

Lecture notes: February 10, 2017

Topics:

1. Recursion
2. Using gdb to inspect the behavior of a recursive function

(This material is adapted from the PA06 README)

Recursion

In a superficial sense, recursion is what happens whenever a function f calls itself, as in the "standard" factorial example:

```
int factorial(int n) {
    if (n == 0) return 1;
    else return n * factorial(n - 1);
}
```

It is better to think of recursion as a technique for solving problems. Many problems can be thought of using the following pattern:

1. Break the problem up into "smaller" version(s) of the same problem.
2. Solve the smaller problem(s) by calling the same function (we call this the *inductive* or *recursive case*)
3. Use the solutions to the smaller problem(s) to solve the original problem.

This seems like a process that doesn't end: to solve a big problem, we break it up into smaller versions of the problem -- but then we have to solve the smaller problem, which isn't any different! The key is that you can repeat this process, solving the smaller problems in the same way. At each step, you get smaller and smaller problems. Eventually, the problem is small enough that getting the answer is trivial. We call this the *base case*.

We can see this in the factorial example: rather than computing factorial of n , we realize that $n!$ is just $n * (n-1)!$ (Step 1: break the problem up into a smaller version of the same problem) -- so we can call $\text{factorial}(n - 1)$ (Step 2: solve the smaller problem by calling the same function). We can then multiply this by n to find $\text{factorial}(n)$ (Step 3: use the solution of the smaller problem to solve the original problem). We also see that the base case is simple: we already know what $0!$ is, so there is no need to "break it up" into a smaller problem -- we can just return 1.

(Note: you could also write factorial with a loop, and the loop version would probably be faster, so you might wonder why we need recursion. In some cases, recursion may be just an easier way to think of a problem than writing a loop. Some times recursion might even be much slower! But some times, recursion is the only way to effectively solve a problem.)

One way to think about how to correctly write a recursive function is to think *inductively*. We can assume that the recursive function already works, but only if the function is called on a smaller problem than what we're solving. We can then write the recursive function assuming that it already works. The only thing we have to make sure we do is write correct base cases -- we

need to make sure that for the smallest versions of the function, we compute the correct answer. (This sounds circular, but it works for the same reason that inductive proofs work).

We can see this strategy at work in the factorial code: when figuring out what `factorial(n)` should be, we can assume that `factorial(n-1)` already works, giving us $(n-1)!$ (even though we don't have a working factorial function yet). If we have $(n-1)!$, computing $n!$ is easy: just multiply by n . All that's left is to make sure that we have a working base case: that we just provide the answer for the "smallest" argument we will pass to `factorial`, in this case 0.

Divide-and-conquer Recursion

A very common pattern for recursive problems is divide-and-conquer recursion: to solve a problem on n pieces of data (e.g., an array of length n), we break the input up into two pieces, each with $n/2$ pieces of data (e.g., two arrays, each with half the elements), call the recursive function on these smaller pieces, then write some code to combine the results from those two functions into the final answer. The base case for this style of function is what to do when you have only 1 element.

Consider a toy example where we want to sum up all the values in an input array with n elements. Here, if we divide the array in two and sum those two sub-arrays, we can add the results to get the sum of the whole array. The base case is that the sum of an array with just one element is the value of that element:

```
int sum(int * arr, int nels) {
    if (nels == 1) return arr[0];

    int sum1 = sum(arr, nels/2);
    int sum2 = sum(&arr[nels/2], (nels + 1)/2);

    return sum1 + sum2
}
```

(The $(nels + 1)/2$ stuff is just a fancy way of dealing with arrays that have an odd number of elements, where `sum2` works over a slightly larger array than `sum1`. In integer division, `nels/2` is like computing $\text{floor}(n/2)$, and $(nels + 1)/2$ is like computing $\text{ceiling}(n/2)$. More generally, to compute $\text{ceiling}(a/b)$ you can do integer division: $(a + b - 1)/b$.)

Using gdb to help understand recursion

So what happens when we call a recursive function like `factorial`? Remember that each time we call a function, we push its *frame* on the stack, which maintains all of its local variables and arguments. So every time `factorial` calls itself, it pushes a new "copy" of `factorial` onto the stack, with its own copy of the argument n . This is the key to recursion: each call to the function has its own local variables, which lets us multiply everything together as the functions return up the stack.

Let's see this in action using `gdb`. Let's say we compile this code:

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

```

int foo(int n) {
    int retval = n;

    if (n == 0) return 1;

    retval = retval * foo(n - 1);

    return retval;
}

void main() {
    int x = foo(5);
    printf("foo(5) = %d\n", x);
}

```

and then run it through gdb:

```

> gdb a.out
(gdb) r
Starting program: /home/dynamo/b/milind/264/recgdbtest/a.out
foo(5) = 120

```

We see the result of the code. Let's set a breakpoint in foo so we can investigate what is going on. But we will be clever: rather than just setting a breakpoint in foo ("b foo"), we'll set a breakpoint in foo that will *only* trigger when n is 0:

```

(gdb) b foo if (n == 0)
Breakpoint 1 at 0x4004cf: file rectest.c, line 5.

```

Now when we run the program, execution will stop when foo is called with argument 0:

```

(gdb) r
Starting program: /home/dynamo/b/milind/264/recgdbtest/a.out

Breakpoint 1, foo (n=0) at rectest.c:5
5      int retval = n;

```

If we step forward one step, we can see the value of retval:

```

(gdb) n
7      if (n == 0) return 1;
(gdb) p retval
$1 = 0

```

So what does the program stack look like? gdb can show us the *calling context* or *call stack* of the program at this point: the chain of functions that had to be called to get us to this point. To do this, we use the command “backtrace” (which can also be written “bt” or “where”):

```
(gdb) bt
#0  foo (n=0) at rectest.c:7
#1  0x00000000004004ef in foo (n=1) at rectest.c:9
#2  0x00000000004004ef in foo (n=2) at rectest.c:9
#3  0x00000000004004ef in foo (n=3) at rectest.c:9
#4  0x00000000004004ef in foo (n=4) at rectest.c:9
#5  0x00000000004004ef in foo (n=5) at rectest.c:9
#6  0x000000000040050f in main () at rectest.c:15
```

We see here the full “history” of the execution up to this point: main called foo(5) at line 15, foo then called foo(4) at line 9, foo then called foo(3) at line 9, and so on. Each one of those lines represents a *stack frame* sitting on the stack, each with its own local variables.

We can even jump to one of those stack frames to see what’s going on:

```
(gdb) f 2
#2  0x00000000004004ef in foo (n=2) at rectest.c:9
9      retval = retval * foo(n - 1);
```

Which jumps to the stack frame numbered 2 (frame 0 is always the frame of the currently running function, frame 1 is the frame of the function that called the current function, etc.)

Now if we inspect the value of `retval`, we’ll see that it has the value from a different version of `foo`:

```
(gdb) p retval
$2 = 2
```

And if we jump back to frame 0, we can see the “current” version of `foo`’s value of `retval` again:

```
(gdb) f 0
#0  foo (n=0) at rectest.c:7
7      if (n == 0) return 1;
(gdb) p retval
$3 = 0
```