

Computer Science & Engineering 150A

Problem Solving Using Computers

Lecture 10 - Recursion

Christopher M. Bourke

Spring 2009

- Functions in C routinely call other functions
- Example: the `main` function calls the `quadraticRoot01` function, which calls the `discriminant` function, which calls the `sqrt` function
- C allows functions to also *call themselves*
- This is known as *recursion*

- Recursive functions are common in mathematics
- Sequences are recursively defined functions
- Recall the interpolation method for computing the square root:

$$x_i = \frac{1}{2} \left(x_{i-1} + \frac{n}{x_{i-1}} \right)$$

- Functions defined using the functions in the definition (*recurrence relations*)
- Canonical example: the Fibonacci sequence

- Fibonacci sequence defined as the sum of its two previous elements
- Named for Leonardo of Pisa, known as Fibonacci (a contraction of filius Bonaccio, "son of Bonaccio")

$$F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{if } n > 1 \end{cases}$$

Sequence:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Fibonacci Sequence

From Math to C

CSCE150A

Introduction

Factorial

Rules

Tracing
Recursive
Calls

Factorial

Pros & Cons

- We can easily translate a recursive mathematical function to a recursive C function
- We just have to be careful to handle certain issues
- We design a function that calls itself: a function that calls a function of the same name

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  /* Computes the n-th Fibonacci number */
5  int fibonacci(int n);
6
7  int main(int argc, char *argv[])
8  {
9      if(argc != 2)
10     {
11         printf("usage: a.out n\n");
12         exit(-1);
13     }
14     int n = atoi(argv[1]);
15     printf("%d! = %d\n", n, fibonacci(n));
16 }
17
18 int fibonacci(int n)
19 {
20     if(n < 0)
21         return -1;
22     else if(n == 0)
23         return 0;
24     else if(n == 1)
25         return 1;
26     else
27         return fibonacci(n-1) + fibonacci(n-2);
28 }
```

When using recursion, some rules must be followed:

- 1 The function must have at least one *terminating condition*
- 2 The function must *make progress* toward a terminating condition

- We need some guarantee that a recursive function will eventually halt
- A recursive function must have at least one *terminating condition*
- A “base case” in which the function does not call itself again
- In the Fibonacci program: Three terminating conditions
- Each returns a specific value without calling `fibonacci` again

- Must, in some way, make progress toward the terminating condition
- Incrementing/decrementing the passed values
- Fibonacci example: each recursive call, `fibonacci(n-1)`, `fibonacci(n-2)` decrements n
- Progress is made toward the terminating conditions
- Out of bounds check: function is undefined for $n < 0$
- If all possibilities are not handled: infinite recursion

- To understand recursion, it is helpful to trace a recursive function call
- Example: `fibonacci(5)`: on the first call, the function makes two calls: `fibonacci(4)` and `fibonacci(3)`
- Each one makes two recursive calls, and each one of those makes its own recursive calls, etc.
- Full computation can be illustrated with a *recursion tree*

Tracing a Recursive Call

CSCE150A

Introduction

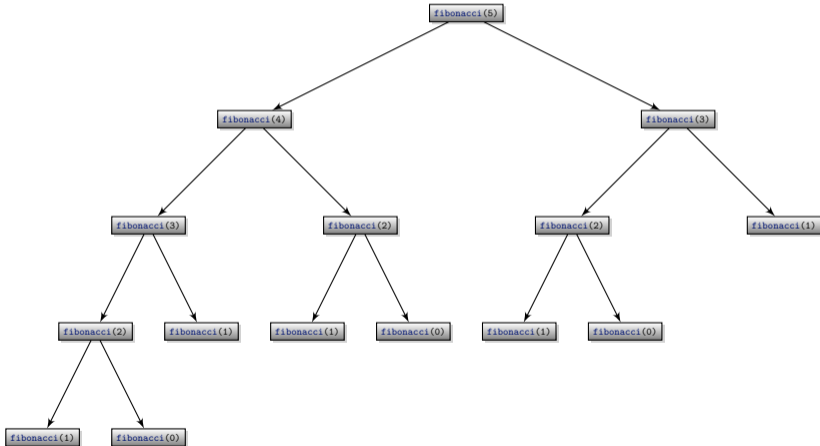
Factorial

Rules

Tracing
Recursive
Calls

Factorial

Pros & Cons



- Note: `fibonacci(5)` required 15 calls to `fibonacci`
- Many calls were unnecessary: `fibonacci(2)` was called three times!
- Extensive recomputation is required in this case
- Number of recursive calls is *exponentially* large
- Better way of computing F_n ?

- Recall the factorial function:

$$n! = n \times (n - 1) \times (n - 2) \times \cdots , 2 \times 1$$

- We can also write a recursive function for this function:

$$F_n = \begin{cases} 1 & \text{if } n = 1 \\ F_{n-1} \times n & \text{otherwise} \end{cases}$$

- Strategy:
 - Identify and handle the base case(s)
 - Identify and handle the recursive call

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int factorial(int n);
5
6  int main(int argc, char *argv[])
7  {
8      if(argc != 2)
9      {
10         printf("usage: a.out n\n");
11         exit(-1);
12     }
13     int n = atoi(argv[1]);
14     printf("%d! = %d\n",n,factorial(n));
15 }
16
17 int factorial(int n)
18 {
19     if(n < 1)
20         return 0;
21     if(n == 1)
22         return 1;
23     else
24         return n * factorial(n-1);
25 }
```

Factorial Recursion Tree

CSCE150A

Introduction

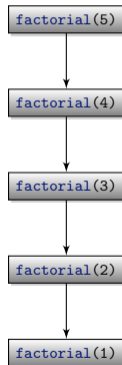
Factorial

Rules

Tracing
Recursive
Calls

Factorial

Pros & Cons



CSCE150A

Introduction

Factorial

Rules

Tracing
Recursive
Calls

Factorial

Pros & Cons

- $5!$ only requires five calls to `factorial`
- Recursion is linear in depth and number of calls

CSCE150A

Introduction

Factorial

Rules

Tracing
Recursive
Calls

Factorial

Pros & Cons

- Programming languages do not have to support recursion
- Non-trivial fact: Any recursive function can be made non-recursive
- Arguments for and against recursion exist

Advantages:

- Simplified code
- Closely matches a Divide & Conquer approach to problems solving
- In some (*limited*) instances, recursion may be more efficient

Disadvantages:

- Generally inefficient: requires many system stack swaps
- May needlessly recompute values (Fibonacci sequence)
- May be harder to debug and/or consider all possibilities

CSCE150A

Introduction

Factorial

Rules

Tracing
Recursive
Calls

Factorial

Pros & Cons

Exercise

In class exercise: Write a non-recursive function for the fibonacci sequence. Modify the main driver program to count the number of additions that are preformed (this will require a global variable) and compare the performance of the two functions.

The binomial coefficients, $C(n, k)$ or $\binom{n}{k}$ (“ n choose k ”), are defined as the number of ways you can select k distinct items from a collection of n items. A direct combinatorial definition is

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

An alternative is Pascal’s identity, which gives a recurrence to compute this value:

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Where $\binom{n}{0} = 0$ for any n and for all $k > n$, $\binom{n}{k} = 0$. Finally, $\binom{n}{1} = n$.

Exercise

Write a recursive function to compute $\binom{n}{k}$ using this formula. Then write a function that uses the factorial definition and try to compute $\binom{30}{12}$ with each one. What answers do you get and why? Write a main function that takes n, k as command line arguments and outputs the result of $\binom{n}{k}$ for both the recursive definition and for the factorial definition.