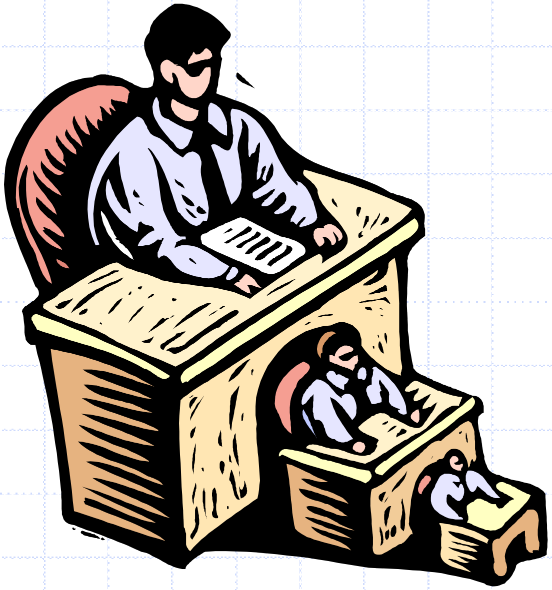
Using Recursion



The Recursion Pattern

- **Recursion**: when a method calls itself
- Classic example--the factorial function:
  - n! = 1· 2· 3· ··· · (n-1)· n
- Recursive definition:

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

- As a C++ method:

```
// recursive factorial function
recursiveFactorial( n)
   if  (n  ==  0)  return  1;     // basis case
   else return  n  *  recursiveFactorial(n- 1);  // recursive case
```

Linear Recursion

- Test for base cases
  - Begin by testing for a set of base cases (there should be at least one).
  - Every possible chain of recursive calls must eventually reach a base case, and the handling of each base case should not use recursion.
- Recur once
  - Perform a single recursive call
  - This step may have a test that decides which of several possible recursive calls to make, but it should ultimately make just one of these calls
  - Define each possible recursive call so that it makes progress towards a base case.

Example of Linear Recursion

Algorithm LinearSum(*A, n*):

Input:

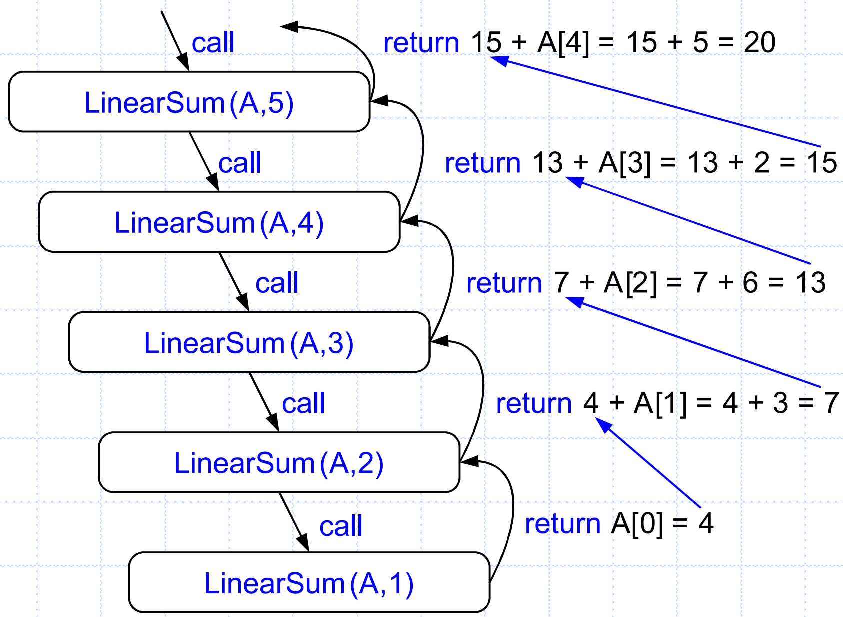
A integer array *A* and an integer *n* = 1, such that *A* has at least *n* elements

Output:

The sum of the first *n* integers in *A*

if *n* = 1 **then**
  return *A*[0]
else
  return LinearSum(*A, n* - 1) + *A*[*n* - 1]

Example recursion trace:



Reversing an Array

Algorithm ReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

if $i < j$ **then**

Swap $A[i]$ and $A[j]$

ReverseArray(A, $i + 1$, $j - 1$)

return

Defining Arguments for Recursion

- In creating recursive methods, it is important to define the methods in ways that facilitate recursion.
- This sometimes requires we define additional paramaters that are passed to the method.
- For example, we defined the array reversal method as ReverseArray(A, i, j), not ReverseArray(A).

Computing Powers

- The power function, $p(x,n)=x^n$, can be defined recursively:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x, n-1) & \text{else} \end{cases}$$

- This leads to an power function that runs in O(n) time (for we make n recursive calls).
- We can do better than this, however.

Recursive Squaring

- We can derive a more efficient linearly recursive algorithm by using repeated squaring:

$$p(x,n) = \begin{cases} 1 & \text{if } n = 0 \\ x \cdot p(x,(n-1)/2)^2 & \text{if } n > 0 \text{ is odd} \\ p(x,n/2)^2 & \text{if } n > 0 \text{ is even} \end{cases}$$

- For example,

$2^4 = 2^{(4/2)2} = (2^{4/2})^2 = (2^2)^2 = 4^2 = 16$

$2^5 = 2^{1+(4/2)2} = 2(2^{4/2})^2 = 2(2^2)^2 = 2(4^2) = 32$

$2^6 = 2^{(6/2)2} = (2^{6/2})^2 = (2^3)^2 = 8^2 = 64$

$2^7 = 2^{1+(6/2)2} = 2(2^{6/2})^2 = 2(2^3)^2 = 2(8^2) = 128.$

Recursive Squaring Method

Algorithm Power(x, n):
    Input: A number x and integer $n = 0$
    Output: The value x^n
    if $n = 0$ **then**
        return 1
    if n is odd **then**
        y = Power(x, $(n - 1)/2$)
        return $x \cdot y \cdot y$
    else
        y = Power(x, $n/2$)
        return $y \cdot y$

Analysis

Algorithm Power(x, n):
    Input: A number x and integer $n = 0$
    Output: The value x^n
    if $n = 0$ **then**
        return 1
    if n is odd **then**
        y = Power(x, $(n - 1)/2$)
        return $x \cdot y \cdot y$
    else
        y = Power(x, $n/2$)
        return $y \cdot y$

Each time we make a recursive call we halve the value of n; hence, we make log n recursive calls. That is, this method runs in O(log n) time.

It is important that we use a variable twice here rather than calling the method twice.

Tail Recursion

- Tail recursion occurs when a linearly recursive method makes its recursive call as its last step.
- The array reversal method is an example.
- Such methods can be easily converted to non-recursive methods (which saves on some resources).
- Example:

Algorithm IterativeReverseArray(A, i, j):

Input: An array A and nonnegative integer indices i and j

Output: The reversal of the elements in A starting at index i and ending at j

while $i < j$ **do**

Swap $A[i]$ and $A[j]$

$i = i + 1$

$j = j - 1$

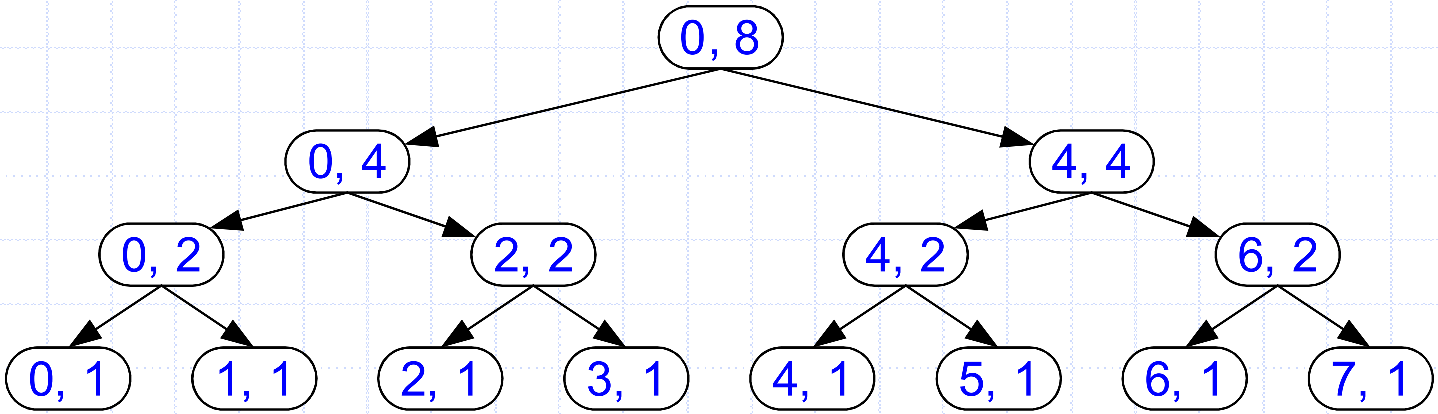
return

Another Binary Recusive Method

- Problem: add all the numbers in an integer array A:

Algorithm BinarySum(A, i, n):
Input: An array A and integers i and n
Output: The sum of the n integers in A starting at index i
if $n = 1$ **then**
return $A[i]$
return BinarySum(A, i, $n/2$) + BinarySum(A, $i + n/2$, $n/2$)

- **Example trace:**



Computing Fibonacci Numbers

- Fibonacci numbers are defined recursively:

  $F_0 = 0$

  $F_1 = 1$

  $F_i = F_{i-1} + F_{i-2}$ for $i > 1$.

- Recursive algorithm (first attempt):

Algorithm BinaryFib(k):

Input: Nonnegative integer k

Output: The kth Fibonacci number F_k

if K = 0 then return 0

if $k = 1$ **then return** *1*

else

return BinaryFib(k - 1) + BinaryFib(k - 2)

Analysis

- Let n_k be the number of recursive calls by BinaryFib(k)
  - $n_0 = 1$
  - $n_1 = 1$
  - $n_2 = n_1 + n_0 + 1 = 1 + 1 + 1 = 3$
  - $n_3 = n_2 + n_1 + 1 = 3 + 1 + 1 = 5$
  - $n_4 = n_3 + n_2 + 1 = 5 + 3 + 1 = 9$
  - $n_5 = n_4 + n_3 + 1 = 9 + 5 + 1 = 15$
  - $n_6 = n_5 + n_4 + 1 = 15 + 9 + 1 = 25$
  - $n_7 = n_6 + n_5 + 1 = 25 + 15 + 1 = 41$
  - $n_8 = n_7 + n_6 + 1 = 41 + 25 + 1 = 67.$
- Note that n_k at least doubles every other time
- That is, $n_k > 2^{k/2}$. It is exponential!

Analysis

$T(N) = T(N-1) + T(N-2) + 1$

$= [T(N-2)+T(N-3)+1]+[T(N-3)+T(N-4)+1]+1$

$= T(N-2) + T(N-3) + T(N-3) + T(N-4) + 3$

If we repeat the recurrence, we're going to get 8 T's on level 3. Then 16, 32, and so on...

- So we get 2^k T's at level k.
- To get down T(N-1) to the base case T(2), we'll need to go to level k = N-2.
- We'll have 2^N-2 T's there, so T(N) = O(2^N).

GCD

Function definition:

gcd(x,y) = x, if y = 0
= gcd(y, reminder(x,y)) if y > 0

function gcd is:
input: integer x, integer y such that x >= y and y >= 0

1. if y is 0, return x
2. otherwise, return [gcd(y, (remainder of x/y))]

end gcd

GCD Analysis

To see why notice that: GCD(a,b) = GCD(b,a mod b) = GCD(a mod b, b mod (a mod b)). Now since a mod b = r such that a = bq + r, it follows that r<b, so a>2r. So every two iterations, the larger number is reduced by a factor of 2 (at least) so there are at most O(lgn) iterations.

Examples- Recursion

```
void myFunction( int counter)
{
        cout<<"hello"<<counter<<endl;
        if ( counter ==0 ) {
            myFunction(--counter);
            cout<<counter<<endl;
        }
        return;

}
```

Examples- Recursion 1

- What will it do if Counter is 8 ?
- What will it do if counter is set to -8 ?
  - What to do about it ?

Examples- Recursion 2

- Write a recursive program to find prime number.
- The main is as follows:
- int main(int argc, char** argv) {
- int b;
- int n;
- n = 13;
- b = isPrime(n,2);
- cout << b << '\n';
- return 0;
- }

- isPrime(n, p) {
-         if (n == p) { return true; }
-         if (n % p == 0) { return false }
-         return(isPrime(n, p+1));
- }

Examples- Recursion 2

- bool isPrime(int p, int i) {
-     if (I == p) return 1; //or better if (i*i>p) return 1;
-     if (p % i == 0) return 0;
-     return isPrime(p, i + 1);
- }

Exercise - Recursion 3

- Write a recursive program to find if a string is palindrome or not ?
- The main is as follows:
- int main() {
-     cout << "Enter a string: ";
-     char str[20];

-     cin.getline(str, 20, '\n');
-     cout << "The entered string " << ((palindrome(str, strlen(str) + 1)) ? "is" : "is not") << " a Palindrome string." << endl;
-     return 0;
- }

Examples- Recursion 3