Computer Science & Engineering 155E
Computer Science I: Systems Engineering Focus

Lecture – Recursion

Christopher M. Bourke
cbourke@cse.unl.edu

Recursion

- 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

- 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

- 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:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots$$

Fibonacci Sequence

From Math to C

- 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

Recursion

C Code

```c
#include<stdlib.h>
#include<stdio.h>

/* Computes the n-th Fibonacci number */
int fibonacci(int n);

int main(int argc, char *argv[])
{
  if(argc != 2)
  {
    printf("usage: a.out n\n");
    exit(-1);
  }
  int n = atoi(argv[1]);
  printf("%d! = %d\n",n,fibonacci(n));
}

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

Recursion
Rules

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

Recursion
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

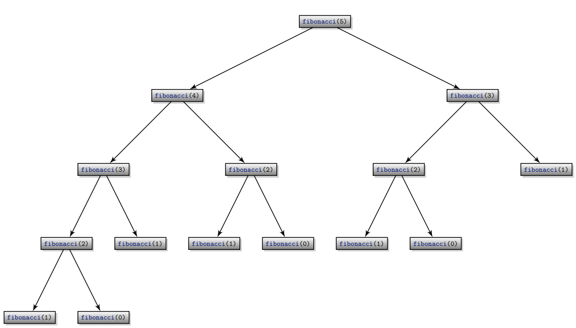
Recursion
Progress

- 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

Tracing a Recursive Call

- 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



Tracing a Recursive Call

- 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?

Factorial Example

- 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:
  1. Identify and handle the base case(s)
  2. Identify and handle the recursive call

Factorial C Code

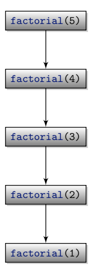
```c
#include<stdlib.h>
#include<stdio.h>

int factorial(int n);

int main(int argc, char *argv[])
{
  if(argc != 2)
  {
    printf("usage: a.out n\n");
    exit(-1);
  }
  int n = atoi(argv[1]);
  printf("%d! = %d\n",n,factorial(n));
}

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

Factorial Recursion Tree



Factorial Example

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

Advantages & Disadvantages I

Overview

- Some languages may not 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

Advantages & Disadvantages II

Disadvantages:

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

Better Alternatives:

- Tail recursion (no local state to take up the program stack)
- Use smarter data structures (stacks)
- Use *memoization* (use of a table to store function values to avoid repeating the same call)

Exercise I

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.

Exercise II I

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

Exercise II II

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.