CSC212

# Data Structure

## - Section FG

# Lecture 14

# Reasoning about Recursion

Instructor: Feng HU

Department of Computer Science

City College of New York

# Outline of This Lecture

- Recursive Thinking: General Form
  - recursive calls and stopping cases
- Infinite Recursion
  - runs forever
- One Level Recursion
  - guarantees to have no infinite recursion
- How to Ensure No Infinite Recursion
  - if a function has multi level recursion
- Inductive Reasoning about Correctness
  - using mathematical induction principle

# Recursive Thinking: General Form

- Recursive Calls
  - Suppose a problem has one or more cases in which some of the subtasks are simpler versions of the original problem. These subtasks can be solved by recursive calls
- Stopping Cases /Base Cases
  - A function that makes recursive calls must have one or more cases in which the entire computation is fulfilled without recursion. These cases are called stopping cases or base cases

# Infinite Recursion

- In all our examples, the series of recursive calls eventually reached a *stopping case*, i.e. a call that did not involve further recursion
- If every recursive call produce another recursive call, then the recursion is an *infinite recursion* that will, in theory, run forever.
- Can you write one?

# Example: power (x,n) = $x^n$

- Rules:
  - power(3.0,2) = $3.0^2$ = 9.0
  - power(4.0, 3) = $4.0^3$ = 64.0
  - power(x, 0) = $x^0$ = 1 if x != 0
  - $x^{-n} = 1/ x^n$ where x<>0, n > 0
    - power(3.0, -2) = $3.0^{-2} = 1/3.0^2 = 1/9$
  - $0^n$
    - = 0 if n > 0
    - invalid if n<=0 (and x == 0)

# ipower(x, n): Infinite Recursion

Computes powers of the form $x^n$

```
double ipower(double x, int n)
// Library facilities used: cassert
   {
   if (x == 0)
      assert(n > 0); //precondition

   if (n >= 0)
   {
      return ipower(x,n); // postcondition 1
   }
   else
   {
      return 1/ipower(x, -n); // postcondition 2
   }
}
```

# ipower(x, n): Infinite Recursion

Computes powers of the form $x^n$

```
double ipower(double x, int n)
// Library facilities used: cassert
   {
   if (x == 0)
      assert(n > 0); //precondition

   if (n >= 0)
   {
      return ipower(x,n); // need to be developed into a stopping case
   }
   else
   {
      return 1/ipower(x, -n); // recursive call
   }
}
```

```
double product =1;
for (int i = 1; i<=n; ++i)
        product *= x;
return product;
```

# power(x, n): One Level Recursion

Computes powers of the form $x^n$

```
double power(double x, int n)
// Library facilities used: cassert
   {
   double product; // The product of x with itself n times
   int count;
   if (x == 0) assert(n > 0);
   if (n >= 0)  // stopping case
   {
      product = 1;
      for (count = 1; count <= n; count++)
         product = product * x;
      return product;
   }
   else // recursive call
      return 1/power(x, -n);
}
```

# One Level Recursion

- First general technique for reasoning about recursion:
  - Suppose that every case is either a stopping case or it makes a recursive call that is a stopping case. Then the deepest recursive call is only one level deep, and no infinite recursion occurs.

# Multi-Level Recursion

- In general recursive calls don't stop a just one level deep – a recursive call does not need to reach a stopping case immediately.
- In the last lecture, we have showed two examples with multiple level recursions
- As an example to show that there is no infinite recursion, we are going to re-write the power function – use a new function name pow

# power(x, n) => pow(x,n)

Computes powers of the form $x^n$

```
double power(double x, int n)
// Library facilities used: cassert
  {
  double product; // The product of x with itself n times
  int count;
  if (x == 0) assert(n > 0);
  if (n >= 0)  // stopping case
  {
     product = 1;
     for (count = 1; count <= n; count++)
        product = product * x;
     return product;
  }
  else // recursive call
     return 1/power(x, -n);
}
```

change this into a recursive call based on the observation

$x^n = x\, x^{n-1}$ if $n>0$

# pow (x, n): Alternate Implementation

Computes powers of the form $x^n$

```
double pow(double x, int n)
// Library facilities used: cassert
{
    if (x == 0)
    {   // x is zero, and n should be positive
        assert(n > 0);
        return 0;
    }
    else if (n == 0)
        return 1;
    else if (n > 0)
        return x * pow(x, n-1);
    else // x is nonzero, and n is negative
        return 1/pow(x, -n);
}
```

All of the cases:

| $x$ | $n$ | $x^n$ |
|---|---|---|
| =0 | <0 | underfined |
| =0 | =0 | underfined |
| =0 | > 0 | 0 |
| !=0 | < 0 | $1/x^{-n}$ |
| !=0 | = 0 | 1 |
| !=0 | > 0 | $x*x^{n-1}$ |

# How to ensure NO Infinite Recursion

- when the recursive calls go beyond one level deep
- You can ensure that a stopping case is eventually reached by defining a numeric quantity called variant expression - without really tracing through the execution
- This quantity must associate each legal recursive call to a single number, which changes for each call and eventually satisfies the condition to go to the stopping case

# Variant Expression for pow

- The variant expression is abs(n)+1 when n is negative and
- the variant expression is n when n is positive
- A sequence of recursion call
  - pow(2.0, -3) has a variant expression abs(n)+1, which is 4; it makes a recursive call of pow(2.0, 3)

# Variant Expression for pow

- The variant expression is abs(n)+1 when n is negative and
- the variant expression is n when n is positive
- A sequence of recursion call
  - pow(2.0, 3) has a variant expression n,
    which is 3; it makes a recursive call of pow(2.0, 2)

# Variant Expression for pow

- The variant expression is abs(n)+1 when n is negative and
- the variant expression is n when n is positive
- A sequence of recursion call
  - pow(2.0, 2) has a variant expression n,
    which is 2; it makes a recursive call of pow(2.0, 1)

# Variant Expression for pow

- The variant expression is abs(n)+1 when n is negative and
- the variant expression is n when n is positive
- A sequence of recursion call
  - pow(2.0, 1) has a variant expression n,
    which is 1; it makes a recursive call of pow(2.0, 0)

# Variant Expression for pow

- The variant expression is abs(n)+1 when n is negative and
- the variant expression is n when n is positive
- A sequence of recursion call
  - pow(2.0, 0) has a variant expression n, which is 0; this is the stopping case.

# Ensuring NO Infinite Recursion

- It is enough to find a variant expression and a threshold with the following properties (p446):
  - Between one call of the function and any succeeding recursive call of that function, the value of the variant expression decreases by at least some fixed amount.
    - What is that fixed amount of pow(x,n)?
  - If the function is called and the value of the variant expression is less than or equal to the threshold, then the function terminates without making any recursive call
    - What is the threshold of pow(x,n)
- Is this general enough?

# Reasoning about the Correctness

- First show NO infinite recursion **then** show the following two conditions are also valid:
  - Whenever the function makes no recursive calls, show that it meets its pre/post-condition contract (BASE STEP)
  - Whenever the function is called, by assuming all the recursive calls it makes meet their pre-post condition contracts, show that the original call will also meet its pre/post contract (INDUCTION STEP)

# pow (x, n): Alternate Implementation

Computes powers of the form $x^n$

```
double pow(double x, int n)
// Library facilities used: cassert
{
    if (x == 0)
    {   // x is zero, and n should be positive
        assert(n > 0);
        return 0;
    }
    else if (n == 0)
        return 1;
    else if (n > 0)
        return x * pow(x, n-1);
    else // x is nonzero, and n is negative
        return 1/pow(x, -n);
}
```

All of the cases:

| x | n | $x^n$ |
|---|---|---|
| =0 | <0 | underfined |
| =0 | =0 | underfined |
| =0 | > 0 | 0 |
| !=0 | < 0 | $1/x^{-n}$ |
| !=0 | = 0 | 1 |
| !=0 | > 0 | $x*x^{n-1}$ |

# Summary of
# Reason about Recursion

- First check the function always terminates (not infinite recursion)
- next make sure that the stopping cases work correctly
- finally, for each recursive case, pretending that you know the recursive calls will work correctly, use this to show that the recursive case works correctly

# Reading, Exercises and Assignment

- Reading
  - Section 9.3
- Self-Test Exercises
  - 13-17
- Assignment 5 online
  - four recursive functions
  - due on Wednesday, November 9, 2016
- **Exam 2 – November 07 (**Monday**)**
  - Come to class on Wed (Nov 2) for reviews and assignment discussions