Data Structures

(Feodor F. Dragan)
Department of Mathematics and Computer Science
Kent State University

Fall, 2000

CHAPTER 8

Recursion

Contents

- What is Recursion?
- Proofs by Mathematical Induction
- Numerical Applications
- Divide-and-Conquer Algorithms

What is Recursion?

- A recursive function directly or indirectly calls itself
- Very powerful problem-solving tool
- Many applications in CS

The Factorial Function

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

Iterative Definition

```
prod=1;
for (x=n; x>0; x--)
      prod *=x;
return(prod);
```

Recursive Definition

```
long fact( long n )
{   if (n<=1)
            return 1;
    else
            return( n* fact( n-1 ));
}
```

5!=5*4!
4!=4*3!
3!=3*2!
2!=2*1!
1!=1*0!
0!=1

Proofs by Mathematical Induction

- MI is an important proof technique used to establish theorems that hold for positive integers

  A proof by induction shows that the theorem is true for some simple cases and then shows how to extend the range of true cases indefinitely.

Theorem 1. *For any integer* $N \geq 1,$

$$\sum_{i=1}^{N} i = 1 + 2 + \cdots + N = \frac{N(N+1)}{2} \qquad (1)$$

Proof:

- Equation (1) is true for $N = 1$

In a proof by induction, the basis is the easy case that can be shown by hand.

- Suppose that it is true for all $N,\ 1 \leq N \leq k$
- We will prove that it is true for $N = k + 1$

The inductive hypothesis assumes that the theorem is true for some arbitrary case.

We need to prove that, under this assumption, theorem is true for the next case.

$$\sum_{i=1}^{k+1} i = (k+1) + \sum_{i=1}^{k} i =$$

$$(k+1) + \frac{k(k+1)}{2} = \frac{(k+1)(k+2)}{2}$$

Theorem 2. *For any integer* $N \ge 1$,

$$\sum_{i=N}^{1} (-1)^{N-i} i^2 = N^2 - (N-1)^2 + (N-2)^2 - \cdots (-1)^{N-1} = \frac{N(N+1)}{2} \qquad (2)$$

Proof:

- **Bases:** Equation (2) is true for $N = 1$
- **Progress:** Suppose that it is true for $N = k$, i.e.,

$$\sum_{i=k}^{1} (-1)^{k-i} i^2 = \frac{k(k+1)}{2}$$

We must show that $(N = k+1)$

$$\sum_{i=k+1}^{1} (-1)^{k+1-i} i^2 = \frac{(k+1)(k+2)}{2}$$

Indeed, $$\sum_{i=k+1}^{1} (-1)^{k+1-i} i^2 = (k+1)^2 - \left\{ k^2 - (k-1)^2 + \cdots \right\}$$

$$= (k+1)^2 - \left\{ \sum_{i=k}^{1} (-1)^{k-i} i^2 \right\}$$

$$= (k+1)^2 - \frac{k(k+1)}{2}$$

$$= \frac{(k+1)(k+2)}{2}$$

Recursion

```
// Recursive routine to compute sum of first n integers  (bad use of recursion)
long s( int n)
{
        if ( n==1 )
           return 1;
        else
           return s( n-1 ) + n;
}
```

```
s(5)=5+s(4)
      s(4)=4+s(3)
            s(3)=3+s(2)
                  s(2)=2+s(1)
                        s(1)=1
```

- How it works
  - Function calling and return sequence are stack operations
  - Always have at least one base case that can be solved without recursion
  - Any recursive call must progress toward a base case
- Recursion can always be removed by using explicit stack
  - Recursive algorithm can be rewritten by iteration (to save space); see Fibonacci series (next slide)
- Too much recursive calls can be dangerous
  - Do not use recursion as a substitute for a simple loop

Fibonacci Sequences: Another bad example

$$f(n) = \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ f(n-1) + f(n-2) & \text{if } n \geq 2 \end{cases}$$

```
if (n<=1)
        return ( n ) ;
f1=0;
f2=1;
for ( i=2; i<=n; i++ )
{
        x=f1;
        f1=f2;
        f2=x+f1;
}
 return ( f2 );
```

```
// compute the nth Fibonacci number
// bad example of recursion use

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

1. f(7)=f(5)+f(6)
2. f(6)=f(4)+f(5)
3. f(5)=f(3)+f(4)
4. f(4)=f(2)+f(3)
5. f(3)=f(1)+f(2)
6. f(2)=f(1)+f(0)
7. 1   0

- The major drawback of this routine is redundant calls
  - each call to $f(n\text{-}1)$ or $f(n\text{-}2)$ makes a call to $f(n\text{-}3)$ ➔3 calls to $f(n\text{-}3)$
  - each call to $f(n\text{-}2)$ or $f(n\text{-}3)$ makes a call to $f(n\text{-}4)$ ➔ 5 calls to $f(n\text{-}4)$
  - ... .
- In general, let $C(N)$ be the number of calls to *fib* during the evaluation of *fib(n)*
- Then,

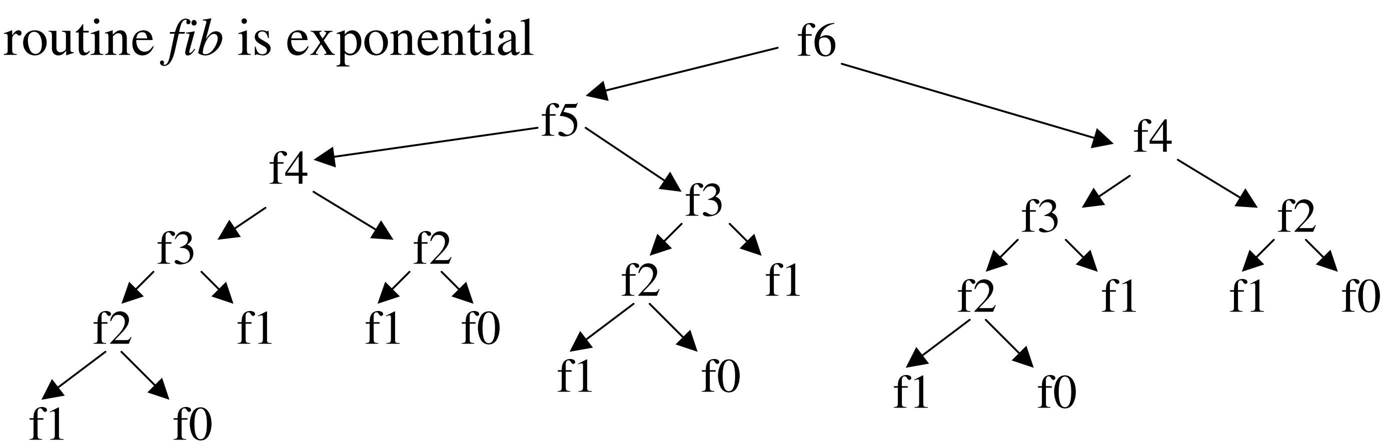
$$C(1) = C(0) = 1$$

$$C(N) = C(N\text{-}1) + C(N\text{-}2) + 1$$

By induction, we can easily verify that for $N>2$, the solution to this recurrence relation is

$$C(N) = f(N+2) + f(N\text{-}1) - 1$$

- The recursive routine *fib* is exponential



Four Basic Rules of Recursion

- *m* **Base case(s):** You must always have some base cases, which can be solved without recursion.
- **Making Progress:** For the cases that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case.
- **Design rule:** Assume that all recursive calls work and correctly perform current needed calculations (... to prove correctness of the algorithm by induction).
- **Compound Interest Rule:** Never duplicate work by solving the same instance of a problem in separate recursive calls.

Binary Search

- Search is a common activity in computing
- Linear or sequential search is not efficient for large files
- For large data set, binary search is more suitable

```
binsearch( a, x, low, high)
int a[];
int x;
int high, low;
{
        int mid;
        if ( low > high )
          return( -1);
        mid=(low+high)/2;
        return( x==a[mid] ? mid : x< a[mid] ? binsearch (a, x, low, mid-1) :
                binsearch (a, x, mid+1, high) );
}
```

- To make it more efficient, variables *a* and *x* can be eliminated and externally referenced by the main program. Therefore the function would be *binsearch (low, high)*
- An important requirement for a recursive algorithm is to avoid infinite sequence of calls. To do this one or more nonrecursive exit arguments should be presented in the program.

Indirect recursion

- A recursive function may call itself indirectly via another function
- More than two routines may participate in a *recursive chain*

```
a ( formal parameters )
{
            .
            .
            .
     b ( arguments );
}
```

```
b ( formal parameters )
{
            .
            .
            .
     a ( arguments );
}
```

Numerical Applications

Modular Exponentiation

- How to compute $X^N \pmod{P}$ efficiently?
- One way is

```
result =1;
for (i=0; i<N; i++) { result = (result * X) mod P }
```

- If N is huge, doing N multiplications is impractical
- A faster algorithm is based on the following recurrence relation

$$X^N = \begin{cases} (X \cdot X)^{\lfloor N/2 \rfloor} & \textit{if } N \textit{ is even} \\ X \cdot (X \cdot X)^{\lfloor N/2 \rfloor} & \textit{if } N \textit{ is odd} \end{cases}$$

Modular Exponentiation Routine

```
// Compute x^n ( mod p ).
// HugeInt: must have copy constructor, operator=,
//              conversion from int, *, /, %, ==, and !=
//              Assumes that p is not zero and power( 0, 0, p ) is 1
template <class HugeInt>
HugeInt power( const HugeInt & x, const HugeInt & n , const HugeInt & p )
{
        if ( n == 0 )
                return 1;
        HugeInt tmp = power( ( x * x ) % p, n / 2, p );
        if( n % 2 != 0 )
                tmp = ( tmp * x ) % p;
        return tmp;
}
```

- Time complexity
  - Let *M(N)* be the # of multiplications used by *power*
  - $M(N) \leq M\lfloor N/2 \rfloor + 2$ Why?
    - If N is even, *1*+ those done recursively
    - If N is odd, *2*+ those done recursively
  - *M(0)=0*
  - Using halving principle, on average *M(N)=(3/2)log N*
- Exponentiation can be done in logarithmic number of multiplications

Greatest Common Divisor (gcd)

- The largest integer that divides two integers?
- $\gcd(A,B) \equiv \gcd(A-B,B)$ Way?
  - if *D* divides *A* and *B,* it must divide *A*-*B*
  - if *D* divides *A*-*B* and *B,* it must divide *A*
- **Euclid's Algorithm:**
  - repeatedly subtract *B* from *A* until *A<B (A=A-B)*
  - repeatedly subtract *A* from *B* until *B<A (B=B-A)*
  - repeat again until *B=0*
  - then *gcd(A,0)=A*
- Example
  - *A=18, B=4*
  - *A=18*➔ *A=14* ➔ *A=10*➔ *A=6*➔ *A=2*
  - *B=4*➔ *B=2*➔ *B=0*
  - *gcd(2,0)=2* ➔ *gcd(A,B)=2*

- In a more efficient way

$$gcd\ (A,\ B) = gcd\ (B,\ (A\ mod\ B)) \Rightarrow gcd\ (18,4) = gcd\ (4,2) = 2$$

- This recursive equation gives an algorithm

```
// Returns the greatest common divisor of a and b

template <class HugeInt>
HugeInt gcd( const HugeInt & a, const HugeInt & b )
{
        if ( b==0 )
            return a;
        else
            return gcd( b, a%b);
}
```

- The number of recursive calls is proportional to the logarithm of *A*

  since *A mod B* is < *A/2* (consider two cases: *B* ≤ *A/2 and B>A/2*)

Divide-and-Conquer Algorithms

- Divide: Smaller problems are solved recursively (except, of course, base cases)
- Conquer: Solution to the original problem is formed from the solutions of the subproblems
- Inherently recursive

Maximum Contiguous Subsequence Sum

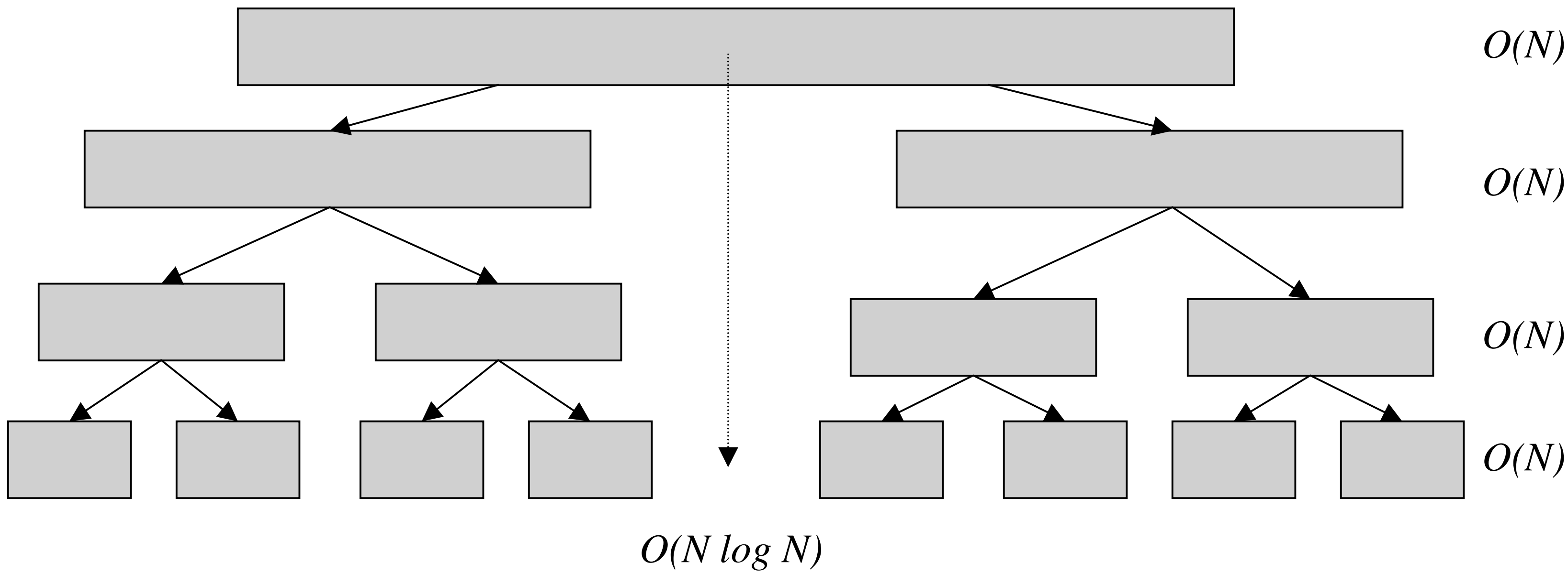
- Divide the sequence into two halves
- The MCSS can reside
  1. in the 1st half   2. in the 2nd half   3. in both
- In case 3, the last element of the 1st half and the first element of the 2nd half are parts of MCSS

  using two N/2 loops starting from these points, we can find two MCSS from each half, and then add them ➔ *O(N)*
- Cases 1 and 2 can be done recursively
- Choose the max among the three cases

```
// Recursive O( N log N ) maximum contiguous subsequence sum algorithm.
template <class Comparable>
Comparable maxSubSum( const vector<Comparable> & a, int left, int right )
{
        Comparable maxLeftBorderSum = 0, maxRightBorderSum = 0;
        Comparable leftBorderSum = 0, rightBorderSum = 0;
        int center = ( left + right ) / 2;
        if( left == right ) // Base Case.
            return a[ left ] > 0 ? a[ left ] : 0;
        Comparable maxLeftSum = maxSubSum( a, left, center );
        Comparable maxRightSum = maxSubSum( a, center + 1, right );
        for( int i = center; i >= left; i-- )
        {
                leftBorderSum += a[ i ];
                if( leftBorderSum > maxLeftBorderSum )
                    maxLeftBorderSum = leftBorderSum;
        }
        for( int j = center + 1; j <= right; j++ )
        {
                rightBorderSum += a[ j ];
                if( rightBorderSum > maxRightBorderSum )
                    maxRightBorderSum = rightBorderSum;
        }
        return max3( maxLeftSum, maxRightSum, maxLeftBorderSum + maxRightBorderSum );
}
// Public driver.
template <class Comparable>
Comparable maxSubsequenceSum( const vector<Comparable> & a )
{ return a.size( ) > 0 ? maxSubSum( a, 0, a.size( ) - 1 ) : 0; }
```

- Time analysis ➔ *O(N log N)*

Assuming N is a power of 2, the solution to the equation $T(N) = 2\,T(N/2) + N$, with initial condition $T(1) = 1$ is $T(N) = N \log N + N$



Trace of recursive calls for recursive maximum contiguous subsequence sum algorithm

Homework Problem 8.27 (a), page 317