CS50 Recursion

Overview

Recursive solutions to problems are typically contrasted with iterative ones. In a **recursive solution**, a function (or a set of functions) repeatedly invokes slightly modified instances of itself, with each subsequent instance tending closer and closer to a base case. In the meantime, the intermediate calls are all left waiting, having "passed the buck" to a downstream call to give it the answer it needs. Recursive procedures, when contrasted with iterative ones, can sometimes lead to incredibly efficient, elegant, and, some might even say, beautiful solutions.

Key Terms

- recursive solution
- iterative solution
- base case
- recursive case
- call stack
- active frame

Recursion versus Iteration

Recursive solutions can often replace clunkier iterative ones. One great example of this is with programs that calculate the factorial of a number. Remember that "n factorial," or `n!`, simply represents the product of all integers less than and including `n`. So `3!` would be six (`3 * 2 * 1`). Consider the two approaches on the right for implementing a function to find the factorial of an integer. The first implementation looks familiar. This is known as an **iterative solution** as we are iterating through a loop, substituting in different values for `i`. In this solution, we have declared two variables (`product` and `i`), while in the recursive solution, we have not declared any. Also, note that `recurse()` is fewer lines shorter than `iterate()`.

```
int iterate(input)
{
    int product = 1;
    for(int i = input; i > 0; i--)
    {
        product *= i;
    }
    return product;
}
```

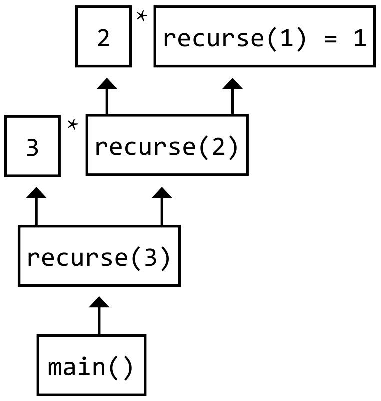
```
int recurse(input)
{
    if(input == 1)
    {
        return 1;
    }
    return input * recurse(input - 1);
}
```

Implementation

Recursive solutions to problems are made up of two parts: the base case and the recursive case. The **base case** is what allows us to break out of an infinite loop. Without a base case our program would continue to run until it no longer had the space to do so and resulted in a segmentation fault. In our example, the base case is when input == 1. The **recursive case** is where the function invokes itself. This appears in the last line of `recurse()`, where recurse is called again. In this way, recurse repeatedly calls itself until `1` is the value being passed into the function.

Call Stack Representation

When a recursive function (or any function for this matter) is called, it creates a new frame on the stack. Every subsequent function called within `main()` is created on top of the previous frame. This stack, where all of our function calls exist, is called the **call stack**. This means that the function at the top of the stack is the most recently called function. We call this the **active frame**. Say we pass in 3 as the input for `recurse()`, then `main()` will call `recurse(3)`, which will call `recurse(2)`, and so on. This process will continue to occur until the base case is met. Once this happens, that return value trickles down and is plugged back into the function calls left open in the call stack. In our example, 1 would be plugged into `recurse(1)`, destroying this frame in the call stack and leaving `recurse(2)` as the active frame. The number 2 would be passed into `recurse(2)` in the same way and so on until `recurse(3)` returned 6 and passed that back to `main()`. At this point, `main()` would be the only function left in the call stack, since all the other calls to `recurse()` would have been destroyed after returning a value.



Note that in our sample iterative solution above, there would only be one function called, `iterate()`. So in some ways iterative solutions are simpler than recursive ones. And since iterative solutions can usually solve the same types of problems as recursive ones, there will almost never be a real world problem that requires us to use recursion as a means to solve it. Rather, recursion can be used to make our code more elegant and efficient.