Recursion

1 Recursion

Important questions:

- What is *recursion*?
- Why is it so useful?
- How is it like induction?
- When should we use recursion?

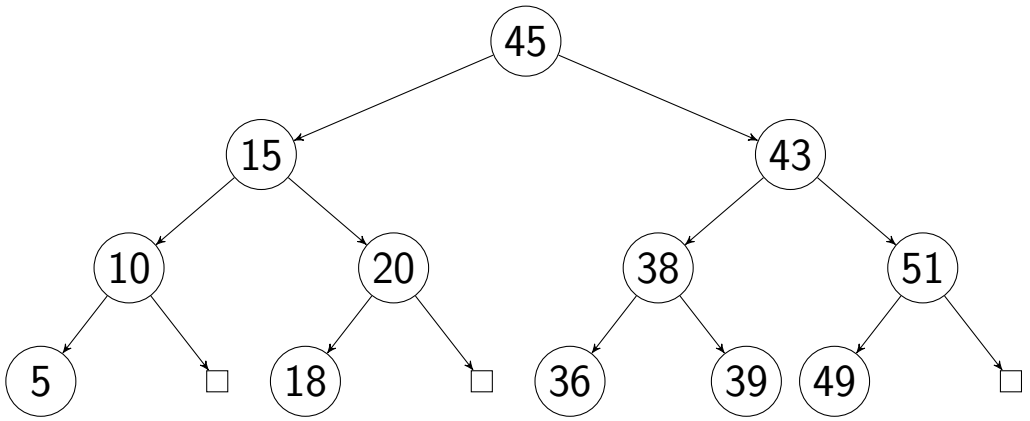
- What is *recursion*?
  Function that uses itself.
- Why is it so useful?
  Natural formulation for many problems.
- How is it like induction?
  Base case and a general case.
- When should we use recursion?
  When it works.
  How do we know it will work?
  Works (time and memory)

1.1 Example: Printing a tree in-order

Recursion is often an easy way to solve complex problems. Consider a tree structure:

```
struct TreeNode
{
    int  data;
    TreeNode *left;
    TreeNode *right;
};
```

This structure can be used to represent a *binary* tree. How could we print the following binary tree?[1]



[1]The small square, □, represents a NULL.

The method `PrintTree()` displays all the data stored in the tree!

```
void  PrintTree( TreeNode *t )
{
    if( t != NULL )   //  check that node is valid
    {
        PrintTree( t->left );

        cout << t->data << endl;

        PrintTree( t->right );
    }
}
```

This particular problem and several others will be examined in detail in the near future.

1.2 Example: Factorial

The *factorial* function can be defined using the following definition:

$$Factorial(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } 1 \\ n * Factorial(n-1) & n > 1 \end{cases}$$

How can we write this?

```
int Factorial( int n )
{
   if( n == 0 || n == 1 )
      return n;              // was return n;
   else
      return n * Factorial(n-1);
}
```

Output for $0 \ldots 6$:

```
1 1 2 6 24 120 720
```

1.3 Example: Fibonacci Series

The *fibonacci* series can be defined using the following definition:

$$Fib(n) = \begin{cases} 1, & \text{if } n = 0 \text{ or } 1 \\ Fib(n-1) + Fib(n-2) & n > 1 \end{cases}$$

How can we write this?

```
int Fib( int n )
{
   if( n == 0 || n == 1 )
      return n;
   else
      return Fib(n-1) + Fib(n-2);
}
```

Output:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

In Ruby, we can write a *function* as:

```
def fib(n)
    return n if n == 0 || n == 1
    return fib(n-1) + fib(n-2)
end

def fibUpTo(max)
   i1, i2 = 1, 1   #  parallel assignment
   while i1 <= max
      yield i1
      i1, i2 = i2, i1+i2
   end
end

fibUpTo( 1000 ) { |f| print f, " " }
```

Output:

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987

fib(25)  75025
fib(35)  9227465       (after a long delay!)
```

1.4 Example: Printing Numbers in Decimal

Write a function to print a number one digit per line.

Things to consider before attempting to implement:

- What is the base case?
- How could you do this *without* using recursion?

```
/*  PrintDecimal
 *
 *  Print a decimal number, n, one digit per line.
 */

void  PrintDecimal( int n )
{
   if( n < 10 )   //  base case
   {
       cout << n << endl;
   }
   else
   {
       PrintDecimal( n/10 );
       cout << n % 10 << endl;
   }
}
```

Things to think about:

- What happens if we reverse the two statements in the `else` portion?
- Why is there an `endl` in the output statements?
- How could this be extended to print to an `ofstream`?

1.5 Example: Printing Numbers in Binary

Write a function to print an integer in binary form.

Things to consider before attempting to implement:

- What is the base case?
- How could you do this *without* using recursion?

```
/*  PrintBinary
 *
 *  Print a decimal number, n, in binary form.
 */

void  PrintBinary( int n )
{
   if( n < 2 )    //  base case
   {
       cout << n;
   }
   else
   {
       PrintBinary( n/2 );
       cout << n % 2;
   }
}
```

A Python loop version

http://twistedmatrix.com/users/jh.twistd/python/moin.cgi/PostYourCode

```
def dec2bin(x):
    res = ""
    while x > 0:
        res = "%d%s" % (x & 1, res)
        x = x >> 1
        if res == "":
            res = 0
    return res
```

Things to think about:

- How is this function like the algorithm (repeated division) we discussed in lecture previously?
- Why is there no `endl` in the output statements?
- How could this be extended to print to an `ofstream`?
- How could this function be extended to print octal or hexadecimal numbers?

Types of Recursion:

- Tail
- Structural
- Mutual

Tail Recursion

Tail-recursive functions are a special kind of recursive function where all the work is done *before* the recursive call; the value returned by the call is passed to the level above with no further changes.[2]

Structural Recursion

Structural recursion refers to a style of programming where the structure of a function's recursive calls mirrors the structure of its input.

Mutual Recursion

Mutual-recursive functions are recursive functions that recurse on each other.

[2]Lisp: A Gentle Introduction to Symbolic Computation, David S. Touretzky