

Preconditions and Postconditions

A way to specify what the function accomplishes

- A precondition is “a requirement of a function upon its argument ... it must be true for the function to perform its action correctly.”
- A postcondition is a “promise about the return value”.

“... it forces you to think about what your function requires.”

Recursion

Breaking Problems Into Smaller Subproblems of the
same problem

**THEN COMBINE THE SUBPROBLEMS TO SOLVE
THE ORIGINAL PROBLEM**

Recursive Mathematical Formulas

Benefits of recursion

- clear
- short

$f(0) = 1$ } base case (or simple case which has an easy to determine answer)

$f(x) = x * f(x-1)$ } recursive definition (a simpler problem of the same form)

C++ Implementation

```
int factorial( int x )  
{  
    if( x == 0 ) } base case  
        return 1;  
    else  
        return x * factorial( x - 1 ); } recursive case  
}
```

It may be hard to understand how this step works. When you design a recursive algorithm you need to just "believe" the recursive step works!

Note: Don't use recursion to implement a loop! This is only for illustrative purposes.

C++'s simulation of recursion is sometimes inefficient.

Recursive Paradigm:

1. There must be ***simple cases*** which can be solved “***easily***”
- 2 For non simple cases, the problem can be solved by breaking the problem into simpler problems of the same form, which can be combined to solve the original problem.

Recursively adding the numbers 1 thru n

$$S(1) = 1$$

$$S(n) = S(n-1) + n$$

Recursively add the numbers from 1 to n

```
int Sum(int n)
{
  // pre : n >= 0
  if (n==0) return 0; } base case
  return n+Sum(n-1); } recursive case
}
```

Stack

return 2 + Sum(1)
return 3 + Sum(2)
return 4 + Sum(3)
return 5 + Sum(4)

Sum(1)

Sum(2)

Sum(3)

Sum(4)

Sum(5)

Too many recursive calls can use up all the space in the stack and cause your program to crash or hang.

Note: Don't use recursion to implement a loop! This is only illustrative purposes.

Dictionary Definition:

Recursion

See "Recursion".

Recursion Examples

- Recursive printing:
 - if $n \leq 9$ print n
 - else print $n/10$ then print $n\%10$ (last digit)
 - (More generally ... change of base)
$$624 = 6 * 10^2 + 2 * 10^1 + 4 * 10^0$$
- Recursive Max Contiguous Subsequence
 - Simpler than $O(n)$ algorithm
 - $O(n \log n)$
- Two of recursive sorting algorithms (later)

Can we write a recursive program
to change the base of a number?

Change 65 to base 3

$$65_{10} = 2102_3$$
$$= 3 * (2 * 3^2 + 1 * 3^1 + 0 * 3^0)_{10} + 2 * 3^0_{10}$$

$$65_{10} \quad \text{👉} \quad 65_{10} / 3 = 21 \quad 65_{10} \bmod 3 = 2$$

$$n_b = (c_n c_{n-1} c_{n-2} \cdots c_1 c_0)_b = c_n b^n + b^{n-1} c_{n-1} + b^{n-2} c_{n-2} + \cdots + b^1 c_1 + b^0 c_0$$

$$n_b \bmod b = (c_n b^n + b^{n-1} c_{n-1} + b^{n-2} c_{n-2} + \cdots + b^1 c_1 + b^0 c_0) \bmod b = c_0$$

$$n_b / b = (c_n b^{n-1} + b^{n-2} c_{n-1} + b^{n-3} c_{n-2} + \cdots + b^0 c_1)$$

Recursively change the base of a number: ChangeBase(n,b)

$$10 = 1010_2 = 101_3 = 20_5$$

Base Case: if $n < b$, $\text{ChangeBase}(n,b) = n$

Recursive Case: $\text{ChangeBase}(n,b) = \text{ChangeBase}(n/b, b) \cdot n \% b$

Example: $\text{ChangeBase}(15,3) = \text{ChangeBase}(5,3) \cdot 20$
 $\text{ChangeBase}(5,3) = \text{ChangeBase}(1,3) \cdot 2$
 $\text{ChangeBase}(1,3) = 1$

Recursively change the base of a number

What if we changed
the order here?

```
// Print n in base base, recursively.  
// Precondition: n >= 0, 2 <= base <= MAX_BASE.  
// Postcondition:  
const string DIGIT_TABLE = "0123456789abcdef";  
const int MAX_BASE = DIGIT_TABLE.length( );  
  
void printIntRec( int n, int base )  
{  
    if( n >= base )  
        printIntRec( n / base, base );  
    cout << DIGIT_TABLE[ n % base ];  
}
```

} non base case - recursive case

What if the number is negative, or the base is out of range?

```
const string DIGIT_TABLE = "0123456789abcdef";
const int MAX_BASE = DIGIT_TABLE.length( );
void printIntRec( int n, int base )
{
    if( base <= 1 || base > MAX_BASE )
        cerr << "Cannot print in base " << base << endl;
    else {
        if( n < 0 )
        {
            cout << "-";
            n = -n;
        }

        if( n >= base )
            printIntRec( n / base, base );
        cout << DIGIT_TABLE[ n % base ];
    }
}

int main( )
{
    for( int i = 0; i <= 17; i++ )
    {
        printInt( -1000, i );
        cout << endl;
    }
    return 0;
}
```

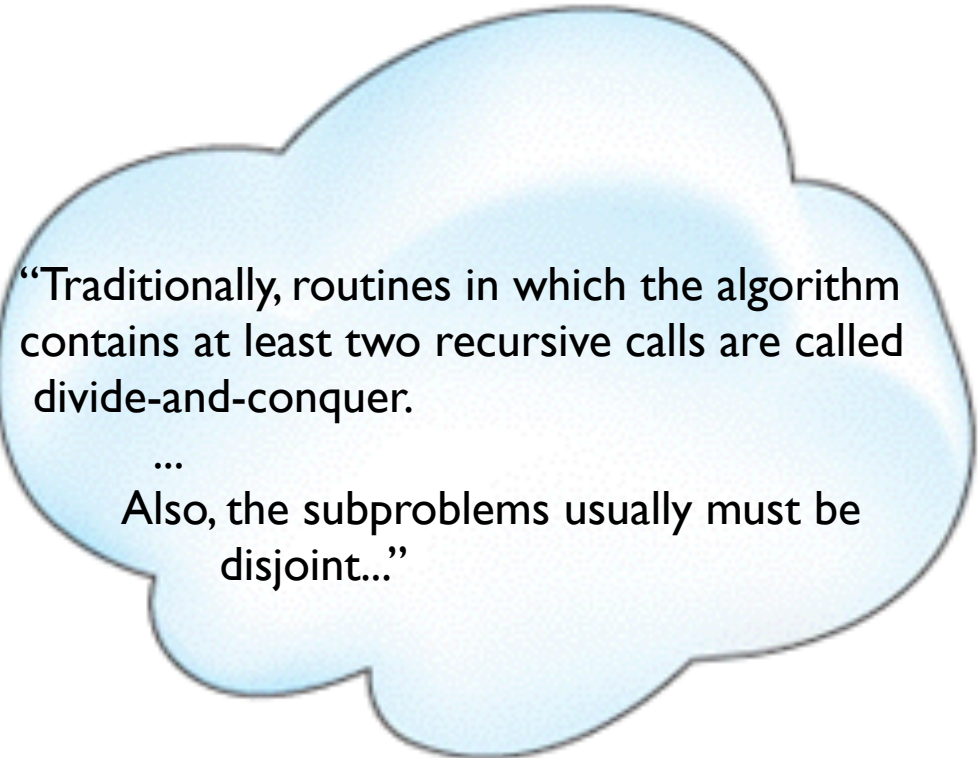
Driver Routine: test validity of first call and sets up recursive calls

```
void printInt( int n, int base )
{
    if( base <= 1 || base > MAX_BASE )
        cerr << "Cannot print in base " << base << endl;
    else
    {
        if( n < 0 )
        {
            cout << "-";
            n = -n;
        }
        printIntRec( n, base );
    }
}
```

```
int main( )
{
    for( int i = 0; i <= 17; i++ )
    {
        printInt( -1000, i );
        cout << endl;
    }
    return 0;
}
```

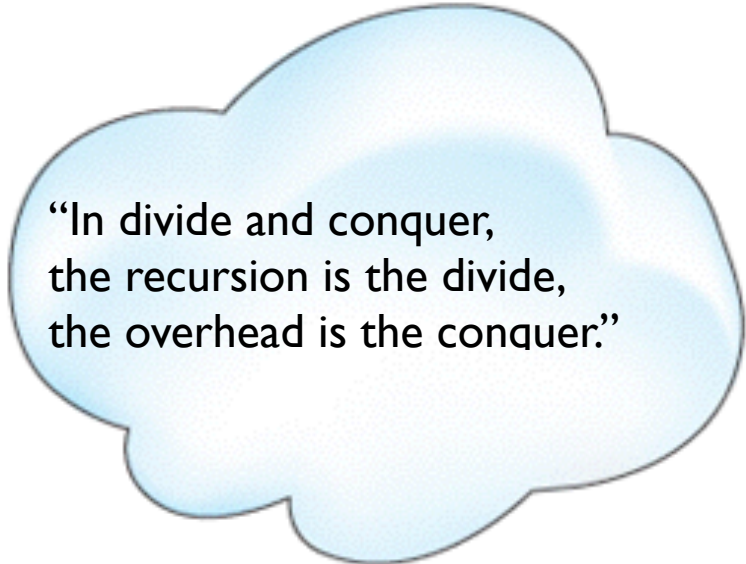
DIVIDE AND CONQUER

- Divide: smaller problems - solved recursively
- Conquer: Solution from original problem is formed from the solutions to the subproblems



“Traditionally, routines in which the algorithm contains at least two recursive calls are called divide-and-conquer.

...
Also, the subproblems usually must be disjoint...”



“In divide and conquer, the recursion is the divide, the overhead is the conquer.”

Can we find a recursive maximum contiguous subsequence algorithm?

Max Contiguous Subsequence Problem

Definition:

- Given sequence A_1, \dots, A_n of numbers find i and j such that $A_i + \dots + A_j$ is maximal.
- $A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9$
- 1, 2, -4, 1, 2, -1, 4, -2, 1
- Max subsequence is 1,2,-1,4 whose sum is 6

RECURSIVE MAXIMUM CONTIGUOUS SUBSEQUENCE ALGORITHM

Case 1: The max resides entirely in the first half

Case 2: The max resides entirely in the second half

Case 3: The max begins in the first half but ends in
the second half

Running time? $O(n \log(n))$ time


```
int maxSumRec( const vector<int> & a, int left, int right )
{
    /* 1 */
    /* 2 */
    /* 3 */
    /* 4 */
    /* 5 */
    /* 6 */
    /* 7 */
    /* 8 */
    /* 9 */
    /*10*/
    /*11*/
    /*12*/
    /*13*/
    /*14*/
    /*15*/
    /*16*/
    /*17*/
    /*18*/
    /*19*/
}
```

Base Case

Recursively find maximum sum subsequence
in the left half and in the right half

Iteratively find the largest sum
that starts from the center and goes left

Iteratively find the largest sum
that starts from the center + 1 and goes right

Return the largest of the three sums

```

int maxSumRec( const vector<int> & a, int left, int right )
{
    /* 1*/      if( left == right )  // Base case
    /* 2*/          if( a[ left ] > 0 )
    /* 3*/              return a[ left ];
    /* 4*/              else
    /* 5*/                  return 0;

    /* 5*/      int center = ( left + right ) / 2;
    /* 6*/      int maxLeftSum  = maxSumRec( a, left, center );
    /* 7*/      int maxRightSum = maxSumRec( a, center + 1, right );

    /* 8*/      int maxLeftBorderSum = 0, leftBorderSum = 0;
    /* 9*/      for( int i = center; i >= left; i-- )
    /*10*/          {
    /*11*/              leftBorderSum += a[ i ];
    /*12*/              if( leftBorderSum > maxLeftBorderSum )
    /*13*/                  maxLeftBorderSum = leftBorderSum;
    /*14*/          }

    /*15*/      int maxRightBorderSum = 0, rightBorderSum = 0;
    /*16*/      for( int j = center + 1; j <= right; j++ )
    /*17*/          {
    /*18*/              rightBorderSum += a[ j ];
    /*19*/              if( rightBorderSum > maxRightBorderSum )
    /*20*/                  maxRightBorderSum = rightBorderSum;
    /*21*/          }

    /*22*/      return max3( maxLeftSum, maxRightSum,
    /*23*/                      maxLeftBorderSum + maxRightBorderSum );
}

```

Driver Routine: test validity of first call and sets up recursive calls

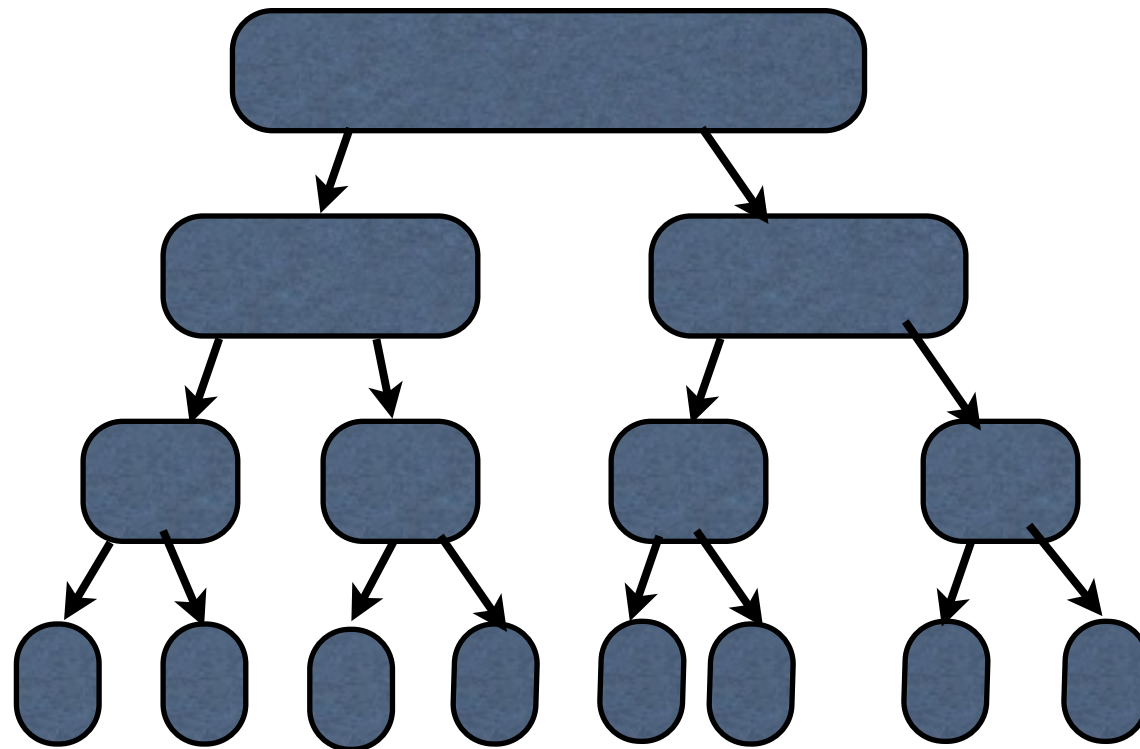
```
* Driver for divide-and-conquer maximum contiguous
* subsequence sum algorithm.
*/
int maxSubSum3( const vector<int> & a )
{
    return a.size( ) > 0 ? maxSumRec( a, 0, a.size( ) - 1 ) : 0;
}
```

```
int main( )
{
    vector<int> a = {4, -3, 5, -2, -1, 2, 6, -2};
    int maxSum;

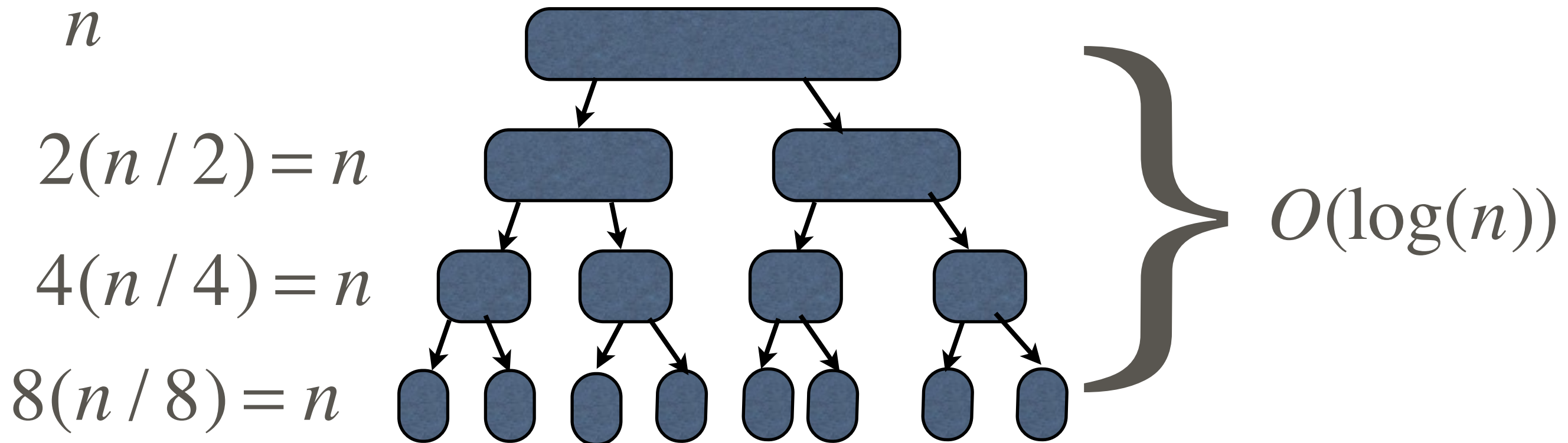
    maxSum = maxSubSum3( a );
    cout << "Max sum is " << maxSum << endl;

    return 0;
}
```

Finding the maximum contiguous subsequence sum of 4,-3,5,-2,-1,2,6,-2



Discovering the Running Time ($n=8$)



Analysis of Basic Divide and Conquer Recurrence with linear additional work

$$T(1) = 1$$

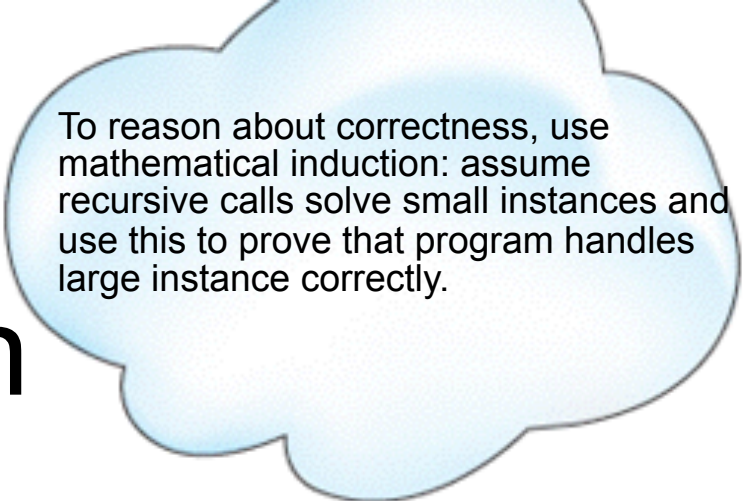
$$T(n) = 2T(n/2) + n$$

$$T(n) = 2(2T(n/4) + n/2) + n$$

...

$$T(n) = 2^k T(n/2^k) + kn$$

If $n=2^k$ $T(n) = n + n \log(n)$



To reason about correctness, use mathematical induction: assume recursive calls solve small instances and use this to prove that program handles large instance correctly.

4 Rules of Recursion

- Base Case: Always have at least one case that can be solved without recursion
- Make Progress: All recursive calls must make progress toward the base case
- “you gotta believe”: Always assume that the recursive call works
- Compound interest rule: Never duplicate work by solving the same instance of a problem in separate recursive calls

"To understand recursion,
you must understand recursion."

When **NOT** to use recursion

- Sometimes the same problem instance occurs in many recursive calls, e.g. Fibonacci series:
 - $\text{Fib}(0) = 0; \text{Fib}(1) = 1;$
 - $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$ for $n \geq 2$
- Recursive formulation: exponential running time!
- In such cases it's better to use dynamic programming, computing small instances first and remembering the results.

Recursively Computing the Fibonacci Sequence

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

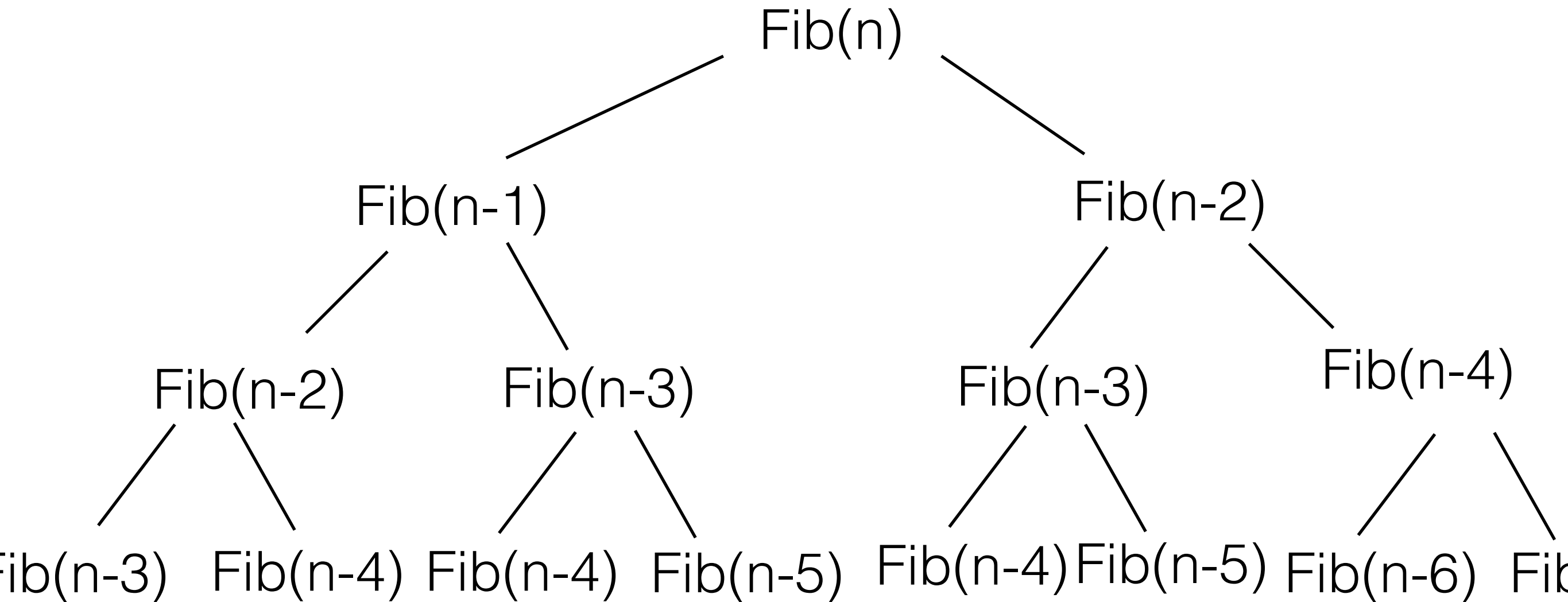
```
int main( )  
{  
    cout << "fib( 7 ) = " << fib( 7 ) << endl;  
    cout << "fibonacci( 7 ) = " << fibonacci( 7 ) << endl;  
    return 0;  
}
```

Running time?

Why did this take so long?

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

Fib (200) takes will finish after people have projected the sun turns to a red giant (a dying star)



**I keep repeating the same calculations again
and again and again ...**

Using Dynamic Programming to Compute the Fibonacci Sequence

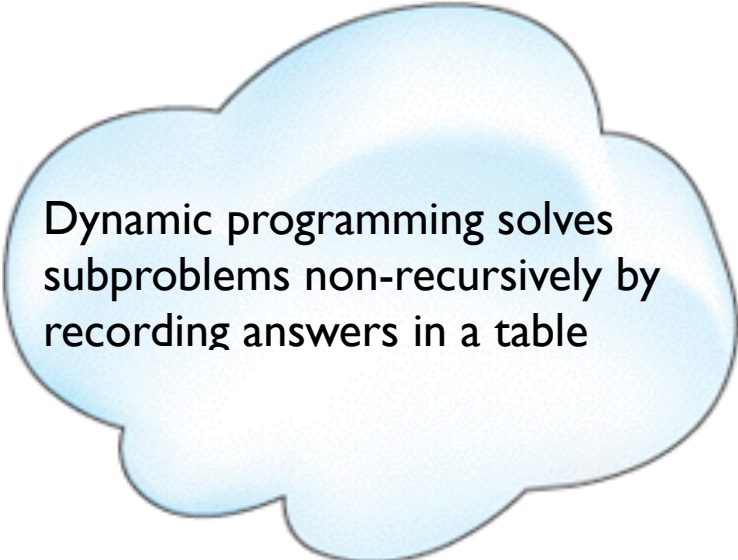
```
int DynamicProgrammingFib(int n)
{
    vector<int> f(n+1);

    f[1] = f[0] = 1;

    for (int i = 2; i <= n; i++)
        f[i] = f[i-1] + f[i-2];

    return f[n];
}
```

Running time?



Dynamic programming solves subproblems non-recursively by recording answers in a table

When to think twice about using Recursion

- When recursive solution has same $O()$ running time as iterative solution:
 - Recursive solution is often cleaner and more elegant
 - Iterative solution usually runs faster by some constant factor, since it avoids the overhead costs of function calls
- Tail recursion
 - Single recursive call at end of function
 - Usually easy to recode iteratively

Stack Memory Management

- At run time, memory for local variables, parameters, return values is stored on stack of activation records
- Each activation record (stack frame) stores parameters, locals, return value, return address (ip of instruction to execute after return from the function), plus some bookkeeping info needed to restore the stack
 - Dynamic link: address of caller's stack frame
 - Sometimes additional info for finding non-local variables
- Stack frame pushed when function is called, popped when function returns

Recursive Memory Management

- Several stack frames may be active for the same function.

AR for fact:

- Example

```
int fact(int n)
{ int tmp1, tmp2;
  if (n==0) return 1;
  tmp1 = fact(n-1);
  tmp2 = n*tmp1;
  return tmp2;
}
```

n
tmp1
tmp2
RP
dynamic link

