1]This assumes that we don't have some very complicated conditions being checked which actually might enable us to stop iterating even if we've made a swap.

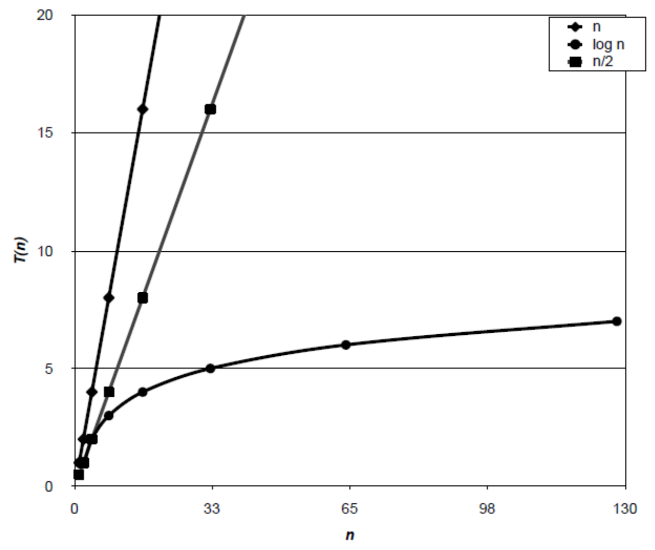
[2]Actually, if you're keeping track, we only need to make 7 swaps to move 1 from the end of the array to the beginning, but we have to iterate through the array once more and make no swaps in order to know that it is sorted.

we'll start walking from index 1 in the array since we already know that the number in index 0 is in the right place. This time when we find the smallest number in the array, we swap it into index 1. The third pass through the array will swap a number into index 2, and so on.

- Instead of swapping the smallest number with the leftmost number, we could pull the smallest number out of the array and then shift the other numbers to the right. However, this would be unnecessarily expensive, as swapping only takes 1 step and shifting would take more than 1 step.
- How many steps does this algorithm take? Although we made some optimizations whereby we started at index 1 on the second pass through the loop and at index 2 on the third pass through the loop, this algorithm still takes roughly n^2 steps in the worst case. To find the first smallest number, it took us n steps because we walked through the entire array once. To find the second smallest number, it took us $n-1$ steps because we started from index 1. So the whole algorithm will take $n+n-1+n-2\ldots$. This series sums to $\frac{n(n+1)}{2}$ and although that technically means the algorithm takes $\frac{1}{2}n^2+\frac{1}{2}n$ steps, we throw away all but the highest-order term (the one with the largest exponent) and all of the coefficients because as n gets very large, they have negligible effect on the result.
- What about the best-case scenario? In fact, it still takes n^2 steps because the computer has no way of knowing on any iteration through the array that the smallest number is already in the correct position.

4.3 Big O Notation and Runtime

- Computer scientists use what's called big O notation to denote the worst-case runtime of an algorithm. We say that both bubble sort and selection sort are in $O(n^2)$. To describe the best-case runtime, we refer to Ω and say that bubble sort is in $\Omega(n)$ while selection sort is in $\Omega(n^2)$. If the best-case and worst-case runtime are the same for an algorithm, we use Θ. We say, for example, that selection sort is in $\Theta(n^2)$.
- To see these sorting algorithms in action, check out this demo. Unfortunately, it doesn't work properly on Macs, so it's best to view it on a PC. In this demo, longer bars represent larger numbers. Even though swaps are being made pretty quickly and the longer bars are bubbling to the right, the demo takes a long time to complete. This gives you a pretty good idea that bubble sort is actually quite slow. Likewise, selection sort feels pretty slow although it seems slightly faster than bubble sort.
- Take a look at the graphs below of n versus $n/2$ versus $\log n$:



From these graphs, we can see that if we had 9 numbers in our array instead of 8, it would take us one additional step in the best-case scenario using bubble sort.

- On the first day of class, when we counted all of the students in Sanders Theater two at a time rather than one at a time, we were cutting the runtime in half. This is what the $n/2$ graph represents. With our final algorithm, we got half of the class to sit down on each iteration, meaning we were effectively cutting the problem in half with each step. This is an extremely compelling algorithm, as its runtime is $\log n$. As you can see, the graph has a very gradual slope, meaning that the number of steps it takes to complete increases only very slightly as the size of the problem increases.
- Unfortunately, we'll never be able to sort n numbers in $\log n$ time. This is because no matter how we sort, we're going to have to make at least n comparisons, that is, walking through the array at least once, in order to verify that it's sorted.

5 Recursion (65:00–71:00)

5.1 `sigma1.c`

- In general, if an algorithm repeats itself multiple times and only the size of the problem changes on each iteration, we can use *recursion* to implement it. A recursive function is one that calls itself. Of course, we'll need to make sure that at some point our program breaks out of this recursion lest that function call itself infinitely and we run out of memory.

- Take a look at `sigma1.c` which implements a non-recursive function to sum up the numbers 1 through n:

```
/****************************************************************************
 * 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(void)
{
    // 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);
}


/*
 * 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;
}
```

Here we use a `do while` loop to prompt the user for a positive integer and to keep prompting him if he doesn't provide one. In our `sigma` function, we do a sanity check to make sure the number it's been passed isn't less than 1 and then we iterate up to the number the user provided, summing along the way.

- 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.

5.2 `sigma2.c`

- `sigma2.c` solves the same summation problem as before, but does so using a recursive function:

```
/****************************************************************************
 * 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(void)
{
    // 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);
}


/*
 * 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));
}
```

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.

6 A Teaser (71:00–72:00)

- As a teaser for next time, check out this demo which allows you to compare sorting algorithms side by side. Try running selection sort and bubble sort against merge sort and see which one wins!