CSE 143
SUMMER 1998

Recursion
[Sections 6.1,6.3-6.7]

Recursion

- A recursive definition is one which is expressed in terms of itself
- Examples:
  - "A horse is a four-legged animal which is the progeny of two horses"
  - Compound interest: "The value after 10 years is equal to the interest rate times the value after 9 years."
  - An arithmetic expression is either a number, or two arithmetic expressions with a +,-,* or / between them

Computer Science Examples

- We've already seen recursive data structures:

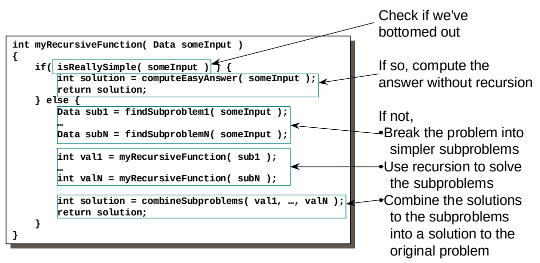
```
struct Node {
    int    data;
    Node   *link;
};
```

- Functions (and methods) are also allowed to be defined recursively
  - Recursive function: a function that calls itself (possibly indirectly)
  - Recursive functions are fundamental in computer science

The Essense of Recursive Computation

- Recursive functions only work when there's a well-defined notion of **making progress**
- If a function calls itself to solve a subproblem, two things had better happen:
  - The subproblem has to be simpler than the original problem
  - Some subproblems have to be so simple that you can solve them without recursion (bottoming out)

What to Look for



```
int myRecursiveFunction( Data someInput )
{
    if( isReallySimple( someInput ) ) {
        int solution = computeEasyAnswer( someInput );
        return solution;
    } else {
        Data sub1 = findSubproblem1( someInput );
        …
        Data subN = findSubproblemN( someInput );

        int val1 = myRecursiveFunction( sub1 );
        …
        int valN = myRecursiveFunction( subN );

        int solution = combineSubproblems( val1, …, valN );
        return solution;
    }
}
```

Check if we've bottomed out

If so, compute the answer without recursion

If not,
- Break the problem into simpler subproblems
- Use recursion to solve the subproblems
- Combine the solutions to the subproblems into a solution to the original problem

Factorial

- n! ("n factorial") can be defined in two ways:
  - Non-recursive definition
    - $n! = n * (n-1) * (n-2) \ldots * 2 * 1$
  - Recursive definition
    - $n! = \begin{cases} 1 & \text{if } n = 0 \\ n\,(n-1)! & \text{if } n > 0 \end{cases}$

The Recursive Factorial Function

```
int factorial( int n )
{
    if( n == 0 ) {
        return 1;
    } else {
        return n * factorial( n - 1 );
    }
}
```

- Note that the factorial function invokes itself.
- How can this work?

Automatic Variables

- Recall:
  - Static memory is set aside ahead of time
  - Dynamic memory is explicitly managed at runtime
- Automatic memory holds local variables and parameters to functions
  - Organized into a stack
  - Size changes dynamically at runtime
  - Managed automatically by the compiler
    - Block entered: push stack
    - Block exited: pop stack

Activation Records

- An activation record is a struct that holds information about a function call
  - Function parameters, local variables, other stuff
- A single function can have multiple activation records on the stack simultaneously
  - This is how recursive functions are possible
  - Each recursive call gets its own activation record

Example

```
int factorial( int n )
{
    if( n == 0 ) {
        return 1;
    } else {
        int sub = factorial( n - 1 );
        return n * sub;
    }
}

int main( void )
{
    int x = factorial( 4 );
    cout << "4! = " << x << endl;
}
```

Infinite Recursion

- Remember to always check for bottoming out first
  - Look for "base case"
  - If not…

```
int badFactorial( int n )
{
    int sub = badFactorial( n - 1 );
    if( sub == 1 ) {
        return 1;
    } else {
        return n * sub;
    }
}
```

- What is the value of **`badFactorial( 2 )`**?

Recursive Algorithms for Recursive Data Structures

- Many pointer-based data structures use the pointer-to-another-instance idea
- This recursive structure allows for elegant recursive algorithms

Printing a Linked List

```
void IntList::print()
{
    printFrom( front );
}

void IntList::printFrom( Node *from )
{
    if( from != NULL ) {
        cout << from->data << " ";
        printFrom( from->link );
    }
}
```

- How many recursive calls do we make when printing the list <1,2,3,4>?

Printing in Reverse Order

- Difficult problem: in our singly-linked list, links only point forward
- How can we modify recursive list printing to print in reverse?

Other Recursive List Operations

- Can also use recursion to, e.g. sum a list

```
int IntList::sumFrom( Node *from )
{
    if( from == NULL ) {
        return 0;
    } else {
        return from->data + sumFrom( from->link );
    }
}
```

- How would you modify this to count the length of a list? Add N to each element of a list? Find the maximum in the list?

Recursion: Not Just for Lists

- Works fine on arrays:

```
void selectionSort( int data[], int size, int from )
{
    if( (size - from) > 1 ) {
        int mi = 0;
        for( int idx = from + 1; idx < size; idx++ ) {
            if( data[ idx ] < data[ mi ] ) {
                mi = idx;
            }
        }
        swap( data[ from ], data[ mi ] );
        selectionSort( data, size, from + 1 );
    }
}
```

- Another example: trees

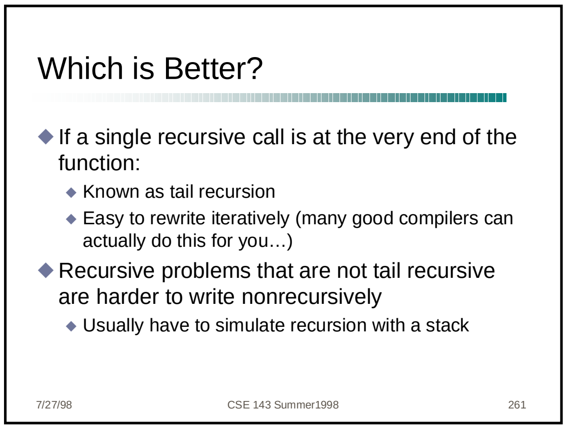
Challenge Problem

- What does this function do?

```
int mystery( int x )
{
    assert( x > 0 );
    if( x == 1 ) {
        return 0;
    } else {
        return 1 + mystery( x / 2 );
    }
}
```

Recursion vs. Iteration

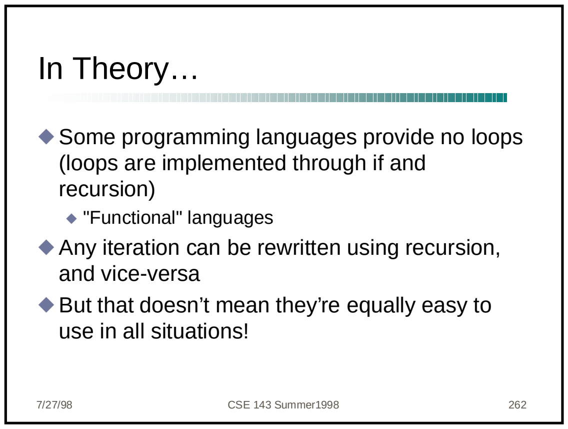
- When to use recursion?
  - Processing recursive data structures
  - Some algorithms can be expressed more elegantly in recursive form
    - “Divide & Conquer” algorithms
- When to use iteration instead?
  - Nonrecursive data structures
  - Problems without obvious recursive structure
  - Sometimes because it's faster



Which is Better?

- If a single recursive call is at the very end of the function:
  - Known as tail recursion
  - Easy to rewrite iteratively (many good compilers can actually do this for you…)
- Recursive problems that are not tail recursive are harder to write nonrecursively
  - Usually have to simulate recursion with a stack

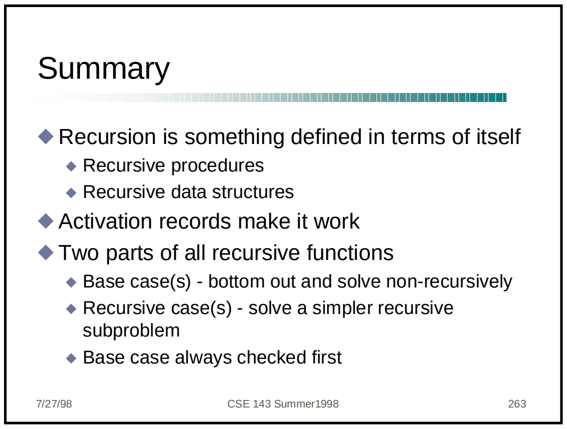
7/27/98 CSE 143 Summer1998 261



In Theory…

- Some programming languages provide no loops (loops are implemented through if and recursion)
  - "Functional" languages
- Any iteration can be rewritten using recursion, and vice-versa
- But that doesn't mean they're equally easy to use in all situations!

7/27/98 CSE 143 Summer1998 262



Summary

- Recursion is something defined in terms of itself
  - Recursive procedures
  - Recursive data structures
- Activation records make it work
- Two parts of all recursive functions
  - Base case(s) - bottom out and solve non-recursively
  - Recursive case(s) - solve a simpler recursive subproblem
  - Base case always checked first

7/27/98 CSE 143 Summer1998 263