Data Structures

Recursion 1

Chapter 5

Data Structures and Algorithms by Adam Drozdek

Recursive Function Call

- A recursive call is a function call in which the called function is the same as the one making the call

- In other words, recursion occurs when a function calls itself!

- We must avoid making an infinite sequence of function calls (infinite recursion)

Finding a Recursive Solution

- Each successive recursive call should bring you closer to a situation in which the answer is known
- A case for which the answer is known (and can be expressed without recursion) is called a base case
- Each recursive algorithm must have at least one base case, as well as the general (recursive) case

General Format for Many Recursive Functions

```
if (some condition for which answer is known)
                                          //base case
      solution statement
else                                      // general case
      recursive function call
```

Writing a Recursive Function to Find Factorial

- The function call Factorial(4) should have value 24, because that is 4 * 3 * 2 * 1
- For a situation in which the answer is known, the value of 0! is 1
- So our base case could be along the lines of

```
if ( number == 0 )
    return 1;
```

Writing a Recursive Function to Find Factorial

- Now for the general case . . .
- The value of Factorial(n) can be written as n * the product of the numbers from (n -1) to 1, that is,

```
 n * (n -1) * (n-2) * (n-3) * ...* 3 * 2 * 1
```

or, n! = n * (n -1)! = n * (n-1) * (n-2)! = ... = ...1! = ...1*0!

And notice that the recursive call Factorial(n -1) gets us "closer" to the base case of Factorial(0)

Recursive Solution

```
int factorial ( int n ){
    // Pre: number is assigned and number >= 0
    if ( n==0 )// base case, for termination
        return 1 ;
    else`
        return n * factorial ( n -1 );
}
```

Three-Question Method of Verifying Recursive Functions

Base-Case Question: Is there a non-recursive way out of the function?

The answer should be "yes"

Smaller-Caller Question: Does each recursive function call involve a smaller case of the original problem leading to the base case?

The answer should be "yes"

General-Case Question: Assuming each recursive call works correctly, does the whole function work correctly?

The answer should be "yes"

Another Example With Natural Recursion

- From mathematics, we know that

  $2^0 = 1$ and $2^5 = 2 * 2^4$

- In general,

  $x^0 = 1$ and $x^n = x * x^{n-1} = x*x*x^{n-2}$

  for integer x, and integer n > 0

- Here we are defining x^n recursively, in terms of x^{n-1}
- $x^n = x * x * x * x* \ldots\ldots$ n times

Another Example With Natural Recursion

- From mathematics, we know that

  $2^0 = 1$ and $2^5 = 2 * 2^4$
- In general,

  $x^0 = 1$ and $x^n = x * x^{n-1} = x*x*x^{n-2}$

  for integer x, and integer n > 0

- Here we are defining x^n recursively, in terms of x^{n-1}
- $x^n = x * x * x * x*$ …… n times

Recursive Solution Power Function

```
int power (int x , int n ){
    // Pre: n >= 0. x, n are not both zero
    // Post: Function value = x raised to the power n
    if ( n==0 )// base case, for termination
        return 1 ;
    else`
        return x * power (x, n -1 );
}
```

How Recursion Works

Power: x^n

$$x^4 = x \cdot x^3 = x \cdot (x \cdot x^2) = x \cdot (x \cdot (x \cdot x^1)) = x \cdot (x \cdot (x \cdot (x \cdot x^0)))$$
$$= x \cdot (x \cdot (x \cdot (x \cdot 1))) = x \cdot (x \cdot (x \cdot (x))) = x \cdot (x \cdot (x \cdot x))$$
$$= x \cdot (x \cdot x \cdot x) = x \cdot x \cdot x \cdot x$$

call 1	$x^4 = x \cdot x^3$	$= x \cdot x \cdot x \cdot x$
call 2	$x \cdot x^2$	$= x \cdot x \cdot x$
call 3	$x \cdot x^1$	$= x \cdot x$
call 4	$x \cdot x^0$	$= x \cdot 1 = x$
call 5	1	

or alternatively, as

call 1	`power(x,4)`
call 2	`power(x,3)`
call 3	`power(x,2)`
call 4	`power(x,1)`
call 5	`power(x,0)`
call 5	1
call 4	x
call 3	$x \cdot x$
call 2	$x \cdot x \cdot x$
call 1	$x \cdot x \cdot x \cdot x$

Non-recursive solution

Power: x^n

// Pseudo code. Remember: $x^n = x * x * x * x* \ldots\ldots$ n times
$= 1*x*x*\ldots.x$ n times

```
int nonRecursivePower(int x, int n){
      result = 1;
      for (i = 1; i<=n ; i++)
            result = result * x;
}
```

Another Example With Natural Recursion

- From mathematics, we know that

  $2^0 = 1$ and $2^5 = 2 * 2^4$

- In general,

  $x^0 = 1$ and $x^n = x * x^{n-1} = x*x*x^{n-2}$

  for integer x, and integer n > 0

- Here we are defining x^n recursively, in terms of x^{n-1}
- $x^n = x * x * x * x* \ldots\ldots$ n times

Why use recursion?

- Those examples could have been written without recursion, using iteration instead. The iterative solution uses a loop, and the recursive solution uses an if statement

- However, for certain problems the recursive solution is the most natural solution. This often occurs when pointer variables are used

- Recursion is easy to code

Function BinarySearch

- Binary Search takes sorted array info, and two subscripts, fromLoc and toLoc, and item as arguments. It returns false if item is not found in the elements info[fromLoc…toLoc]. Otherwise, it returns true.
- BinarySearch can be written using iteration, or using recursion

Non-recursive solution

```
boolean BinarySearch (int *d, int item , int from , int to ){
  // Pre: info [ from . . to ] sorted in ascending order
  // Post: Function value = ( item in info [ from.. . to] )
  int mid ;
  if ( from > to ) // base case --not found
    return false ;
  else {
    mid = ( from + to ) / 2 ;
    if ( d [ mid ] == item ) // base case--found at mid
      return true ;
    else if ( item < d [ mid ] ) //search lower half
      return BinarySearch ( d, item, from, mid-1 ) ;
    else // search upper half
    return BinarySearch( d, item, mid + 1, to ) ;
  }
}
```

Some Examples

Tail Recursion

```
void tail (int i) {
  if (i > 0) {
    System.out.print (i + "");
    tail(i-1);
  }
}
```

Non-Tail Recursion

```
void nonTail (int i) {
  if (i > 0) {
    nonTail(i-1);
    System.out.print (i + "");
    nonTail(i-1);
  }
}
```

Iteration / Loop

```
void iterativeEquivalentOfTail (int i) {
  for ( ; i > 0; i--)
  System.out.print(i+ "");

}
```

Recursion is not always good

- It may be very slow when excessive recursion calls-Example: Compute Fibonacci Number

$$\mathrm{Fib}(n) = \begin{cases} n & \text{if } n < 2 \\ \mathrm{Fib}(n-2) + \mathrm{Fib}(n-1) & \text{otherwise} \end{cases}$$

$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

Compute Fibonacci Number

- Recursive solution:

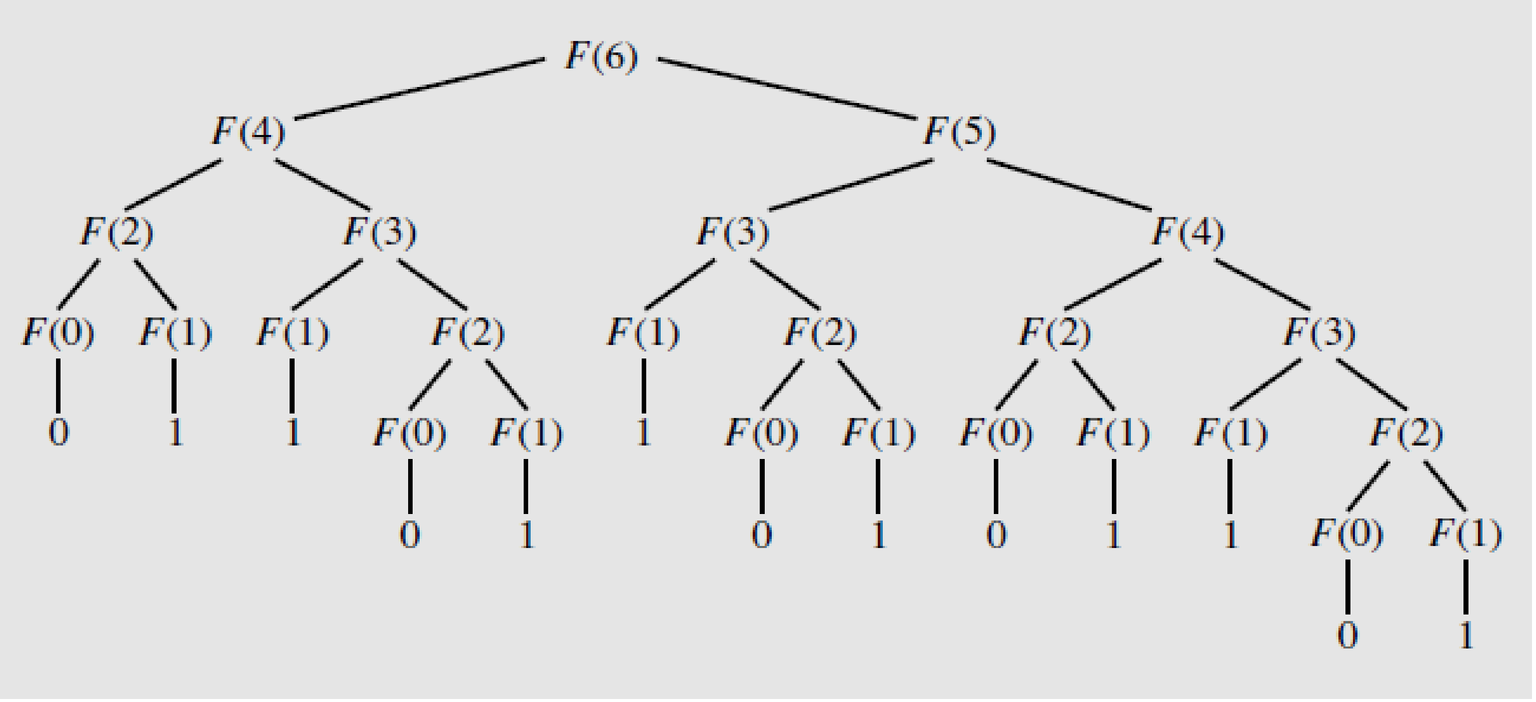
```
int Fib (int n) {
  if (n < 2)
    return n;
  else return Fib(n-2) + Fib(n-1);
}
```

```
  Fib(6) =              Fib(4)                 + Fib(5)
         =    Fib(2)       +        Fib(3)       + Fib(5)
         = Fib(0)+Fib(1)   +        Fib(3)       + Fib(5)
         =   0   +   1     +        Fib(3)       + Fib(5)
         =       1         + Fib(1)+   Fib(2)    + Fib(5)
         =       1         + Fib(1)+Fib(0)+Fib(1) + Fib(5)
```

etc.

Compute Fibonacci Number – Tree Representation

The tree of calls for `Fib(6)`.



Do you see what is the problem here?

-25 calls !!!to compute F(6)-The problem is: Same function is repeated again and again. For example, F(1) has been computed 8 times!