Objectives

In these lectures, we'll consider:

- Recursive Definitions
- Function Calls and Recursive Implementation
- Anatomy of a Recursive Call
- Tail Recursion
- Nontail Recursion
- Indirect Recursion
- Excessive Recursion

Anatomy of a Recursive Call

```
double power (double x, unsigned int n) {
    if (n == 0)
        return 1.0;
    else
        return x * power(x,n-1);
}
```

- Using the definition, the calculation of x^4 would be calculated as follows: $x^4 = x \cdot x^3 = x \cdot (x \cdot x^2) = x \cdot (x \cdot (x \cdot x^1)) = x \cdot (x \cdot (x \cdot (x \cdot x^0))) = x \cdot (x \cdot (x \cdot (x \cdot 1))) = x \cdot (x \cdot (x \cdot (x))) = x \cdot (x \cdot (x \cdot x)) = x \cdot (x \cdot x \cdot x) = x \cdot x \cdot x \cdot x$
- Notice how repeated application of the inductive step leads to the base case

Anatomy of a Recursive Call

- The base case produces the result of x^0, which is 1, and returns this value to the previous call
- That call, which had been suspended, then resumes to calculate $x \cdot 1$, producing x
- Each succeeding return then takes the previous result and uses it in turn to produce the final result
- The sequence of recursive calls and returns looks like:

call 1	$x^4 = x \cdot x^3$	$= x \cdot x \cdot x \cdot x$
call 2	$x^3 = x \cdot x^2$	$= x \cdot x \cdot x$
call 3	$x^2 = x \cdot x^1$	$= x \cdot x$
call 4	$x^1 = x \cdot x^0$	$= x \cdot 1$
call 5	$x^0 = 1$	

3

Tail Recursion

- The nature of a recursive definition is that the function contains a reference to itself
- This reference can take on a number of different forms
- We're going to consider a few of these, starting with the simplest, ***tail recursion***
- The characteristic implementation of tail recursion is that a single recursive call occurs at the very end of the function
- Such a statement might look like this:

```
return same_function_name(<revised argument values>);
```

- No other statements follow the recursive call, and there are no other recursive calls prior to the call at the end of the function

4

Tail Recursion

- An example of a tail recursive function is the code:

```
void tail(int i) {
   if (i > 0) {
      cout << i << " ";
      tail(i-1);
   }
}
```

- In most cases, languages supporting loops should use a loop construct rather than recursion, to avoid the requirement to store multiple activation records – if the initial value of i is too large, the stack could get too big for the process's memory, and could crash!

5

5

Tail Recursion

- An example of an iterative form of the function is shown below (in not nearly as elegant code, IMO!):

```
void iterativeEquivalentOfTail(int i) {
   for ( ; i > 0; i--)
      cout << i << '';
}
```

6

6

Nontail Recursion

- As an example of another type of recursion, consider the following code:

```
void reverse() {
    char ch;
    cin >> ch;
    if (ch != '\n') {
        reverse();
        cout << ch;
    }
}
```

- The function simply displays a line of chars in reverse order
- This is NOT tail recursion, since there's more things to be done after the recursive call!

7

7

Nontail Recursion

- Iterative (non-recursive) versions of these functions illustrate several important points:
  - First, when implementing iteratively, the stack must be explicitly implemented and handled, whereas with recursion we simply use the function call trace as a natural "stack"
  - Second, the clarity of the algorithm, and often its brevity, are sacrificed as a consequence of the conversion
  - Therefore, unless compelling reasons are evident during the algorithm design, it's not unusual to use the recursive versions of such functions

8

8

Indirect Recursion

- The previous discussions have focused on situations where a function, `f()`, invokes itself recursively (***direct recursion***)
- However, in some situations, the function may be called not by itself, but by a function that it calls, forming a chain:

$$f() \rightarrow g() \rightarrow f()$$

- This is known as ***indirect recursion***
- The chains of calls may be of arbitrary length, such as:

$$f() \rightarrow f_1() \rightarrow f_2() \rightarrow \cdots \rightarrow f_n() \rightarrow f()$$

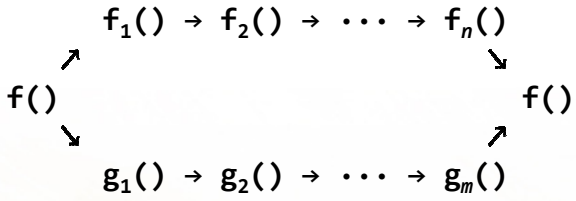
- It is also possible that a given function may be a part of multiple chains, based on different calls

9

9

Indirect Recursion

- So our previous chain might look like:



10

10

Excessive Recursion

- Recursive algorithms tend to exhibit simplicity in their implementation and are typically easy to read and follow
- However, this straightforwardness does have some drawbacks
- Generally, as the number of function calls increases, a program suffers from some performance decrease
- Also, the amount of stack space required increases dramatically with the amount of recursion that occurs
- This can lead to program crashes if the stack runs out of memory
- More frequently, though, is the increased execution time leading to poor program performance

11

11

Excessive Recursion

- As an example of this, consider the Fibonacci numbers
- They are first mentioned in connection with Sanskrit poetry as far back as 200 BCE
- Leonardo Pisano Bigollo (also known as Fibonacci), introduced them to the western world in his book *Liber Abaci* in 1202 CE
- The first few terms of the sequence are 0, 1, 1, 2, 3, 5, 8, …(*) and can be generated using the function:

(*) Fibonacci himself is credited for bringing the idea of zero (previously defined by Arabic, Mayan, Indian, and Mesopotamian mathematicians) to so-called "Western" mathematics! The original sequence started with 1,1,… instead of 0,1,1,… and zero today is called is usually called the "zeroth Fibonacci number."

$$Fib(n) = \begin{cases} n & \text{if } n < 2 \\ Fib(n-2) + Fib(n-1) & \text{otherwise} \end{cases}$$

12

12

Excessive Recursion

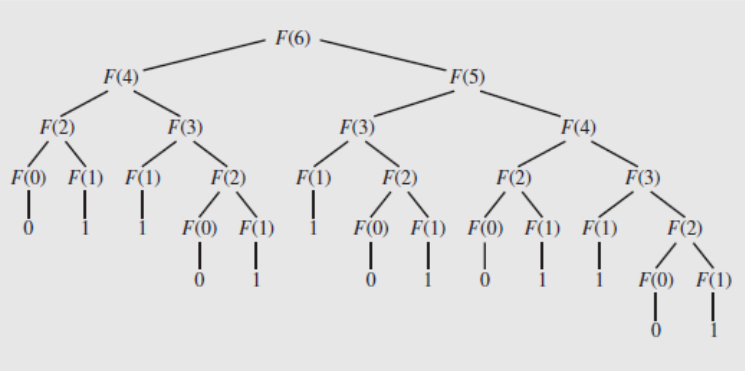
- This tells us that that any Fibonacci number after the first two (0 and 1) is defined as the sum of the two previous numbers
- However, as we move further on in the sequence, the amount of calculation necessary to generate successive terms becomes excessive
- This is because every calculation ultimately has to rely on the base case for computing the values, since no intermediate values are remembered
- The following algorithm implements this definition; again, notice the simplicity of the code that belies the underlying inefficiency

13

13

Excessive Recursion (continued)

- This process can be represented using a tree to show the calculations:



- Counting the branches, it takes 25 calls to **`Fib()`** just to calculate **`Fib(6)`**

14

14

Excessive Recursion

- The total number of additions required to calculate the n^{th} number can be shown to be Fib(n + 1) – 1
- With two calls per addition, and the first call taken into account, the total number of calls is 2 · Fib(n + 1) – 1
- Values of this are shown in the following table:

n	Fib(n+1)	Number of Additions	Number of Calls
6	13	12	25
10	89	88	177
15	987	986	1,973
20	10,946	10,945	21,891
25	121,393	121,392	242,785
30	1,346,269	1,346,268	2,692,537

15

15

Excessive Recursion

- Notice that it takes almost 3 million calls to determine the 31st Fibonacci number!
- This exponential growth makes the algorithm unsuitable for anything but small values of n
- Fortunately, there are acceptable iterative algorithms that can be used far more effectively (O(n) or even O(log n))

16

16

Concluding Remarks

- While there are no specific rules that require we use or avoid recursion in any particular situation, we can develop some general guidelines
- Recursion is generally less efficient than most iterative equivalents
- However, if the difference in execution times is fairly small, other factors such as clarity, simplicity, and readability may be taken into account
- Recursion often is more faithful to an algorithm's logic

17