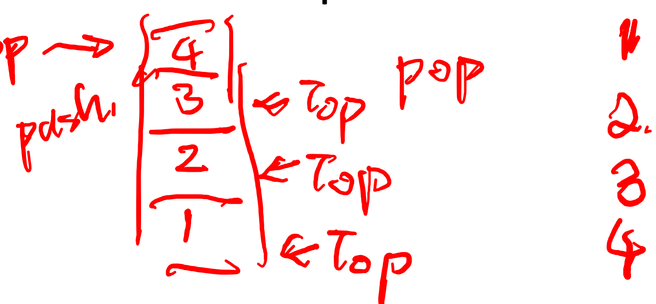
Applications of Stacks

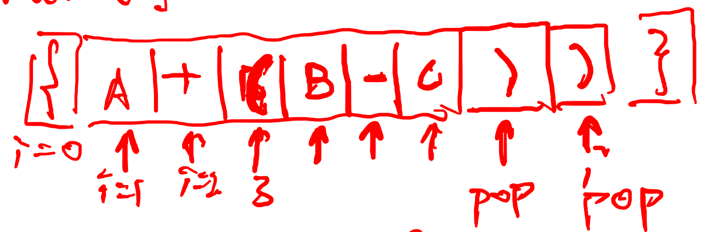
- Reversing a list
  - A list of numbers can be reversed by pushing each number from the first position to the last position onto a stack and then popping each number off the stack starting with the first position in the reversed list.
  - List: 1, 2, 3, 4
  - Stack: 4->3->2->1
  - RList: 4, 3, 2, 1

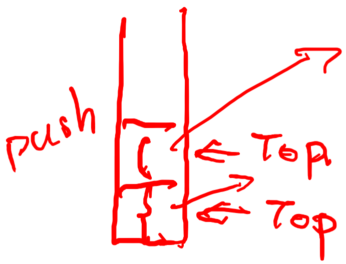


- Parentheses checker
  - Stacks can be used to push open parentheses or braces and pop them as closing parentheses/braces are encountered.
  - If mismatches occur or any leftover parentheses in the stack or expression, then the parentheses or braces would be incorrect.
  - Expression: (A+B}, Stack: (, Error when popping (on }, invalid
  - Expression: {A+(B-C)}, Stack: {(, pop (matches), pop { matches }, valid

$\{A+(B-C)\}$

valid ?





( ) ✓

{) × → print invalid

if they matches

Recursion & Stack ADT

- **Recursion** is an implicit application of the **STACK ADT** 📚.
- A **recursive function** calls itself 🔁 to solve a smaller version of its task until a **base condition** is met ✅.

A( )

A( ) ~~B( )~~

🧩 **Key Components of Recursion:**

1. 🛑 **Base Case:** No function call.

1. The simplest scenario where the problem can be solved **directly** without further recursive calls.

2. 🔄 **Recursive Case:**

1. The problem is **broken down** into smaller subproblems ✂️.
2. The function **calls itself** 🔁 with these subproblems. function call
3. The **final result** is obtained by **combining** solutions of the subproblems.

⚠️ **Recursion is powerful**, but must be carefully designed to avoid **infinite loops** 🔂 and **excessive memory usage** 💾.

problem : Calculate Factorial of a number

✓ Fact(4) = 4 × 3 × 2 × 1

Method 1: for loop

Method 2: recursion

Fact(1) = 1

Fact(2) = 2 × 1 = 2 × Fact(1)

Fact(3) = 3 × 2 × 1 = 3 × Fact(2)

Fact(4) = 4 × 3 × 2 × 1 = 4 × Fact(3)

⋮

Fact(n) = n × Fact(n-1)

↓

recursive Function

pattern

✓ ① Divide the problem into small subproblems

✓ ② Find Base condition

```
Fact (int n) {
  if ( n == 1 )        → Base
  {
    return 1;
  }
  else
  {
    return n * Fact(n-1);     recursive
  }
}
```

runs interatively

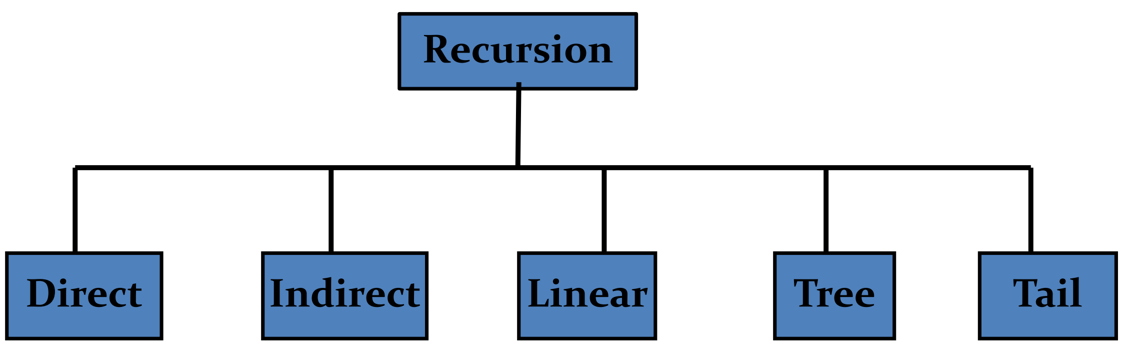
Divide and Conquer

↓

combine

Types of Recursion

- Any recursive function can be characterized based on:
  - whether the function calls itself directly or indirectly (direct or indirect recursion).
  - the structure of the calling pattern (linear or tree-recursive).
  - whether any operation is pending at each recursive call (tail-recursive or not).



Direct Recursion

- A function is said to be *directly* recursive if it explicitly calls itself.
- For example, consider the function given below.

```
int Func( int n)
{
    if(n==0)
        return n;
    return (Func(n-1));
}
```

Indirect Recursion

- A function is said to be *indirectly* recursive if it contains a call to another function which ultimately calls it.
- Look at the functions given below. These two functions are indirectly recursive as they both call each other.

``int Func1(int n)``` ```{``` ``` if(n==0)``` ``` return n;``` ``` return Func2(n);``` ```}```	```int Func2(int x)``` ```{``` ``` return Func1(x-1);``` ```}``` cycle

Tail Recursion

- A recursive function is said to be *tail recursive* if no operations are pending when the recursive function returns to its caller.

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

not Tail

= Fact1(int n, n·Fact(n-1))

acc

return (Fact(n-1)); Tail.

Fact is not tail-recursive since "n*" remains to be done after Fact(n-1) has returned.

Convert Non Tail → Tail.

Fact is now tail-recursive since the result is computed using "n*acc" which matches to formal parameter "acc".

```
int Fact(n)
{
    if (n==0)
        return 1;
    return Fact1(n, 1);
}
```

```
int Fact1(int n, int acc)
{
    if (n==0)
        return acc;
    return Fact1(n-1, n*acc);
}
```

acc = n·Fact(n-1)

recursive

Tail Recursion

few ~~more~~ unfinished function into stack

① save memory.

unfinished

non-tail → needs more memory.

② few function calls

③ ~~int~~ iterative procedure /code using for while

- Tail recursive functions are highly desirable.
- Modern compilers do tail call elimination to optimize the tail recursive code to avoid making recursive calls.

```
int Fact1(int n, int acc)
{
 start:
       if (n==1)
          return acc;
       else
          acc = n*acc;
          n = n - 1;
       goto start;
}
```

acc = n. acc.

} base case.

loop.

no function call.

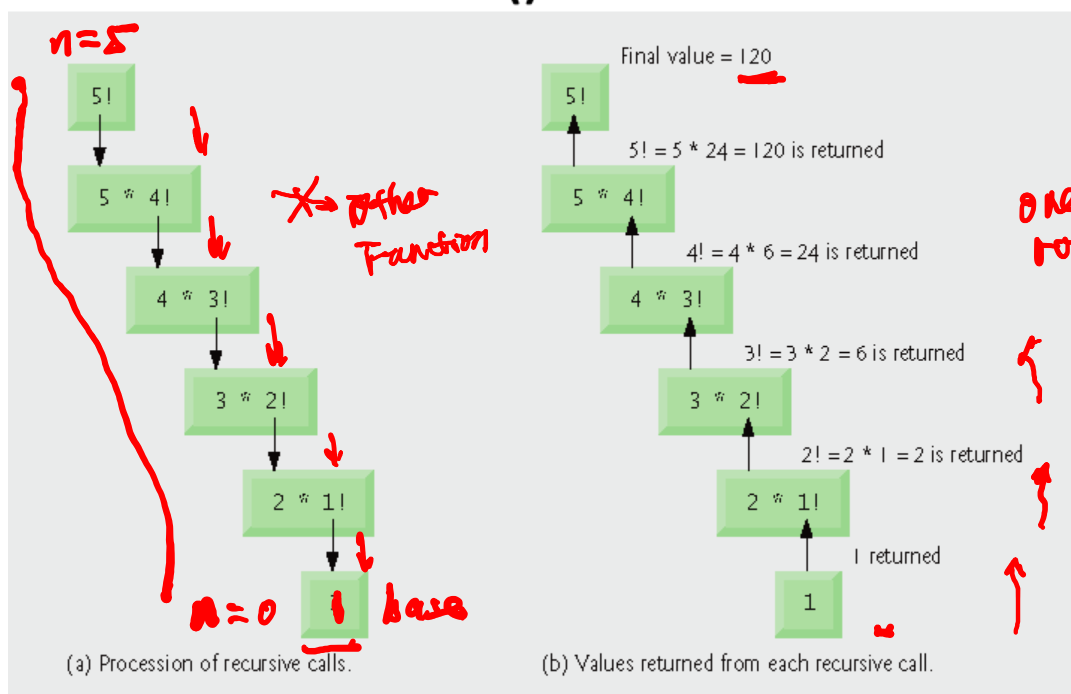
Linear Recursion

- A recursive function is said to be *linearly* recursive when no pending operation involves another recursive call to the function.
- For example, the factorial function is linearly recursive as the pending operation involves only multiplication to be performed and does not involve another call to the Fact() function.

$$n! = n \cdot (n-1)!$$

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

~~1 Fact~~(n-1)



(a) Procession of recursive calls.

(b) Values returned from each recursive call.

Tree Recursion

- A recursive function is said to be *tree* recursive (or *non-linearly* recursive) if the pending operation makes another recursive call to the function.
- For example, the Fibonacci function begins with 0 and 1. Each subsequent Fibonacci number is the sum of the previous two Fibonacci numbers. It can be defined recursively as follows:

Fibonacci(0) = $\overset{=n}{0}$, Fibonacci(1) = $\overset{=n}{1}$

Fibonacci(n) = Fibonacci(n – 1) + Fibonacci(n – 2) for n > 1

```
int Fibonacci(int n)
{
        if(n <= 1)
                return n;
        return ( Fibonacci (n - 1) + Fibonacci(n – 2));
}
```

=n=0 & n=1

base

expand

Fibonacci Series

- The Fibonacci series can be given as:

0 1 1 2 3 5 8 13 21 34 55

……

Fibonacci(n) = Fibonacci(n – 1) + Fibonacci(n – 2)

- Each term in the series is the sum of the two previous terms except for the first and second terms of 0 and 1. → base case
- The recursive call FIB(7) has the following recursive call tree.

n=7

Note the repetitive calls.