Computer Science 50 — Week 4 Monday: September 26, 2011
Fall 2011 — Andrew Sellergren
Scribe Notes

Contents

1 Announcements and Demos (0:00–1:00, 4:00–5:00)

- First on the docket is a receipt that shows the price for various items at a restaurant as \$1.48999, a clear indication of floating point errors in the cash register. Computer science in the real world!
- CS50 wifi is now available in Sanders! The SSID is cs50 and the password is 12345678. Yup.
- Another CS50 Lunch is this Friday, this time with an alumnus who now works at a venture capital firm called NEA. RSVP if you'd like to join!

2 Problem Set 3 (1:00–4:00)

- This week's problem set will task you with building The Game of Fifteen. Whereas for the previous problem sets, you started from scratch,[1] for this problem set, you will start with distribution code, a skeleton framework of files and functions. Our goal is to get you accustomed to designing larger programs and working with someone else's code.
- In the Hacker Edition, you are tasked with implementing the game as well as a solver mode. When asked for a tile to move, you should be able to enter `GOD` to cause the tiles to move automatically until the puzzle is solved.
- Also in Problem Set 3, you'll be asked to implement binary search and one of the sorting algorithms we've looked at so far, perhaps bubble sort or selection sort which are in $O(n^2)$. If you're taking on the Hacker Edition, you'll also need to implement a sorting algorithm, but one that runs in $O(n)$.

3 From Last Time (5:00–10:00)

- We looked at linear search as compared to the much-faster binary search.
- In terms of sorting algorithms, we looked at bubble sort, which works by comparing two adjacent values, swapping them if they are out of order, and stopping when no more swaps are made. We also looked at selection sort, which works by finding the smallest number and placing it in its correct position on each pass. We discovered that selection sort takes n^2 steps in both the best- and worst-case scenarios whereas bubble sort takes n^2 steps in the worst-case scenario, but only n steps in the best-case scenario.
- As this demo shows, there are many more sorting algorithms than the ones we looked at, most of which are much faster than bubble sort and

[1]Or from Scratch, lol!

selection sort.[2] Even Barack Obama knows that bubble sort isn't the best solution.

4 Debugging (10:00–26:00)

- 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 seemed to take on their original values. This bug was due to our passing variables *by value* to the `swap` function, rather than *by reference*. We saw this pictorially when we visualized the program's stack, which showed that `main`'s variables were in a completely different section of memory than `swap`'s variables.

- If we run `buggy3`, we can confirm that the values of `x` and `y` aren't actually swapped. Now, instead of running `buggy3` directly, let's run `gdb buggy3` which will run our program in the context of GDB and allow us to step through it. 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:

[2]Bogosort is still the best.

```
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 step past the lines of code that initialize `x` and `y`, we can print out the value of `y` like so:

  ```
  (gdb) print y
  $1 = 2
  ```

  As we expect, the value of `y` at this point is 2. The `$1` allows us to refer back to variables we've already printed later in the program's execution.

- 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.

- So far, this exercise hasn't been very useful because it only confirmed what we already know: the program isn't working as intended. If we want to know *why* it's not working as intended, we'll need to dig a little deeper. Let's type `run` and say yes when GDB asks us if we want to start the program over again. Once again, GDB will stop us at the beginning of `main`. Now let's try printing `x` before it's been initialized:

```
$6 = 3219444
```

This is a strong reminder to initialize your variables before you use them! If we don't explicitly assign a value to `x`, we have no way of knowing what it contains. More than likely, it will contain some garbage value, the remnants of whatever was in memory there previously.

- If we want to actually examine what's going on inside `swap`, we will type `step` when we reach line 27. This tells GDB to step inside any functions that are called on the next line. Stepping into `swap` gives us the following:

  ```
  (gdb) step
  swap (a=1, b=2) at buggy3.c:41
  41      int tmp = a;
  ```

  Once we get to line 44, we can print `a` and `b` to see that they contain the values 2 and 1, respectively. This confirms that they've actually been swapped. However, as soon as we step out of the `swap` program, we can print `x` and `y` to see that they retain their original values and we can also verify that `a` and `b` no longer exist if we try to print them. Although this example is somewhat elementary, hopefully you can see how useful GDB will be as your programs get more and more complex.

- Check out the GDB section on the Resources section of the course website. Admittedly there's a learning curve with GDB and it might be tempting to leave off using it till later in the course, but if you take the time to experiment with it now, it will pay huge dividends even for this week's problem set.

- Question: will GDB allow you to print something like an array? It will do its best. It should show you all of the members of an array, for example, with the possible exception of if you start passing that array around to different functions, in which case GDB might lose track of its length and have trouble printing it.

- As a sidenote, know that you can examine the contents of a `core` file by running `gdb <program> core` from the command line. Recall that we dumped a file like this earlier when we wrote a function that called itself infinitely and eventually ran out of memory for new stack frames.

5 Recursion and Merge Sort (24:00–60:00)

- A function which calls itself is said to be *recursive*. Unlike the recursive function we wrote earlier as a demonstration of a segmentation fault, most recursive functions have branches in their logic that dictate when the function should stop calling itself.

- Consider the following pseudocode which is actually an implementation of 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.
  ```

  If someone asked you how you would sort a list of items, you could actually push back on them and say that you would sort the left half and sort the right half, then merge the two sorted halves. In turn, this person could follow up and ask you how you would sort each half. In response,[3] you could again say that you would sort the left half and sort the right half, then merge the two sorted halves. As you can see, this algorithm, which is actually a description of merge sort, lends itself quite naturally to recursion. Although it feels like you're just being a jerk,[4] you are in fact answering the question. At some point in this back and forth, you'll get to a point where one half of the original list is actually only one item. At that point, that half of the list is already sorted, and your work is done.

- To unravel this concept of recursion, let's begin with a non-recursive function that sums 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);
  ```

[3]Because you're a jerk, perhaps?
[4]Perhaps because you are.

```
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;
}
```

Note that in our prototype for the `sigma` function, we don't need to specify a name for the argument, only a type. 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.c`, we see that it works perfectly correctly. Well, almost. What happens if we give it a number larger than 4 billion, the maximum that a 32-bit integer can store? The program seems to loop infinitely, although it might just be choking on the very large number.
- Interestingly, we can implement the same functionality in an entirely different way using recursion. After all, the sum of the numbers 1 through n is equivalent to n plus the sum of the numbers 1 through $n-1$. And the sum of the numbers 1 through $n-1$ is equivalent to $n-1$ plus the sum of the numbers 1 through $n-2$. And so on until we're considering the sum of the numbers 0 through 0, which is, of course, 0. This case which ends our recursion we'll call the *base case*. Take a look at how we utilize this approach in `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(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`. All that differs is `sigma`, which seems simple even when implemented with recursion. Without the comments, we have but 4 lines of code in `sigma`. As you can see, our base case is when `m` is less than or equal to 0. Note that we're making a decision here not to sum up negative numbers. In our recursive case, we call `sigma` and pass it one less than our current number. That's all there is to it!

- The function we wrote that called itself infinitely was an example of a recursive function without a base case.
- When we compile and run `sigma2.c`, we see that it works very well for small numbers, but a number like 1000000 still causes a segmentation fault. It seems that even though our function does not call itself infinitely, it calls itself enough times that it runs out of memory in which to place new stack frames. Recursion might present a problem in this case, but in the case of sorting, it should still be a reasonable solution. With our divide-and-conquer approach, even a list of size 1000000 will only require approximately 20 steps and thus 20 function calls.
- On stage we have 8 milk crates with the numbers 1 through 8. Joseph has volunteered to be our sorter as we follow the algorithm for merge sort:

  4 2 6 8 1 3 7 5

- First, we ask if the number of elements (8) is less than 2. Since it's not, we continue to the recursive case, which tells us to sort the left half. Second, we again ask if the number of elements (4) is less than 2. Since it's not, we continue to the recursive case, which tells us to the sort the left half. We continue in this vein until we're only considering 1 element, which is the left half of the left half of the left half of the original list. Because 1 is less than 2 and a list with only 1 element is inherently sorted, we finally return.

- Once we return, we reach the step "Sort right half." Again, this half has only 1 element, so it is already sorted. That leads us to the merge step. In this case, the left half is 4 and the right half is 2. Merging them is actually where the magic happens: we compare 4 and 2 and put 2 to the left of 4. Now our array looks as follows:

  ```
  2    4    6    8    1    3    7    5
  ```

  Finally, we've gotten through all the steps of the sort algorithm once. Now, to pick up where we left off, we need to sort the right half of our size-4 array. Because the number of elements (2) is not less than 2, we again divide and conquer. The left half, 6, and right half, 8, are both sorted because they contain only one element each, so we go to the merge step. 6 is already to the left of 8, so our merging is also already done. So the left half and the right half of our size-4 array are sorted, meaning we need to merge them. All the numbers are in their correct order already, so the left half of our original size-8 array is sorted:

  ```
  2    4    6    8    1    3    7    5
  ```

- When we repeat this algorithm for the right half of the original size-8 array, we get the following:

  ```
  2    4    6    8    1    3    5    7
  ```

  Now that our left half and right half of the entire array are sorted, we need to merge them. Here is where things get interesting. Although thus far we've been sorting in place, at this stage, we need a second, empty array of size 8 in which to put our final sorted numbers.

- In this final merge step, we will walk through each element of both halves, comparing them as we go. 2 is greater than 1, so we put 1 into the first position in our second, empty array:

  ```
  2    4    6    8         3    5    7

  1
  ```

  Now our right half has 3 as its leftmost element, so we compare it with the leftmost element of the left half, 2, and realize that we need to put 2 into the second array next:

```
      4     6     8           3     5     7

1     2
```

The left half now has 4 as its leftmost element, so we compare it with the leftmost element of the right half, 3, and choose the 3. If we continue in this vein, we put 4, then 5, then 6, then 7, then 8. And we're done!

- Now, how many steps did this take? Although we used the divide-and-conquer approach, the answer isn't just $\log n$. At the very least, we had to examine each element once, so our lower bound here is n steps. But it appears that every time we need to complete the merge step, we have to walk through both halves to do so. We can simplify this by saying it takes n steps per merge. How many merges are there? $\log n$ since there are $\log n$ divisions. So our merge sort algorithm is in $O(n \log n)$.

- 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 $n < 2$

  $T(n) = T(n/2) + T(n/2) + n$ 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 n!

- Ex: Suppose we want to find $T(16)$:

  $$T(16) = 2T(8) + 16$$
  $$T(8) = 2T(4) + 8$$
  $$T(4) = 2T(2) + 4$$
  $$T(2) = 2T(1) + 2$$
  $$T(1) = 0$$
  $$T(16) = 2(2(2(2(0 + 2) + 4) + 8) + 16) = 64$$

  We add 16 in the first step because it takes 16 steps to merge both lists of 8. 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.

- Have a listen to What different sorting algorithms sound like! They all sound like Pac-Man to me.