Recursion

By:

Dr. Hakimzadeh and Adaikkalavan
Computer Science and Informatics
IU South Bend
Recursion

- A technique of defining a process in term of itself.

- More specifically, a function that calls itself.
Recursion

Example 1:

Factorial function is recursively defined as follows:

- \( \text{factorial} (n) = 1, \text{ if } n = 0 \)
- \( = n \times \text{factorial} (n-1), \text{ if } n > 0 \)

- \( 0! = \text{factorial} (0) = 1 \)
- \( 3! = \text{factorial} (3) = 3 \times \text{factorial}(2) \)
  - \( = 3 \times 2 \times \text{factorial}(1) \)
  - \( = 3 \times 2 \times 1 \times \text{factorial}(0) \)
  - \( = 3 \times 2 \times 1 \times 1 \)
Recursion

Example 1:
- Factorial function is recursively defined as follows:
  - factorial (n) = 1, if n = 0
  - = n * factorial (n-1), if n>0

```c
int factorial_recursive(int value) {
    if (value == 0) {
        return 1;
    } else {
        return(value * factorial_recursive(value - 1));
    }
}
```
Recursion

- Advantages of Recursion:
  - Powerful for solving certain types of problems (tree traversal, factorial, power, Fibonacci numbers.)
  - Reduces the complexity of the algorithms needed to solve certain type of problems.
  - Can be a more elegant solution.
  - Hides the details.
Recursion

Methods for verifying if a recursive functions works properly:
- The three-question method.

Ask yourself the following questions:

1) **Base question**
- Is there a non-recursive way out of the procedure or function, and does the routine work correctly for this “base” case? In other words can we stop the recursion?

2) **Smaller Caller question**
- Does each recursive call reduces the original problem? In other words are we getting closer to a solution?

3) **General Case question**
- Assuming that the recursive calls are working proper, does the whole function work correctly.
Recursion

Recursive Functions:

- Example 3:
  - A Recursive Function to determine if a value is in an array.

```c
bool linear_search_recursive(int array[], int value, int startIndex, int endIndex)
{
    if (array[startIndex] == value)    // base case, value is found.
        return (true);
    else {
        if (startIndex == endIndex) // base case, value is not in the list
            return (false);
        else
            return(linear_search_recursive(array, value, startIndex + 1, endIndex));
    }
}
```

- The red font is the recursive call which reduces the problem by reducing the array size by one each time it is called.

(c) Copyright 2009-2016, H. Hakimzadeh
Recursion

Recursive Functions:

- Example 3:
  - A Recursive Function to determine if a value is in an array.

```c
int sum_recursive (int MyArray[], int ArraySize)
{
    if(ArraySize == 1)
        return(MyArray[0]);
    else
        return(MyArray[0] + sum_recursive( &MyArray[1], ArraySize-1 ));
}
```

- The red font is the recursive call which reduces the problem by reducing the array size by one each time it is called.
Other examples:

- Power()
- BinarySearch()
- ReverseList()
- etc
Recursion

Recursive Functions:

- Example 2:
  - A Recursive Function to calculate the Fibonacci series.
  - \( F_n = F_{n-1} + F_{n-2} \)
  - The sequence begins: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

```c
int fibonacci(int n)
{
    if(n == 0) return 0;
    else if(n == 1) return 1;
    else return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- The red font is the recursive call which reduces the problem by reducing the size of the sequence by 1, every time it is called.
Infinite recursion (stack overflow)
Thinking about efficiency
Thinking about elegance (also less coding!)
Thinking outside of the box!
- Thinking Recursively vs. Iteratively

When not to use Recursion?
Recursion—How does it work?

- Computer tracks recursive calls
  - Recursive calls are just another function call.
  - We pause the current function
  - Must know results of new recursive call before proceeding
  - Saves all information needed for current call
    - To be used later
  - Proceeds with evaluation of new recursive call
  - When THAT call is complete, returns to "outer" computation
Recursion Big Picture

Outline of successful recursive function:

- One or more cases where function accomplishes it’s task by:
  - Making one or more recursive calls to solve smaller versions of original task
  - Called "recursive case(s)"
- One or more cases where function accomplishes it’s task without recursive calls
  - Called "base case(s)" or stopping case(s)
Infinite Recursion

- Base case MUST eventually be entered

- If it doesn’t → infinite recursion
  - Recursive calls never end!
Infinite Recursion Example

- Consider alternate factorial function definition:

```c
int factorial_recursive(int value)
{
    return(value * factorial_recursive(value -1));
}
```

- Seems "reasonable" enough
- Missing "base case"
- Recursion never stops
Stacks and Recursion!
Stacks for Recursion

- A stack
  - Specialized memory structure
  - Like stack of paper
    - Place new items on top of stack
    - When needed, remove it from top
  - Called "last-in/first-out" memory structure

- Recursion uses stacks
  - Each recursive call placed on stack
  - When one completes, last call is removed from stack
Stack Overflow

- Size of stack limited
  - Memory is finite

- Long chain of recursive calls continually adds to stack
  - All are added before base case causes removals

- If stack attempts to grow beyond limit:
  - Stack overflow error

- Infinite recursion always causes this
Recursion vs. Iteration
Recursion Versus Iteration

Recursion is not always "necessary"

Any task accomplished with recursion can also be done without it

- Non-recursive: called iterative, using loops

Recursive:
- Runs slower, uses more storage
- But typically is more elegant, and less code
Recursion.... How it work!

(c) Copyright 2007, H. Hakimzadeh
Recursion – How does it work!

```c
int power (int x, int n)
{
    if ( n > 0)
        return (power(x, n - 1) * x );
    else
        return (1);
}

main()
{
    power( 2, 3);
}
```
Calling Function power()

Larger example:

\[\text{power}(2,3)\]

\[\rightarrow \text{power}(2,2) \times 2\]

\[\rightarrow \text{power}(2,1) \times 2\]

\[\rightarrow \text{power}(2,0) \times 2\]

\[\rightarrow 1\]

- Reaches base case
- Recursion stops
- Values "returned back" up stack
Tracing the power() function:

Display 13.4  Evaluating the Recursive Function Call \emph{power}(2, 3)

Sequence of recursive calls

1

\begin{align*}
\text{power}(2, 0) \times 2 \\
\text{power}(2, 1) \times 2 \\
\text{power}(2, 2) \times 2 \\
\text{power}(2, 3)
\end{align*}

How the final value is computed

1

\begin{align*}
1 \times 2 \text{ is } 2 \\
2 \times 2 \text{ is } 4 \\
4 \times 2 \text{ is } 8 \\
\text{power}(2, 3) \text{ is } 8
\end{align*}
Thinking Recursively

- Ignore details
  - Forget how stack works
  - Forget the suspended computations
  - Yes, this is an "abstraction" principle!
  - And encapsulation principle!

- Let computer do the "bookkeeping"
  - Programmer just think "big picture"
Recursive Design Techniques

- Don’t trace entire recursive sequence!
- Just check 3 properties:
  1. No infinite recursion
  2. Stopping cases return correct values
  3. Recursive cases return correct values
Design Check for power()

- Check power() against 3 properties:
  1. No infinite recursion:
     - 2nd argument decreases by 1 each call
     - Eventually must get to base case of 1
  2. Stopping case returns correct value:
     - power(x, 0) is base case
     - Returns 1, which is correct for \( x^0 \)
  3. Recursive calls correct:
     - For \( n > 1 \), power(x, n) returns power(x, n-1) \( \times x \)
     - Plug in values → correct
Summary 1

- Reduce problem into smaller instances of same problem -> recursive solution
- Recursive algorithm has two cases:
  - Base/stopping case
  - Recursive case
- Ensure no infinite recursion
- Use criteria to determine recursion correct (check the three essential properties)