What is recursion?

- Sometimes, the best way to solve a problem is by solving a **smaller version** of the exact same problem first.

- Recursion is a technique that solves a problem by solving a **smaller problem** of the same type.
Recursive Function

- A function is called recursive if it calls itself
- In C, all functions can be used recursively
- Example:

```c
#include <stdio.h>

int main(void)
{
    printf("The universe is never ending\n");
    main();
    return 0;
}
```

- This will act like an infinite loop
Recursive Function: Example

- This code computes the sum of first $n$ positive integers.
- For $n = 4$

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

<table>
<thead>
<tr>
<th>Function Call</th>
<th>Value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{sum}(1)$</td>
<td>1</td>
</tr>
<tr>
<td>$\text{sum}(2)$</td>
<td>$2+\text{sum}(1)$ or $2+1$</td>
</tr>
<tr>
<td>$\text{sum}(3)$</td>
<td>$3+\text{sum}(2)$ or $3+2+1$</td>
</tr>
<tr>
<td>$\text{Sum}(4)$</td>
<td>$4+\text{sum}(3)$ or $4+3+2+1$</td>
</tr>
</tbody>
</table>
Recursive Function

- There is a base case (or cases) that is tested upon entry.
- And a general recursive case
  - in which one of the variables, is passed as an argument in such a way as to ultimately lead to the base case.

```c
int sum(int n) {
    if(n <= 1) {
        return n;
    } else {
        return (n+sum(n-1));
    }
}
```
Problems Defined Recursively

- There are many problems whose solution can be defined recursively

**Example: factorial n**

\[
\begin{align*}
n! & = \begin{cases} 
1 & \text{if } n = 0 \\
(n-1)! \cdot n & \text{if } n > 0
\end{cases} \\
n! & = \begin{cases} 
1 & \text{if } n = 0 \\
1 \cdot 2 \cdot 3 \cdots \cdot (n-1) \cdot n & \text{if } n > 0
\end{cases}
\end{align*}
\]

(recursive solution)

(closed form solution)
Coding the Factorial Function

- Recursive Implementation

    ```c
    int Factorial(int n)
    {
        if (n==0)       // base case
            return 1;
        else
            return n * Factorial(n-1);
    }
    ```

    For n > 12 this function will return incorrect value as the final result is too big to fit in an integer
Trace of Recursion: Factorial

\[
\begin{align*}
5! & \\
5 \times 4! & \\
4 \times 3! & \\
3 \times 2! & \\
2 \times 1! & \\
1 \times 0! & = 1
\end{align*}
\]

\[
\begin{align*}
5! & \\
5 \times 4! & \\
4 \times 3! & \\
3 \times 2! & \\
2 \times 1! & \\
1 \times 0! & = 1
\end{align*}
\]

\[
\begin{align*}
5 \times 24 & = 120 \text{ is returned} \\
4 \times 6 & = 24 \text{ is returned} \\
3 \times 2 & = 6 \text{ is returned} \\
2 \times 1 & = 2 \text{ is returned} \\
1 \times 1 & = 1 \text{ is returned}
\end{align*}
\]
Coding the Factorial Function (cont.)

- Iterative Implementation

```c
int Factorial(int n)
{
    int fact = 1;

    for(int count = 2; count <= n; count++)
        fact = fact * count;

    return fact;
}
```

- Both recursive and iterative version returns same value
Another Example: $n$ choose $k$ (combinations)

Given $n$ things, how many different sets of size $k$ can be chosen?

\[
\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}, \quad 1 < k < n \quad \text{(recursive solution)}
\]

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}, \quad 1 < k < n \quad \text{(closed-form solution)}
\]

with base cases:

\[
\binom{n}{1} = n \quad (k = 1), \quad \binom{n}{n} = 1 \quad (k = n)
\]
**n choose k implementation**

```c
int Combinations(int n, int k)
{
    if(k == 1)    // base case 1
        return n;
    else if (n == k)    // base case 2
        return 1;
    else
        return(Combinations(n-1, k) + Combinations(n-1, k-1));
}
```
Combinations(4, 3) = Combinations(3, 2) + Combinations(3, 3)

Combinations(3, 2) = Combinations(2, 1) + Combinations(2, 2)

Combinations(2, 1) = 2
Combinations(2, 2) = 1

4 = 2 + 1 + 1
Recursion vs Iteration

- Iteration can be used in place of recursion
  - An iterative algorithm uses a *looping construct*
  - A recursive algorithm uses a *branching structure*

- Recursive solutions are often less efficient, in terms of both *time* and *space*, than iterative solutions

- Recursion can simplify the solution of a problem, often resulting in *shorter*, more easily understood source code
How to write a recursive function?

- Determine the **size factor**
- Determine the **base case(s)**
  (the one for which you know the answer)
- Determine the **general case(s)**
  (the one where the problem is expressed as a smaller version of itself)
- Verify the algorithm
  (use the "Three-Question-Method")
Three Question Verification

1. The Base-Case Question
   - Is there a non-recursive way out of the function, and does the routine work correctly for this "base" case?

2. The Smaller-Caller Question
   - Does each recursive call to the function involve a smaller case of the original problem, leading inescapably to the base case?

3. The General-Case Question
   - Assuming that the recursive call(s) work correctly, does the whole function work correctly?
Recursion: Calculation of Fibonacci Sequence

■ Recursive solution

\[ f_0 = 0, \quad f_1 = 1, \quad f_{i+1} = f_i + f_{i-1}, \text{ for } i = 1, 2, \ldots \]

- Except for \( f_0 \) and \( f_1 \), every element in the sequence is the sum of the previous two elements

■ The sequence begins 0, 1, 1, 2, 3, 5, 8, ...

```c
int Fibonacci(int n)
{
    if(n <= 1)   // base case
        return n;
    else
        return(Fibonacci(n-1) + Fibonacci(n-2));
}
```
Recursion: Calculation of Fibonacci Sequence

Fib(5)

Fib(4)

Fib(3)

Fib(2)

Fib(1)

Fib(0)

Fib(1)

Fib(0)
### Number of Function Calls for Recursive Fibonacci

<table>
<thead>
<tr>
<th>Value of n</th>
<th>Value of Fibonacci(n)</th>
<th>#of function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>23</td>
<td>28657</td>
<td>92735</td>
</tr>
<tr>
<td>24</td>
<td>46368</td>
<td>150049</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>42</td>
<td>267914296</td>
<td>8669888873</td>
</tr>
<tr>
<td>43</td>
<td>4334944437</td>
<td>1402817465</td>
</tr>
</tbody>
</table>

A large number of function call is required to compute the nth fibonacci for even moderate values of n

Shebuti Rayana (CS, Stony Brook University)
Pitfalls of Recursion

- Missing base case – failure to provide an escape case.

- No guarantee of convergence – failure to include within a recursive function a recursive call to solve a subproblem that is not smaller.

- Excessive space requirements - a function calls itself recursively an excessive number of times before returning; the space required for the task may be prohibitive.

- Excessive recomputation – illustrated in the recursive Fibonacci method which ignores that several sub-Fibonacci values have already been computed.