CSCI 112
Programming with C

Chapter 9
Recursion

Recursion

- We have seen functions, such as our main, call other functions.
- Sometimes functions call themselves. This is common in mathematics, such as this definition of factorial:

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n >= 1 \end{cases}$$

- Note that the definition of factorial uses the factorial operation itself. This is known as recursion.

Recursion

- Definition
  - Recursion describes the process when a function calls itself.
  - A function, f , is also said to be recursive if it calls another function, g, which in turn calls f.

- Recursion is sometimes less efficient than iterative (loop based) approaches, but often provides a more natural and simple solution.
- Recursion is a powerful programming tool to solve certain problems.

Recursion

- Recursion uses a strategy known as divide and conquer. This approach continually reduces the problem size until it reduces to a simple case with an obvious solution.

- The simple cases are known as base cases.

- The other cases, known as recursive cases, redefine the problem in such a way as to move closer to the base case.

Motivating Example

- Goal:
  - function to compute sum(n) = 0 + 1 + ... + n-1 + n

```
int sum(int n) {
    int i, s = 0;
    for (i = 0; i <= n; i++) {
        s += i;
    }
    return s;
}
```

```
int sum(int n) {
    int i, s = n;
    while (n <= 0) {
        n--;
        s += n;
    }
    return s;
}
```

Motivating Example

- Goal: create a function to compute
  - sum(n) = 0 + 1 + ... + n-1 + n.
- Observe that:
  - sum(n) = n + sum(n-1)
  - sum(n-1) = 0 + 1 + ... + n-1
- We can write this a recursive denition:

$$sum(n) = \begin{cases} 0 & n = 0 \\ n + sum(n-1) & n >= 1 \end{cases}$$

Basics

- The function needs to have these basics:

if this is a simple or base case

    solve it

else

    redefine the problem using recursion

Anatomy of a Recursive Function

- Goal:
  - function to compute sum(n) = 0 + 1 + ... + n-1 + n

$$sum(n) = \begin{cases} 0 & n = 0 \\ n + sum(n-1) & n >= 1 \end{cases}$$

```
int sum(int n) {
    if (n == 0) {
        return 0;
    }
    return n + sum(n-1);
}
```

- Note that the recursive case will converge to the base case.

Another Example

- <multiply.c>

Factorial Example

- Let's return to the factorial example.

$$n! = \begin{cases} 1 & n = 0 \\ n(n-1)! & n >= 1 \end{cases}$$

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

Bad Recursion

```
void bad_recursion(int n) {
   printf("%d\n", n);              // BAD: missing base case
   if (n%2 == 0) {                 // GOOD: reduction case
      bad_recursion(n/2); }        // BAD: only if n is even
   else {
      bad_recursion(3*n + 1);  // BAD: step does not
}                                  //        reduce problem
```

- Recursion will not terminate for all possible inputs of positive n.
- A missing base case means the recursion will not \bottom out".
- This is analogous to an infinite loop.

GCD Example

- Goal: Find largest integer d that evenly divides into m and n.

$$gcd(m,n) = \begin{cases} m & n = 0 \\ gcd(n, m\%n) & otherwise \end{cases}$$

```
int gcd(int m, int n) {
    if (n == 0) {
        return m;
    }
    else {
        return gcd(n, m%n);
    }
}
```

gcd(4,2)
gcd(2,4%2=0) = gcd(2,0)=2

gcd.c

Fibonacci Example

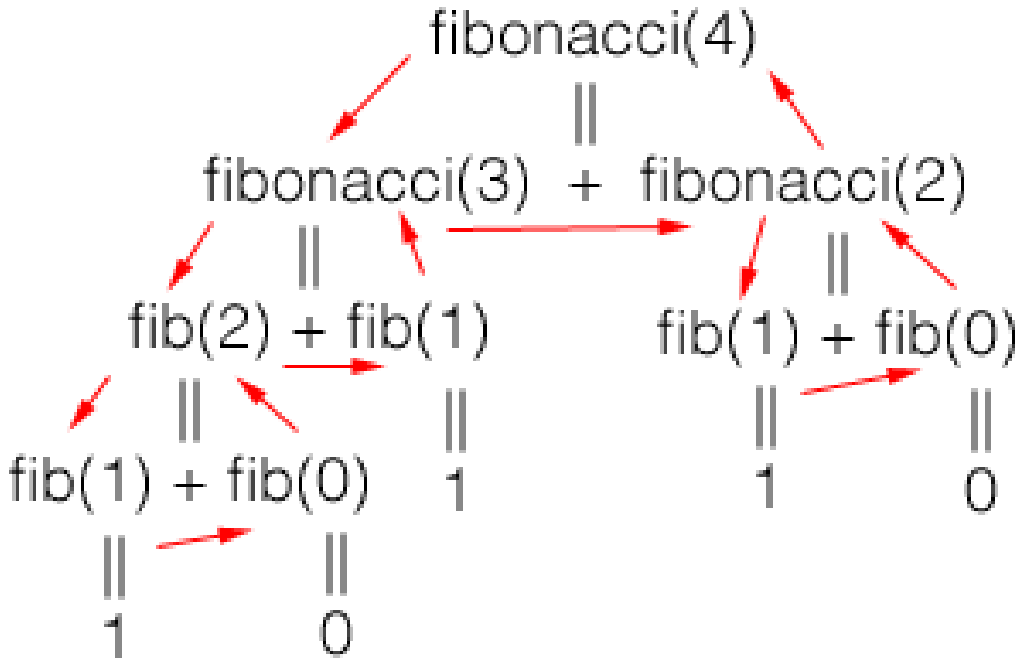
- Consider the Fibonacci sequence:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

$$fib(n) = \begin{cases} 1 & n = 1 \\ 1 & n = 2 \\ fib(n-1) + fib(n-2) & n > 2 \end{cases}$$

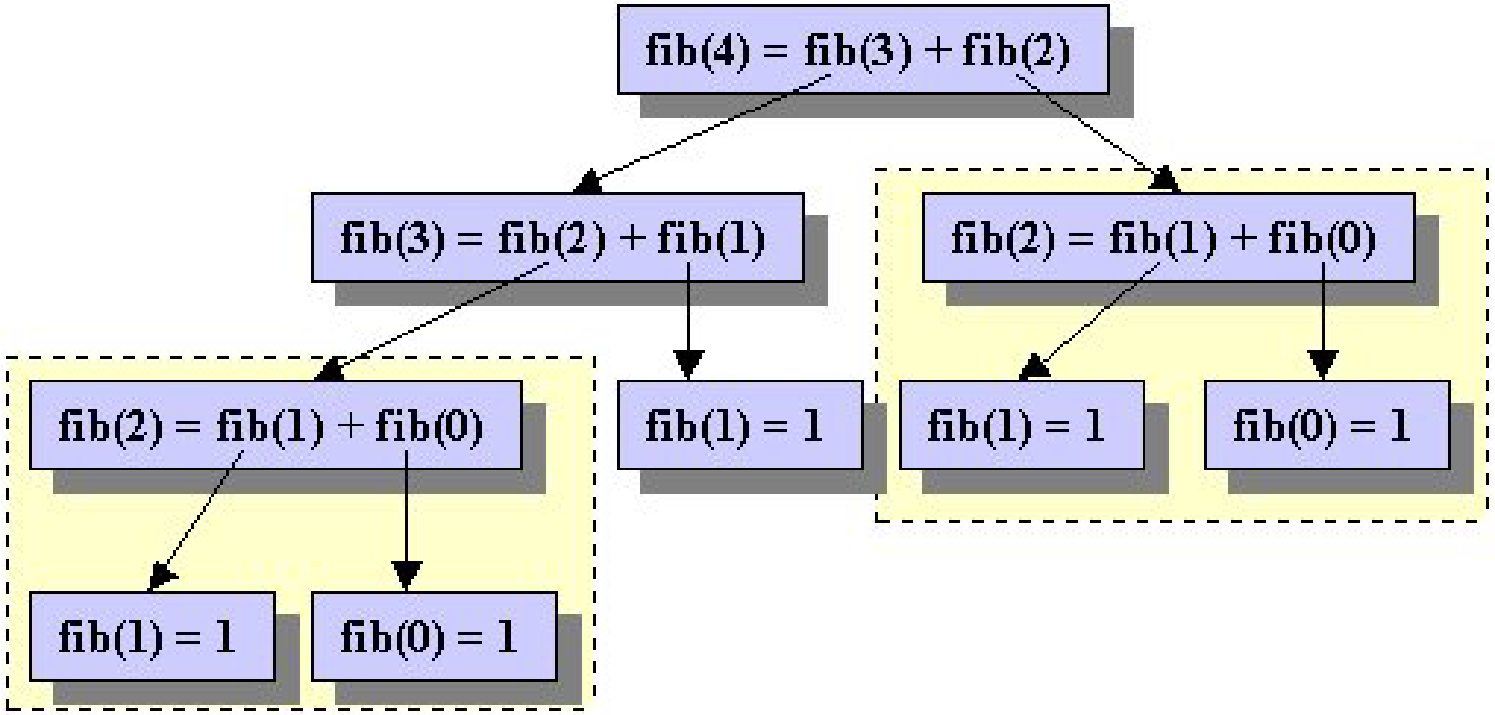
fib.c

```
int fib(int n) {
   if (n == 1) {
      return 1;
   }
   else if (n == 2) {
      return 1;
   }
   else {
      return (fib(n – 1) + fib(n – 2));
   }
}
```

Fibonacci -Order of Function Calls



Fibonacci - Repeated Subproblems



Mutual Recursion

- Goal: determine is number is even or odd.
  - If n is equal to 0, then n is even
  - n is odd if n - 1 is even

```
int is_even(unsigned int n) {
   if (n == 0) {
      return 1;                    // Base case
   }
   else {
      return is_odd(n – 1);     // Recursive call to odd
   }
}

int is_odd(unsigned int n) {
   if (n == 0) {
      return 0;                    // Base case
   }
   else {
      return is_even(n – 1);  // Recursive call to even
   }
}
```

Conclusion

- Things To Remember
  - Recursive functions call themselves, directly or indirectly.
  - Base cases bottom out the recursive process.
  - Recursive cases reduce the problem to smaller problems.
  - Some languages (e.g., LISP) are optimized for recursion.