Recursion
Zero to Hero

Who are you?

By Avi Farah

https://www.linkedin.com/in/avi-farah/

avifarah@hotmail.com

Agenda and resources

- Slides
  - Recursion: What, Why, and How
  - Tail recursion vs. Left recursion
  - Recursion for processing a binary tree

- There are five GitHub repositories that are linked
- https://github.com/avifarah/Recursion
- https://github.com/avifarah/Recursion.Recursion1
- https://github.com/avifarah/Recursion.Recursion-Stripped
- https://github.com/avifarah/Recursion.TreeProcessing
- https://github.com/avifarah/Recursion.TreeProcessing-Stripped

What is Recursion

- A recursive solution is broken down into
  - **Termination condition**
  - **Recursive logic**—Same logic applied to `n` items is applied to `n – 1` items

Example: `n! = 1 * 2 .. * n`

Recursive definition: `n! = n * (n – 1)!`

Terminating Condition: when `n == 1` then `n! == 1`

C# code:

```
Factorial(int n) {
        return n == 1 ? 1 : n * Factorial(n-1);
    }
```

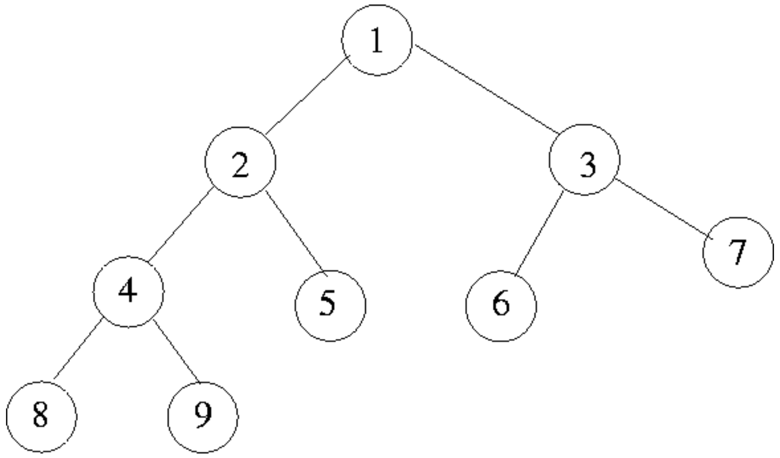
- The above `Factorial(..)` will **NOT** work if `n <= 0`

Why use Recursion

- Every iteration process can be expressed as a recursion

- Not every recursion process can be expressed as iteration

➢ Example: There are structures like trees where iteration is not ideal



The How of Recursion

- To implement recursion, compilers use a stack—the same stack that is used to control the flow of function/method calls.

➢ At the recursive call: local variables and return address are pushed

➢ Upon return: top frame is popped and hydrate local variables

- A recursive solution is therefore more expensive than its iterative counterpart. ***Tail recursion is an exception…***

Tail Recursion -- ***Desirable***

- Factorial of n

```
public static BigInteger Factorial(int n) {
   if (n <= 0) throw new ArgumentException("...", nameof(n));
   return FactorialHelper(n);
}

private static BigInteger FactorialHelper(int n) {
  if (n == 1)  return 1;
  /*option 1*/ return n * FactorialHelper(n-1);
  /*option 2*/ return FactorialHelper(n-1) * n;
}
```

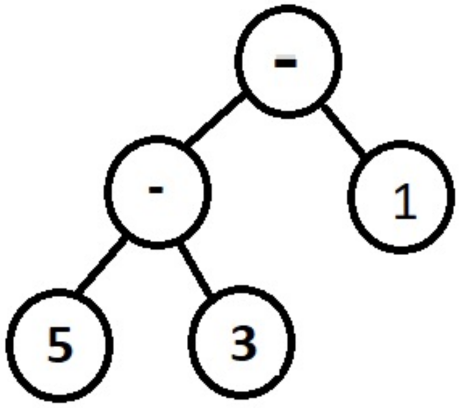
- Option 1 is called “Tail Recursion”
- Option 2 is called “Left Recursion”
- The compiler will turn a Tail Recursion to an iteration.

Left Recursion (example)

- We need to translate an expression from its token parts. Ex: 5-3-1

```
Definition (A parser for arithmetic addition / subtraction)
expr ::=   expr (+|-) term
         | term

term ::= number (a string of digits)
```

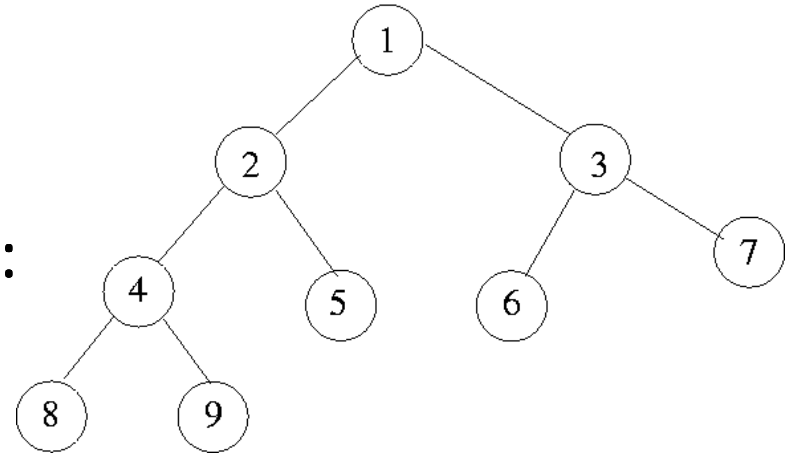


- Translating this to code – let's think about it together
- **Each step through the recursive algorithm must move the process a step closer to the terminating condition**
- Fix expr definition:

```
expr ::= term Rest
Rest ::= (+|-) expr
       | empty
```

Binary Tree

- Binary tree has
  - Node
  - Left subtree (a tree in its own right)
  - Right subtree (a tree in its own right)
- A binary Tree can be recursively processed in:
  - n L R (prefix processing)
  - L n R (infix processing)
  - L R n (postfix processing)
- If the needed operation does not fall into pre/in/post-fix processing, then the tree structure given above is not the best data structure. For example: `For a given depth list all node values`.



Induction

- Inductive reasoning is where we observe a number of special cases (at least 1)
- Then we show that if our observation is true for the nth case, our observed pattern holds for the (n + 1)st case
- Example:

$$\begin{aligned} 1 \qquad\qquad\;\; &= 1 = 1^2 \\ 1 + 3 \qquad\; &= 4 = 2^2 \\ 1 + 3 + 5 &= 9 = 3^2 \end{aligned}$$

$\text{Assume that sum of the first N odd integers} = N^2 \colon \sum_{n=1}^{N}(2n-1) = N^2$

Sum of first N+1 odd numbers:

$$\sum_{n=1}^{N+1}(2n-1) = \left(\sum_{n=1}^{N}(2n-1)\right) + (2(N+1)-1) = N^2 + 2N + 1 = (N+1)^2$$

Induction to Recursion, reverse the process

- We take the Problem:

$$\sum_{n=1}^{N} (2n-1) = Problem(N)$$

$$\sum_{n=1}^{N} Odd(n) = Problem(N)$$

- Break it into

$$\sum_{n=1}^{N} Odd(n) = Odd(N) + \left(\sum_{n=1}^{N-1} Odd(n)\right)$$

$$= Odd(N) + Problem(N-1)$$

- To make it work, we need a terminating condition