Recursion

Functions

Mike Burrell

March 28, 2024

Readings

- https://www.youtube.com/watch?v=aCPkszeKRa4
- https://www.youtube.com/watch?v=k0bb7UYy0pY

Goals for this set of slides

- Trace through a recursive function
- Structure a function recursively
- Understand the call stack better

Recursion

- Recursion is a function that calls itself
- Recursion is also a way of breaking down a problem into smaller problems

Recursion

- Recursion is a function that calls itself
- Recursion is also a way of breaking down a problem into smaller problems
- The smaller problems are still problems of the same form

Recursion in math

- Recursive definitions show up in mathematics a lot
- Defining something recursively sometimes gives a simpler way to explain/define it
- It can also make a good first step to writing a function recursively

Recursion in math

- Recursive definitions show up in mathematics a lot
- Defining something recursively sometimes gives a simpler way to explain/define it
- It can also make a good first step to writing a function recursively
- Example (factorial):

$$n! = \begin{cases} 1 & \text{, if } n \leq 1 \\ n \times (n-1)! & \text{, otherwise} \end{cases}$$

Factorial example in code

$$n! = \begin{cases} 1 & \text{, if } n \leq 1 \\ n \times (n-1)! & \text{, otherwise} \end{cases}$$

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

Algebraic tracing

- If a recursive function has no side-effects, it can be traced through algebraically

Algebraic tracing

- If a recursive function has no side-effects, it can be traced through algebraically

$$n! = \begin{cases} 1 & \text{, if } n \leq 1 \\ n \times (n-1)! & \text{, otherwise} \end{cases}$$

$$3!$$

Algebraic tracing

- If a recursive function has no side-effects, it can be traced through algebraically

$$n! = \begin{cases} 1 & \text{, if } n \leq 1 \\ n \times (n-1)! & \text{, otherwise} \end{cases}$$

$$\begin{aligned} & 3! \\ = \; & 3 \times (3-1)! \end{aligned}$$

Algebraic tracing

- If a recursive function has no side-effects, it can be traced through algebraically

$$n! = \begin{cases} 1 & \text{, if } n \leq 1 \\ n \times (n-1)! & \text{, otherwise} \end{cases}$$

$$\begin{aligned} & 3! \\ = \; & 3 \times (3-1)! \\ = \; & 3 \times 2! \end{aligned}$$

Algebraic tracing

- If a recursive function has no side-effects, it can be traced through algebraically

$$n! = \begin{cases} 1 & \text{, if } n \leq 1 \\ n \times (n-1)! & \text{, otherwise} \end{cases}$$

$$\begin{aligned} & 3! \\ = \; & 3 \times (3-1)! \\ = \; & 3 \times 2! \\ = \; & 3 \times 2 \times (2-1)! \end{aligned}$$

Algebraic tracing

- If a recursive function has no side-effects, it can be traced through algebraically

$$n! = \begin{cases} 1 & \text{, if } n \leq 1 \\ n \times (n-1)! & \text{, otherwise} \end{cases}$$

$$\begin{aligned} & 3! \\ = \; & 3 \times (3-1)! \\ = \; & 3 \times 2! \\ = \; & 3 \times 2 \times (2-1)! \\ = \; & 3 \times 2 \times 1! \end{aligned}$$

Algebraic tracing

- If a recursive function has no side-effects, it can be traced through algebraically

$$n! = \begin{cases} 1 & \text{, if } n \leq 1 \\ n \times (n-1)! & \text{, otherwise} \end{cases}$$

$$\begin{aligned} & 3! \\ = \; & 3 \times (3-1)! \\ = \; & 3 \times 2! \\ = \; & 3 \times 2 \times (2-1)! \\ = \; & 3 \times 2 \times 1! \\ = \; & 3 \times 2 \times 1 \end{aligned}$$

Call stack tracing

- Of course your computer isn't actually evaluating anything algebraically
- If a recursive function has side-effects, it can't be traced easily using algebra anyway
- How the computer is actually evaluating a recursive function is through regular call stack mechanics
  - No slides for this. Let's see it on the whiteboard

Base cases and inductive cases

- Every recursive function needs to finish recursing at some point
  - Infinite recursion is commonly not possible because it will cause a stack overflow
  - Even large (but finite) recursions can cause a stack overflow
- Generally the recursive functions you write will always have `if` statements at the top to distinguish between *base cases* and *inductive cases*

Inductive case

- An *inductive case* is also called a *recursive case*
- It is when we keep recursing (keep calling the same function)
- The most important rule is that, when recursing, the value we're recursing over *must become smaller*
  - For integers, this usually means tending towards zero
  - For arrays, this usually means having smaller elements
  - Other data structures will have other notions of becoming smaller

Inductive case example

Imagine we are writing a recursive function to define exponentiation over $\mathcal{Z}^+$.

$$x^y = \underbrace{x \cdot x \cdot \ldots \cdot x}_{y}$$

Inductive case example

Imagine we are writing a recursive function to define exponentiation over $\mathcal{Z}^+$.

$$x^y = \underbrace{x \cdot x \cdot \ldots \cdot x}_{y}$$

We can rewrite this recursively (inductively) as:

$$x^y = x \cdot x^{y-1}$$

Inductive case example

Imagine we are writing a recursive function to define exponentiation over $\mathcal{Z}^+$.

$$x^y = \underbrace{x \cdot x \cdot \ldots \cdot x}_{y}$$

We can rewrite this recursively (inductively) as:

$$x^y = x \cdot x^{y-1}$$

Note that:

- Our definition is now recursive (x^y is defined in terms of itself)
- One of our values is becoming smaller (y is getting closer to zero)

Inductive case

- Thinking of the inductive case first may help you figure out the general structure of the recursion
  - The hardest part of recursion is typically identifying which value is becoming smaller and how it is becoming smaller
- Some functions may have multiple inductive cases
- But the inductive case is not enough. . .

Base case

- A base case is when we stop recursing
- It's identified by an `if` statement that will identify that the value we're recursing over is now very small

Base case example

Imagine we are writing a recursive function to define exponentiation over $\mathcal{Z}^+$.

$$x^y = \underbrace{x \cdot x \cdot \ldots \cdot x}_{y}$$

$$x^y = x \cdot x^{y-1}$$

Our y value is the value we're recursing over (the value which is decreasing). When does y get too small for this recursive property to hold?

Base case example

Imagine we are writing a recursive function to define exponentiation over $\mathcal{Z}^+$.

$$x^y = \underbrace{x \cdot x \cdot \ldots \cdot x}_{y}$$

$$x^y = x \cdot x^{y-1}$$

Our y value is the value we're recursing over (the value which is decreasing). When does y get too small for this recursive property to hold?

$$x^0 = 1$$

We need to make our *base case* where $y = 0$.

Putting it together

So:

$$
\begin{aligned}
x^0 &= 1 \\
x^y &= x \cdot x^{y-1}
\end{aligned}
$$

Translated into code:

```
long exponent(long x, long y)
{
    if (y == 0)    return 1;
    else           return x * exponent(x, y - 1);
}
```

Exercises

1. Fibonacci number
2. Finding a square root using a binary search
3. Determining if a string is a palindrome
   - We will use the `c_str()` method to convert a `string` object into a NUL-terminated `char*`

Conclusion

We learned:

- How to trace through recursive functions in two ways
- How recursive functions are structured