Introduction to Recursion

Chapter 20.1-4

CS 2308
Spring 2019

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.
  ...
**Tracing the calls**

- 6 nested calls to message:
  - message(5):
    - outputs “This is a recursive function”
  - message(4):
    - outputs “This is a recursive function”
  - message(3):
    - outputs “This is a recursive function”
  - message(2):
    - outputs “This is a recursive function”
  - message(1):
    - outputs “This is a recursive function”
  - 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);
}

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:
  ```
  factorial(4):
    return 4 * factorial(3); = 4 * 6 = 24
  calls factorial(3):
    return 3 * factorial(2); = 3 * 2 = 6
  calls factorial(2):
    return 2 * factorial(1); = 2 * 1 = 2
  calls factorial(1):
    return 1 * factorial(0); = 1 * 1 = 1
  calls factorial(0):
    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:
  ```
  type f(int n) {
    if (n==0) // do the base case
      return 1;
    else
      return f(n-1); // ... f(n-1) ...
  }
  ```
- Recursive functions over lists (arrays, linked lists, strings) use the length of the list in place of 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
**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
**sum of a list (array)**

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

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:
  
  $$gcd(x,y) = y \text{ if } x/y \text{ has no remainder otherwise:}$$
  
  $$gcd(x,y) = gcd(y, \text{remainder of } x/y)$$

- It’s a recursive definition, correctness is proven elsewhere.

**Code:**

```c++
int gcd(int x, int y) {
    if (x % y == 0) {
        return y;
    } else {
        return gcd(y, x % y);
    }
}
```

### Note:

- The recursive fibonacci functions works as written, but it is VERY inefficient.

**Counting the recursive calls to fib:**

The first 40 fibonacci numbers:

<table>
<thead>
<tr>
<th>Number</th>
<th>Recursive Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>7</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>34</td>
</tr>
<tr>
<td>9</td>
<td>55</td>
</tr>
<tr>
<td>10</td>
<td>89</td>
</tr>
<tr>
<td>11</td>
<td>144</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>331,160,281</td>
</tr>
</tbody>
</table>

**Fibonacci numbers**

- **Series of Fibonacci numbers:**
  - Starts with 0, 1. Then each number is the sum of the two previous numbers
    - $F_0 = 0$
    - $F_1 = 1$
    - $F_i = F_{i-1} + F_{i-2}$ (for $i > 1$)
- **It’s a recursive definition**
  ```c++
  int fib(int x) {
    if (x==0 || x==1) {
      return x;
    } else {
      return fib(x-1) + fib(x-2);
    }
  }
  ```