Section Notes 4

CS50 — Fall 2008

Week of October 12, 2008

Contents

1 Recursion

An "elegant" program is one that solves an interesting problem or performs a non-trivial computation in a way that is simple to read and easy to visualize. One of the most useful devices for creating elegant programs is *recursion*. A recursive procedure is one that invokes itself. The appealing design in Figure 1, known as a Sierpinski triangle, is produced by the following recipe:

- Start with an equilateral triangle pointing upward.
- Connect the midpoints of its sides to form four new equilateral triangles, three of which have the same orientation as the original.
- If these triangles are large enough to subdivide further, then apply this recipe to each of the three new upward-pointing triangles.

To be solvable recursively, a problem must be decomposable into subproblems that are just like the original problem, but a step closer to being solved. Any particular drawing of the Sierpinski triangle stops when the triangles become too small to represent effectively. Each recursive invocation of the recipe works on a triangle that is smaller than the one given to its parent. Since the recursion stops when the triangles get small enough, the overall procedure is sure to terminate.

Now take a numerical example. To give someone an intuitive idea of the meaning of the factorial function, you might write down something like this:

$$n! = n \times (n-1) \times \cdots \times 1$$

To someone who's used to mathematical notation, this equation says a lot in a simple way. The ellipsis in the middle means "continue the enclosing pattern in a way that connects the boundaries consistently." Unfortunately, the C programming language doesn't include the powerful $\cdots$ operator. So how should we implement a factorial function in C? We could define variables to count down from n to 1 and to accumulate and eventually return the product. But to understand that program requires reasoning about space (storage locations) and time (order of evaluation) that's more convoluted than the simple definition above deserves.

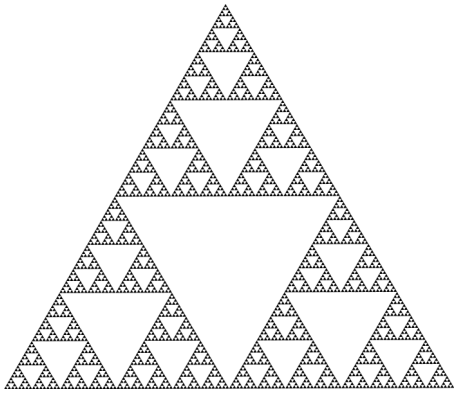

Figure 1: A Sierpinski triangle, a popular fractal named after Waclaw Sierpinski

Fortunately, factorial is a famously recursive function. Its definition can be rewritten like this:

$$n! = \begin{array}{l} 1 \\ n \times (n-1)! \end{array} \quad \begin{cases} \text{if } n \leq 1 \\ \text{otherwise} \end{cases}$$

This recursive definition can be expressed directly in C:

```
unsigned int
factorial(unsigned int n)
{
    if (n <= 1)
       return 1;
    else
       return n * factorial(n - 1);
}
```

Here there are no assignments, and the only loop is the one that's implicit in the recursion. In a theoretical sense, recursion is no more powerful than iteration. But for the some problems, it has a wonderful way of hiding unimportant details.

When the sorting algorithm called *quicksort* was invented (by C. A. R. Hoare), programming languages that supported recursion were not widely used, so it was initially described and implemented using only iterative constructs. Unfortunately, it is quite complicated to correctly state the quicksort algorithm without recursion. In fact, the original implementations were so complex and hard to follow that there was some debate over whether or not they really worked! Once the algorithm was restated in recursive form, however, the algorithm was quickly proved correct.

A function can be recursive even if it doesn't call itself directly. Consider the functions `even` and `odd` defined below. Can you see what they do?

```
bool even(unsigned int n);
bool  odd(unsigned int n);

bool
even(unsigned int n)
{
    return (n == 0) ? true : odd(n - 1);
}

bool
odd(unsigned int n)
{
    return (n == 0) ? false : even(n - 1);
}
```

We say that `even` and `odd` are *mutually recursive*. Each can call the other, so indirectly, it can call itself.

When the input to a recursive algorithm can be handled without recursion, we call it a *base case*. The inputs 0 and 1 lead to the base case of `factorial`, and similarly with input 0 to `even` and `odd`. Other cases, the general

cases that involve recursion, should do so in a way that leads towards a base case. Otherwise, the algorithm may not terminate.

Constructing a correct recursive program is like creating a mathematical proof by induction. You must clearly describe the outcome of an invocation of the recursive procedure, whether it is a side effect or a value returned. This description is analogous to the mathematical theorem. Then you code your base cases and make sure that they're consistent with the overall specification. And finally, you assume that recursive invocations will produce outcomes appropriate to their inputs, and you use those assumptions in solving each general case, which is analogous to the inductive cases in a proof.

As a final recursive example for now, suppose we didn't have `printf` and we needed to print a `long` integer value. Here's a nice way to do that recursively:

```
void
putlong(long i)
{
    if (i < 0)
    {
        putchar('-');
        putlong(-i);
    }
    else
    {
        long q = i / 10;
        long r = i % 10;

        if (q > 0)
            putlong(q);

        putchar('0' + r);
    }
}
```

This function has one base case, but two recursive cases. The first handles negative numbers by printing a minus sign and then recursively printing the magnitude of the argument. For non-negative inputs, we effectively shift the input value one decimal digit to the right by dividing it by 10. The resulting remainder represents the final digit to be printed, and the quotient represents those that must precede it. If the quotient `q` is zero, then there are no preceding digits to be printed, and we only need to print the character corresponding to the remainder `r`. But in the general case, we recursively process `q`, printing the digits of the prefix, and then we conclude with the final digit corresponding to `r`.

Both recursive calls make progress toward the base case. The first one reduces the problem of printing a negative number to that of printing a positive one. (Neither recursive call receives a negative argument.) The second recursive call makes progress because its argument is representable by a string that is a digit shorter than its caller's argument. And when the argument has been reduced to a single-digit number, there is no recursive call.

2 The Call Stack

You've already learned that when a C function is called, its parameters and its other local variables get their own storage. This space for values is independent of that for any other function. In fact, it's even independent of the storage for any other invocation of the *same* function. Now that you know about recursion, you can see why it's important to know that. During a recursive call to `putlong`, say, there may be many other active invocations of `putlong`. Each must have its own storage because the values of local variables will be different from invocation to invocation.

When a function is called, some space is set aside in memory specifically for that call of the function. Local variables are allocated in that space, and other important information is stashed there too, like the location in the code to return to after the call is complete. This space is called a *frame*. Every function call results in a new frame being created.

Of course, more than one function's frame may exist at one time. If `main` calls `make_move`, which calls `get_direction`, then all three of these functions have open frames.

Frames are arranged on top of each other in a *stack*.[1] The frame for the most recently called function is on top.

[1] Think of a spring-loaded stack of plates in a cafeteria or dining hall.

When a function (`get_direction`, say) is entered, a new frame for it is pushed on the top of the stack and becomes the active frame. When `get_direction` finishes, its frame is destroyed and the one for the function below (in this example, `make_move`) becomes active again.

This implementation of C functions calls supports recursion naturally. It's important that nothing special needs to be done to handle the call of a recursive function, because the compiler doesn't always know that a function is recursive. (Imagine that `even` and `odd` were defined in separate `.c` files. The compiler wouldn't know about their recursive relationship when it compiles each of those source files.)

To help you visualize how the call stack works, Figure 2 shows what happens when `main` calls `factorial(5)`. Reading downward in the picture, you see the stack at successive points in time, as it first grows to the right, and then

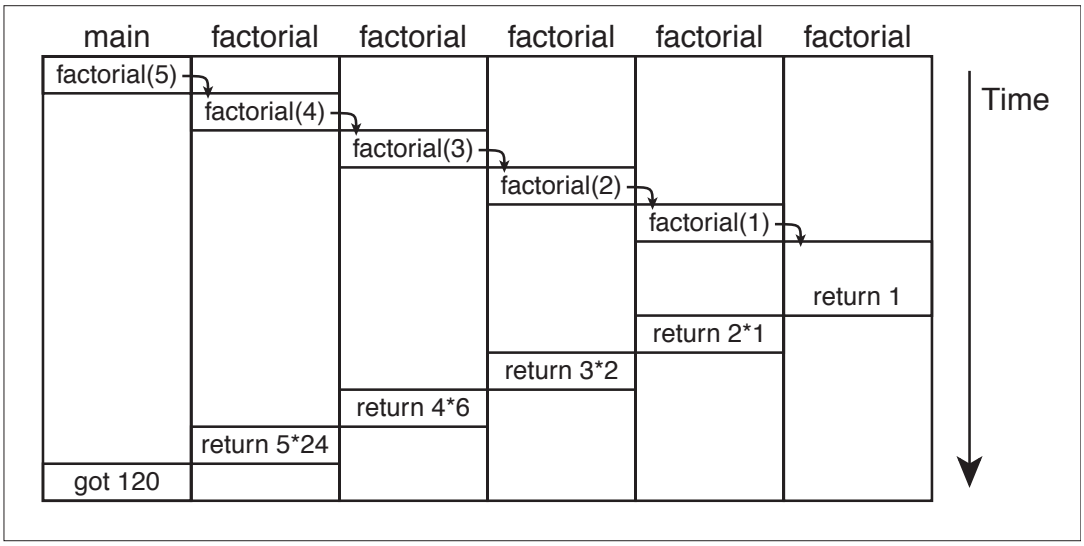

Figure 2: Call stack of `factorial(5)`

shrinks back to the left. When `factorial(5)` is called inside `main`, the frame for `main` is already on the stack, and a new frame is pushed on top of it, frame 2. In frame 2, the value of n is 5. Since 5 is greater than 1, `factorial` calls itself with argument $5 - 1$, creating frame 3, in which the value of n is 4.

The recursive calls continue until the call to `factorial(1)` is made from frame 5. The value of n in frame 6 is 1, so the base case applies and the function returns 1 without making a recursive call. That return removes frame 6 and reactivates frame 5. The returned value 1 is multiplied by the value of n in frame 5, and the process repeats, with successive frames being erased, and successive result values being passed back to callers. Eventually, frame 2 is popped off of the call stack, and the result 120 is returned to `main`.

3 Computational Complexity

Computer scientists analyze the *complexity* of algorithms to better understand the resources they require, both in terms of time required to produce a result and space required for the storage of data. The time complexity of an algorithm is not a description of the exact number of operations needed to complete the algorithm. Complexity deals with how fast this number grows as we increase the size of the input to some arbitrarily large value. This lets us know how well the algorithm "scales" with larger problems. In this course, we usually consider the worst-case complexity of an algorithm, on the assumption that we could run into worst-case inputs in normal use. In lecture, you saw that there is also notation for discussing the expected- and best-case complexity (Θ and Ω), but we will leave those for a future course in algorithms. In CS 50, when we ask for the time behavior of an algorithm, assume that we are asking for the worst case, unless we state otherwise.

By input size, we mean whatever makes sense with respect to the algorithm; if the algorithm operates on bytes, then the input size is the number of bytes. If the algorithm operates on a string, then we can use the string length for the input size. In any case, we'll use n to represent the size of the input.

The number of atomic operations an algorithm performs will be some function of this input size, or $f(n)$. Presumably, these operations take time that is independent of n, and so the time required for the algorithm to run will be proportional to $f(n)$.

3.1 Complexity Classification

Often, we care more about the *order* of function $f(n)$ than its particular structure, because we aren't using complexity measures to predict the performance of one algorithm, but to compare the performance of alternative algorithms as n grows arbitrarily large. So for example, if we know that an algorithm needs time proportional to $f(n) = 2n^2 + 3n$, we say that it runs in "order n^2 time", written $O(n^2)$, because it's the n^2 term in $f(n)$ that characterizes the algorithm's performance for large n. The notation $O(n^2)$ stands for the class of worst-case complexity measures whose growth is proportional to n^2 when n is large.

We can say formally what it means for a function, $f(n)$ to be in the class $O(g(n))$ like this:

> $f(n) \in O(g(n))$ if and only if there are constants $c > 0$ and $n_0 \geq 1$, such that
>
> $$f(n) \leq cg(n)$$
>
> for all $n > n_0$.

In English: "a function $f(n)$ is in the class of functions of order $g(n)$ if and only if $cg(n)$ eventually surpasses $f(n)$ when n becomes large enough.

In practice, we tend to run into a just a few "big-O" classes:

1. $O(1)$ — constant
2. $O(\log n)$ — logarithmic[2]
3. $O(n)$ — linear
4. $O(n^c)$ — polynomial[3]
5. $O(c^n)$ — exponential
6. $O(n!)$ — factorial

Any linear function eventually (for large enough n) surpasses any logarithmic function; any exponential function eventually becomes greater than a polynomial function, and so on.

If we add some rules for combining big-O functions, we can express a wider range of orders:

$$O(f + g) = O(max(f, g)) = max(O(f), O(g))$$

$$O(f * g) = O(f) * O(g)$$

That is, if you add the order of two functions, the one with the larger order wins. If you multiply the order of two functions, you get a new order, which is their product. So you get an $O(n^2)$ function if you call a function that is $O(n)$ inside a loop that is otherwise $O(n)$. And it will "grow faster" than a function of order $n \log n$, because the $O(\log n)$ class is below the $O(n)$ class (and the common $O(n)$ doesn't differentiate the two).

We always throw out "constant factors"; you will never see an algorithm described as $O(3n)$, because this is the same as $O(n)$, and the latter notation is preferred.

3.2 Determining efficiency

More sophisticated techniques will be explored in later computer science classes, but for now, it is sufficient to give an approximation of how many operations need to be performed. Look at two quick examples:

Example 1:

```
for (i = 0; i < n; i++)
{
    // loop body that runs in constant time
}
```

[2] A logarithm is the inverse of an exponential. That is, if $2^a = n$, then $\log n = a$. By convention, in computer science $\log$ has a base of 2, though if we were to use a different base it would change the number by only a constant factor.

[3] Anything more efficient than $O(n^c)$ is also called polynomial.

Example 2:

```
for (i = 0; i < n; i++)
{
    for (j = 0; j < n; j++)
    {
        // loop body that runs in constant time
    }
}
```

The first example runs in $O(n)$ time. It goes through one round of the `for` loop for each n. Example 2 runs in time $O(n^2)$, because the outer loop makes n rounds, each of which runs the inner loop, which runs through n iterations. Therefore the total time is $n * n$, or n^2.

3.3 Efficiency of recursive algorithms

Computing the efficiency of a recursive algorithm can be slightly more complicated than for iterative algorithms. For some algorithms, the analysis can be quite tricky and involve mathematics beyond the prerequisites of this course. Here are some general rules however, that should be helpful in finding the order of an algorithm:

A searching algorithm that reduces its data set by a constant amount (unrelated to n) and then calls itself recursively has $O(n)$ time behavior. Examples of an algorithm with this order include linear search, and searching through a linked list (which we will cover soon in this course).

A searching algorithm that divides its data set in half and then operates recursively on only one half of the data set complexity $O(\log n)$. Examples of an algorithm with this order are binary search, or searching through a balanced binary tree (also coming up!).

An algorithm that divides its data set in half, operates recursively on both halves, and then combines the two halves in an $O(n)$ operation, runs in $O(n \log n)$ time. Quicksort and merge sort (see the next section) are examples of this.

An algorithm that chooses an element in the data set and then performs some function of that element for each item in the rest of the data set, (such as comparing it to all the other elements in the set) before recursing (or looping) has $O(n^2)$ complexity. Selection sort and bubble sort (see below) are examples, although they're usually expressed iteratively rather than recursively.

4 Sorting Algorithms

Selection sort. The selection sort algorithm is quite intuitive. While it's working the data is divided into a sorted part and an unsorted part. The algorithm simply takes the smallest value remaining to be sorted and appends it to the sorted part.

1. Scan the unsorted part of the data to select the smallest value.
2. Switch the smallest value with the first value in the unsorted part of the data.
3. Repeat from 1 if there are still unsorted items.

Here is an example (the `|` divides the sorted region from the unsorted):

```
| 3 5 2 6 4     ( 2 is the smallest, swap with 3)
2 | 5 3 6 4     ( 3 is the smallest, swap with 5)
2 3 | 5 6 4     ( 4 is the smallest, swap with 5)
2 3 4 | 6 5     ( 5 is the smallest, swap with 6)
2 3 4 5 | 6     ( 6 is the only value left, done)
```

The selection sort algorithm takes $O(n^2)$ time in the best, worst, and expected cases.

Bubble sort. Because this method swaps adjacent pairs of items, values tend to float up the list like bubbles until they reach their places in order.

1. While stepping through the data, if two adjacent values are not in sorted order, then swap them.
2. After a full scan of the list, repeat from step 1 if any changes have been made.

For example:

```
Original data:  3 5 2 6 4
First pass:     3 2 5 4 6       swapped (5,2), swapped (6,4)
Second pass:    2 3 4 5 6       swapped (3,2), swapped (5,4)
Third pass:     2 3 4 5 6       no swaps
```

Note that the last pass is necessary, even though it is identical to the previous line, because it is on this pass that the list is guaranteed to be sorted and the algorithm to be complete.

Bubble sort takes O(n^2) time in worst and expected cases. It is `O(n)` in the best case, which occurs when the list is already sorted. In this case, there will be no swaps on the first pass through the list, so the algorithm will have completed after only n comparisons.

Insertion sort. This is like selection sort in that it scans repeatedly to find then next unsorted value to put in place. But instead of swapping two elements at a time, it shifts the unsorted part of the array.

1. Scan the unsorted part of the data to select the smallest value.
2. Insert the smallest value in the first position of the unsorted part of the data, shifting elements to the right as necessary.
3. Repeat until there is no unsorted data.

Here is an example:

```
| 3 5 2 6 4     ( 2 is the smallest, insert, shift 3, 5)
2 | 3 5 6 4     ( 3 is the smallest, insert, no shift)
2 3 | 5 6 4     ( 4 is the smallest, insert, shift 5, 6)
2 3 4 | 5 6     ( 5 is the smallest, insert, no shift)
2 3 4 5 | 6     ( 6 is the only value left, done)
```

This algorithm takes O(n^2) time in the worst and expected cases.

Merge sort. This is a naturally recursive algorithm for sorting. The idea is to divide the input list evenly in two and sort the two halves recursively. (If the input list doesn't have at least two elements, there's nothing to do. That's the base case of the recursion.) The two sorted sublists can be merged in $O(n)$ time, by a simple algorithm: Remove the smaller of the numbers at the heads of the two lists and append it to the merged list, repeating until both of the sublists are used up.

Here is one implementation of merge sort that uses an array to hold the list items and identifies the sublists using pairs of integer indexes into the array: `start` indexes the first item of the sequence to be sorted, and `end` is the index one *beyond* that sequence.

```
mergesort(int array[], int start, int end)
{
    int length = end - start;

    if (length > 1)
    {
        int middle = start + length/2;

        mergesort(array, start, middle);
        mergesort(array, middle, end);

        // merge the two subarrays in place
        merge(array, start, middle, middle, end);
    }
}
```

Exercise for the reader: Write the function `merge` recursively without using an auxiliary array.

Merge sort requires $O(n \log n)$ time in all cases. Since we divide each set to be sorted in half at each level of recursion, there will be $\log n$ levels, since $n/2^{\log n}$ is 1. At each level, a total of n comparisons must be made in order to merge subarrays. Hence, $O(n \log n)$.

5 Debugging with `gdb`

A debugger is an application that is able to run your program in a carefully controlled environment, so that you can analyze what's going wrong when it misbehaves. GDB, the GNU debugger, allows you to pretend that the computer is actually reading and interpreting your source code in order to execute it. You can stop and start the program at arbitrary code locations that you designate. You can look at the values of variables and even change them in the middle of execution. If your program produces a run-time error like a "segmentation fault", GDB allows you to inspect its state before it exits, so that you can try to find out where the error occurred. Even if you have run your program outside of the debugger, and it has dumped its state in the form of a `core` file, GDB is able to read that `core` file for you to help determine where the program died, and what the values of your variables were at the time. It's a remarkable tool, well worth learning to use.

A general strategy for running GDB is to start the debugger and set "breakpoints" in your code where you'd like to take control and observe the program's behavior in slow motion and great detail. Then you tell GDB to start the program with the desired command-line arguments. When it reaches one of your breakpoints, GDB will stop it and allow you to inspect variables and step forward, either by continuing to the next breakpoint, or by executing one statement at a time. You can choose either to step just inside the next function call and inspect its parameters, or you can let whole function calls occur without interruption. When inside a function, you can choose to execute without stopping until the current activation of that function returns, at which point you can inspect its returned value, if any. When stopped inside a loop, you can tell GDB to continue at full speed until the loop exits, then stop again.

To prepare your program to be debugged with GDB, add the `-ggdb` option when compiling with `gcc`. For best results, be sure that no optimization options (such as `-O2`) are used at the same time as `-ggdb`.

There are a number of ways to start GDB:

- On the shell command line, run the command `gdb` ⟨*executable*⟩. That will start the debugger and it will prepare to run the named executable file. (Note that you should name the compiled program file, not your C source code file. The debugger should be able to find your source files if they're in the same directory as the executable.)

- If you add the name of a core dump file to the command line, as in `gdb` ⟨*executable*⟩ `core`, then instead of starting a fresh run of the program, GDB restores the conditions at the time core was dumped, so that you can look at the call stack and at the values of variables. You can't continue execution, the way you could at a normal breakpoint, but it can be very helpful to do a detailed post mortem examination.

- If your program is running outside of the debugger, and you know its process ID (obtained using the `ps` command), you can "attach" to the program and interrupt it with the command `gdb` ⟨*executable*⟩ ⟨*pid*⟩ where ⟨*pid*⟩ is the process ID. If your program was compiled for debugging, you can proceed to debug it just as though you had started it under GDB in the first place.

- If you invoke the command `gdbtui` instead of `gdb` in a terminal window, the debugger will take over the terminal in order to show you the current execution point in your source code. This can be much more efficient than having to list source code lines manually whenever the debugger stops the action.

- If you learn a bit about Emacs, you can use GDB inside the editor. As with `gdbtui` in the shell, you will automatically be shown the current location in your source files whenever the debugger has stopped the program. You'll be able to set new breakpoints from within the code file buffers, and you'll have the full power of Emacs at your disposal to grab statements and expressions that you want the debugger to evaluate for you. The resources list for Emacs on the course web site points to a "guided tour" document that includes a screen shot of the editor's debugger interface. It closely resembles commercial development environments.

The resources list for GDB on the course site includes both a tutorial, which gives the essential commands for using the debugger, and the full on-line manual.