Introduction to Recursion

Chapter 20.1-4

CS 2308
Fall 2018

Jill Seaman

What is recursion?

• Generally, when something contains a reference to itself
• Math: defining a function in terms of itself
• Computer science: when a function calls itself:

```c
void message() {
    cout << "This is a recursive function.\n";
    message();
}
int main() {
    message();
}
```

What happens when this is executed?

How can a function call itself?

• Infinite Recursion:

  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  This is a recursive function.
  ...  

Note: If you encounter infinite recursion in Lab, be sure to STOP your program BEFORE running it again!!!
Tracing the calls

- 6 nested calls to message:
  
  message(5):
  outputs “This is a recursive function”
  calls message(4):
  outputs “This is a recursive function”
  calls message(3):
  outputs “This is a recursive function”
  calls message(2):
  outputs “This is a recursive function”
  calls message(1):
  outputs “This is a recursive function”
  calls message(0):
  does nothing, just returns

- depth of recursion (#times it calls itself) = 5

How to write recursive functions

- Branching is required (If or switch)
- Find a base case
  - one (or more) values for which the result of the function is known (no repetition required to solve it)
  - no recursive call is allowed here
- Develop the recursive case
  - For a given argument (say n), assume the function works for a smaller value (n-1).
  - Use the result of calling the function on n-1 to form a solution for n

Recursive function example

**factorial**

- Mathematical definition of n! (factorial of n)
  
  if n=0 then n! = 1
  if n>0 then n! = 1 x 2 x 3 x ... x n-1 x n

- What is the base case?
  - n=0 (the result is 1)
- Recursive case: If we assume (n-1)! can be computed, how can we get n! from that?
  - n! = n * (n-1)!

Recursive function example

**factorial**

```cpp
int factorial(int n) {
    if (n==0)
        return 1;
    else
        return n * factorial(n-1);
}
```

```cpp
int main() {
    int number;
    cout << “Enter a number “;
    cin >> number;
    cout << “The factorial of “ << number << “ is “ << factorial(number) << endl;
}
```
Tracing the calls

- Calls to factorial:

  \[
  \text{factorial}(4): \\
  \text{return } 4 \times \text{factorial}(3); \quad 4 \times 6 = 24 \\
  \text{calls factorial}(3): \\
  \text{return } 3 \times \text{factorial}(2); \quad 3 \times 2 = 6 \\
  \text{calls factorial}(2): \\
  \text{return } 2 \times \text{factorial}(1); \quad 2 \times 1 = 2 \\
  \text{calls factorial}(1): \\
  \text{return } 1 \times \text{factorial}(0); \quad 1 \times 1 = 1 \\
  \text{calls factorial}(0): \\
  \text{return } 1; \\
  \]

- Every call except the last makes a recursive call
- Each call makes the argument smaller

Recursive functions: ints and lists

- Recursive functions over integers follow this pattern:

  ```
  \text{type } f(\text{int } n) \{ \\
  \quad \text{if } (n==0) \\
  \quad \quad \text{//do the base case} \\
  \quad \text{else} \\
  \quad \quad \text{// ... } f(n-1) \ldots \\
  \}
  ```

- Recursive functions over lists (arrays, linked lists, strings) use the \textit{length} of the list in place of \textit{n}
  - base case: if (length==0) ... // empty list
  - recursive case: assume \( f \) works for list of length \( n-1 \), compute the answer for a list with one more element.

Recursive function example

\textbf{sum of the list}

- Recursive function to compute sum of a list of numbers
- What is the base case?
  - length=0 (empty list) => sum = 0
- If we assume we can sum the first \( n-1 \) items in the list, how can we get the sum of the whole list from that?
  - sum (list) = sum (list[0]..list[n-2]) + list[n-1]

Assume I am given the answer to this

Recursive function example

\textbf{sum of a list (array)}

```c
int \text{sum}(\text{int } a[], \text{int } size) \{ \\
\quad \text{//size is number of elems} \\
\quad \text{if } (\text{size==0}) \\
\quad \quad \text{return } 0; \\
\quad \text{else} \\
\quad \quad \text{return } \text{sum}(a, \text{size-1}) + a[\text{size-1}]; \\
\}
```

- For a list with size = 4: \( \text{sum}(a,4) = \\
  \text{sum}(a,3) + a[3] = \\
  (\text{sum}(a,2) + a[2]) + a[3] = \\
  (((\text{sum}(a,1) + a[1]) + a[2]) + a[3]) = \\
  (0 + a[0] + a[1] + a[2] + a[3])
```
Recursive function example

Greatest common divisor

- Greatest common divisor of two non-zero ints is the largest positive integer that divides the numbers evenly (without a remainder)
- This is a variant of Euclid's algorithm:
  \[ \text{gcd}(x,y) = y \quad \text{if } x/y \text{ has no remainder otherwise:} \]
  \[ \text{gcd}(x,y) = \text{gcd}(y, \text{remainder of } x/y) \]
- It's a recursive definition, correctness is proven elsewhere.

Note: the recursive fibonacci function works as written, but it is VERY inefficient.

Counting the recursive calls to \( \text{fib} \): 

The first 40 fibonacci numbers:

\[
\begin{align*}
\text{fib}(0) &= 0 & \# \text{ of recursive calls to } \text{fib} &= 1 \\
\text{fib}(1) &= 1 & \# \text{ of recursive calls to } \text{fib} &= 1 \\
\text{fib}(2) &= 1 & \# \text{ of recursive calls to } \text{fib} &= 3 \\
\text{fib}(3) &= 2 & \# \text{ of recursive calls to } \text{fib} &= 5 \\
\text{fib}(4) &= 3 & \# \text{ of recursive calls to } \text{fib} &= 9 \\
\text{fib}(5) &= 5 & \# \text{ of recursive calls to } \text{fib} &= 15 \\
\text{fib}(6) &= 8 & \# \text{ of recursive calls to } \text{fib} &= 25 \\
\text{fib}(7) &= 13 & \# \text{ of recursive calls to } \text{fib} &= 41 \\
\text{fib}(8) &= 21 & \# \text{ of recursive calls to } \text{fib} &= 67 \\
\text{fib}(9) &= 34 & \# \text{ of recursive calls to } \text{fib} &= 109 \\
\text{fib}(10) &= 55 & \# \text{ of recursive calls to } \text{fib} &= 170 \\
\text{fib}(11) &= 89 & \# \text{ of recursive calls to } \text{fib} &= 289 \\
\text{fib}(12) &= 144 & \# \text{ of recursive calls to } \text{fib} &= 473 \\
\text{fib}(13) &= 233 & \# \text{ of recursive calls to } \text{fib} &= 756 \\
\text{fib}(14) &= 377 & \# \text{ of recursive calls to } \text{fib} &= 1233 \\
\text{fib}(15) &= 610 & \# \text{ of recursive calls to } \text{fib} &= 2085 \\
\end{align*}
\]

... 

\[ \text{fib}(40) = 102,334,155 \quad \# \text{ of recursive calls to } \text{fib} = 331,160,281 \]