1 Dictionary Analogy

2 Computing Factorial Using Iteration and Recursion

3 Computing Fibonacci Using Iteration and Recursion

4 Writing a Recursive Function
1. Dictionary Analogy

Our goal is to understand what the word “recursion” means, so we look up “recursion” in the dictionary ...

- Recursion is when the algorithm is defined in terms of itself
- No new syntax or semantics
- Understanding recursion simply involves applying what we have already learned with respect to functions, conditionals, iteration
2. Computing Factorial Using Iteration and Recursion

Recall from mathematics, the factorial of a number (n!) is:

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

So in other words:

\[
\begin{align*}
0! &= 1 \\
1! &= 1 \\
2! &= 1 \times 2 = 2 \\
3! &= 1 \times 2 \times 3 = 6 \\
4! &= 1 \times 2 \times 3 \times 4 = 24 \\
5! &= 1 \times 2 \times 3 \times 4 \times 5 = 120
\end{align*}
\]

We can write a function to calculate factorial using a for loop:

```c
int factorial( int n ) {
    int result = 1;
    for ( int i = 1; i <= n; i++ )
        result = result * i;
    return result;
}
```
• The loop implementation does not really resemble the original mathematical formulation

• The mathematical formulation is inherently recursive

• Can we implement factorial more directly using recursion?

\[
n! = \begin{cases} 
1 & \text{if } n = 0 \\
n \times (n - 1)! & \text{if } n > 0 
\end{cases}
\]
We can use the exact same “by-hand” execution approach we learned in the previous topic to understand recursion.

```cpp
int factorial( int n )
{
    // base case
    if ( n == 0 ) {
        return 1;
    }
    // recursive case
    if ( n > 0 ) {
        return n * factorial(n-1);
    }
}

int main()
{
    factorial(3);
    return 0;
}
```

Questions:

- What if n is negative?
- What if the execution arrow reaches end of a non-void function without encountering a return statement?
Recall from mathematics, the Fibonacci sequence is a sequence of integers such that every number after the first two is the sum of the two preceding ones:

\[ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots \]

The numbers in the Fibonacci sequence are called “Fibonacci numbers”. By definition, the first two numbers in the Fibonacci sequence are 0 and 1. Ancient scholars realized the importance of this sequence in both mathematics and nature. Fibonacci sequences can be found in the arrangement of leaves on a stem or pattern in a pine cone.

We can write a function to calculate the \( n \)th Fibonacci number using a for loop:

```c
int fib( int n ) {
    // by definition
    if (n == 0) return 0;
    if (n == 1) return 1;

    int fib_minus2 = 0;
    int fib_minus1 = 1;
    int result = 0;

    for ( int i=2; i<=n; i++ ) {
        result = fib_minus1 + fib_minus2;
        fib_minus2 = fib_minus1;
        fib_minus1 = result;
    }
    return result;
}
```
Can we implement factorial more elegantly using recursion?

Illustrating call tree for $\text{fib}$
4. Writing a Recursive Function

Write pseudo-code for a recursive function which draws the tick marks on a vertical ruler. The middle tick mark should be the longest and mark the 1/2 way point, slightly shorter tick marks should mark the 1/4 way points, even slightly shorter tick marks should mark the 1/8 way points and so on. The recursive function should take one argument: the height of the middle tick mark (i.e., the number of dashes). Use printf to display the tick marks.

```
void ruler( int height ) {

}
```
4. Writing a Recursive Function

- Step 1: Work an example yourself (height = 2, height = 3)
- Step 2: Write down what you just did
  - What is the base case?
  - What is the recursive case?
- Step 3: Generalize your steps (for any height)
- Step 4: Test your algorithm (does it work for height = 4?)
- Step 5: Translate to code

Think about the recursive call tree?

Manually work through example ruler