递归例子-PURE FUNCTION

对于递归函数，是否总存在一种纯函数？ 或者说，只有纯函数的递归是我们最希望的?

> 纯函数是指在给定相同的输入时，始终返回相同的输出，并且没有副作用的函数。换句话说，纯函数的执行不依赖于系统状态的改变，也不会改变系统状态。
>
> 具体来说，纯函数具有以下特性：
>
> 1. **确定性**：对于相同的输入，纯函数总是返回相同的输出。这意味着函数的行为是可预测的，不受外部环境的影响。
> 2. **无副作用**：纯函数不会引起任何可观察到的副作用，如修改全局变量、修改函数参数、写入文件或向网络发送请求等。它只是接受输入并生成输出。
> 3. **可交换性**：纯函数可以随时被替换为其结果，而不会影响程序的行为。这使得并行化和测试更加容易。
> 4. **可缓存性**：对于相同的输入，纯函数的结果可以被缓存，以提高性能。
>
> 纯函数在函数式编程中起着重要作用，因为它们使得代码更加可靠、可测试和易于理解。通过避免副作用，纯函数也有助于减少代码中的错误，并提高代码的可维护性。

阶乘函数

$$f(n) = n! = \prod_{i=1}^{n} i = 1 \times 2 \times 3 \times \ldots \times (n-1) \times n$$

```python
def fact(n: int) -> int:
    if n == 0:
        return 1
    else:
        return n * fact(n - 1)
```

低效不应该被指责！（优化递归--缓存结果）--实际上是对纯函数的优化。

```python
def fact_helper(n: int, accumulator: int = 1) -> int:
    if n == 0:
        return accumulator
    else:
        return fact_helper(n - 1, n * accumulator)

def fact(n: int) -> int:
    if n < 0:
        raise ValueError("阶乘未定义于负数")
    return fact_helper(n)
```

> 因为是纯函数，所以说无额外的副作用。 所以说可以进行优化。

Fibonacci函数

$$F(n) = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F(n-1) + F(n-2) & \text{if } n > 1 \end{cases}$$

```
def fibonacci(n: int) -> int:
    if n < 2:
        return n;
    else:
        return fibonacci(n - 1) + fibonacci(n - 2);
```

低效不应该被指责！（优化递归--缓存结果）--实际上是对纯函数的优化。

```
def additiveSequence(n: int, t0: int, t1: int) -> int :
    if n == 0: return t0
    if n == 1: return t1
    return additiveSequence(n - 1, t1, t0 + t1)

assert(fibonacci(12) == additiveSequence(12, 0, 1))
```

超越递归？ 动态规划

```
def fibonacci(n: int) -> int:
    # 使用列表推导式创建一个长度为 n+1 的列表，每个元素都初始化为 0
    fib = [0 for _ in range(n + 1)]

    # 基础情况：前两个斐波那契数为 0 和 1
    fib[0], fib[1] = 0, 1

    # 使用动态规划计算斐波那契数值
    for i in range(2, n + 1):
        fib[i] = fib[i - 1] + fib[i - 2]

    return fib[n]
```

binarySearch的例子

```
def findInList(key: str, lst: list):
    def binarySearch(key: str, lst: list, p1: int, p2: int):
        if p1 > p2:
            raise ValueError("No match is found")
        mid = (p1 + p2) / 2
        if key == lst[mid]:
            return mid;
        if key < lst[mid]:
            return binarySearch(key, lst, p1, mid - 1)
        else:
            return binarySearch(key, lst, mid + 1, p2)

    return binarySearch(key, lst, 0, len(lst) - 1)
```

间接递归的例子？　尽管低效

```
def isOdd(n: int):
    return !isEven(n)

def isEven(n: int):
    if n == 0:
        return True
    else:
        return isOdd(n - 1)
```

递归的思想

- **定义基本情况（Base Case）；**
- **将问题划分为规模较小的子问题；**

递归是一种解决问题的方法，其中问题被分解为较小的、相似的子问题，并通过解决这些子问题来解决原始问题。递归的思想可以用以下步骤来描述：

1. **定义基本情况（Base Case）**：确定问题的最小规模，即不再需要递归解决的情况。这些基本情况通常可以直接计算出结果，而不需要进一步的递归。
2. **将问题划分为规模较小的子问题**：将原始问题分解为一个或多个与原始问题相似但规模较小的子问题。这些子问题通常可以通过递归调用同一函数来解决。
3. **递归调用**：对每个子问题进行递归调用，直到达到基本情况为止。递归调用的目的是解决子问题并获取子问题的结果。
4. **合并子问题的结果**：一旦子问题得到解决，将它们的结果合并以获得原始问题的解决方案。这通常涉及将子问题的解组合成原始问题的解。
5. **返回解决方案**：最后，返回合并结果作为原始问题的解。

递归是一种强大的编程技术，它使得解决复杂问题变得更加简单和直观。然而，需要注意的是，如果不小心使用递归，可能会导致性能问题或栈溢出错误。因此，在设计递归算法时，需要确保递归的退出条件（基本情况）是有效的，并且递归调用会最终收敛于基本情况。

递归的思考