第十二章 递归算法

1. 递归

在函数内部，可以调用其他函数。如果一个函数在内部调用自身本身，这个函数就是递归函数。
举个例子，我们来计算阶乘 $n! = 1 \times 2 \times 3... \times n$,用函数fac(n) 表示，可以看出：

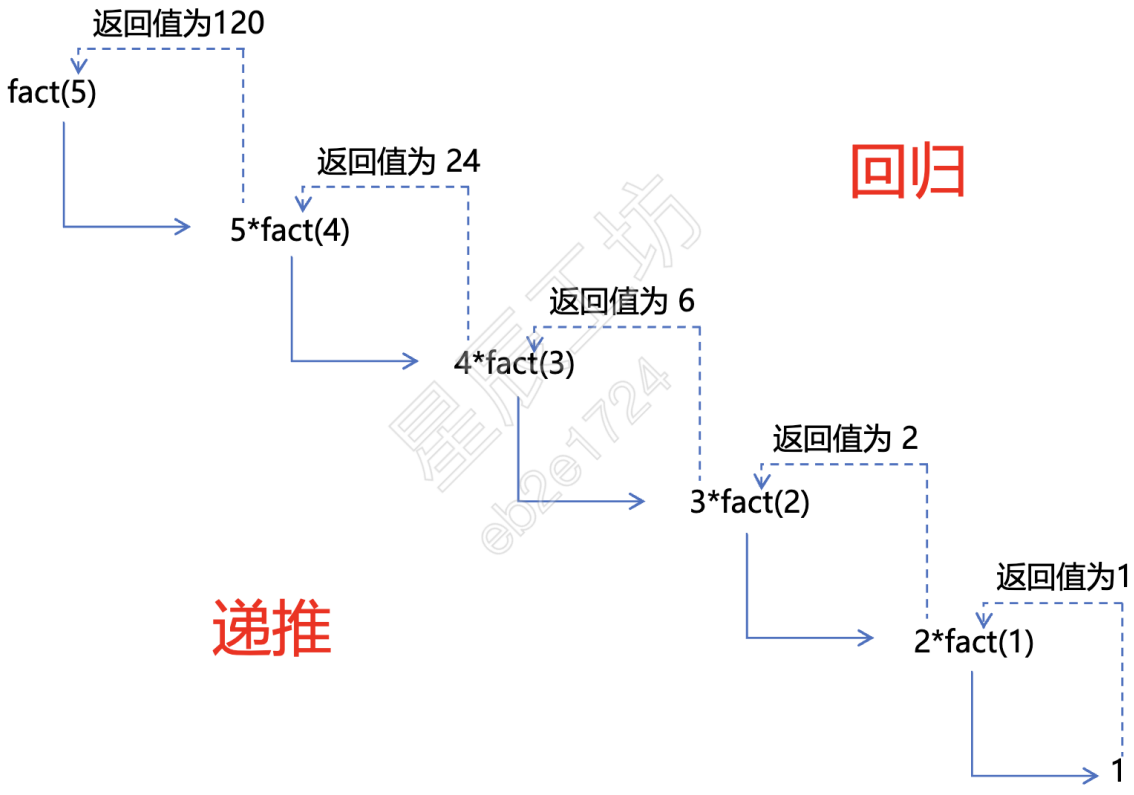
$$fac(n) =! = 1 \times 2 \times 3... \times n = (n-1)! \times n = fac(n-1) \times n$$

所以，fac(n)可以表示为 $n \times fac(n-1)$，只有n==1时需要做特殊处理，因为1不可以再继续运行下去了，fac(0) 无意义的，并且递归函数一般都会有截止的case 于是该程序用递归方式写就是

```
int fact(n):
{
    if (n==1):
        return 1;
    return n * fact(n - 1);
}
```

如果我们计算fac(5),可以根据函数定义看到计算过程如下：
===> fact(5)
===> 5 * fact(4)
===> 5 * (4 * fact(3))
===> 5 * (4 * (3 * fact(2)))
===> 5 * (4 * (3 * (2 * fact(1))))
===> 5 * (4 * (3 * (2 * 1)))
===> 5 * (4 * (3 * 2))
===> 5 * (4 * 6)
===> 5 * 24
===> 120

- 示意图：



- 过程分析：
  n 的值为 5， fact() 函数的执行过程：

  调用 fact(5) 时，由于形参 n 的值为 5，不等于 1，所以执行 5 * fact(4) 并返回它的值。为了求出这个表达式的值，必须先执行 fact(4) 并得到它的返回值，所以编译器转而求 fact(4) 的值，5 * fact(4) 的求值被搁置，等待后续再计算；

  调用 fact(4) 时，由于形参 n 的值为 4，不等于 1，所以执行 4 * fact(3) 并返回它的值。为了求出这个表达式的值，必须先执行 fact(3) 并得到它的返回值，所以编译器转而求 fact(3) 的值，4 * fact(3) 的求值被搁置，等待后续再计算；

  调用 fact(3) 时，由于形参 n 的值为 3，不等于 1，所以执行 3 * fact(2) 并返回它的值。为了求出这个表达式的值，必须先执行 fact(2) 并得到它的返回值，所以编译器转而求 fact(2) 的值，3*fact(2) 的求值被搁置，等待后续再计算；

  调用 fact(2) 时，由于形参 n 的值为 2，不等于 1，所以执行 2 * fact(1) 并返回它的值。为了求出这个表达式的值，必须先执行 fact(1) 并得到它的返回值，所以编译器转而求 fact(1) 的值，2 * fact(1) 的求值被搁置，等待后续再计算；

  调用 fact(1) 时，由于形参 n 的值为 1，函数的返回值为 1；

  知道了 fact(1) 的返回值为 1，先前被搁置的 2*fact(1) 的值就可以计算出来，因此 fact(2) 的返回值为 2；

  知道了 fact(2) 的返回值为 2，先前被搁置的 3*fact(2) 的值就可以计算出来，因此 fact(3) 的返回值为 6；

  知道了 fact(3) 的返回值为 6，先前被搁置的 4*fact(3) 的值就可以计算出来，因此 fact(4) 的返回值为 24；

  知道了 fact(4) 的返回值为 24，先前被搁置的 5*fact(4) 的值就可以计算出来，因此 fact(5) 的返回值为 120。

  递归函数的优点是定义简单，逻辑清晰。理论上，所有的递归函数都可以写成循环的方式，但循环的逻辑不如递归清晰。
  使用递归函数需要注意防止栈溢出。在计算机中，函数调用是通过栈（stack）这种数据结构实现的，每当进入一个函数调用，栈就会加一层栈帧，每当函数返回，栈就会减一层栈帧。由于栈的大小不是无限的，所以，递归调用的次数过多，会导致栈溢出。

2.递归算法

递归算法是一种通过函数直接或者间接调用自身来解决问题的方法。将一个大问题层层转化为一个或多个与原问题相似的规模较小的子问题，直到某个子问题可以直接求解，然后一层层返回得到最终结果 。

- 使用递归算法的必要条件：

1. 子问题具有相同的结构：递归算法的关键是将问题分解成一个或多个子问题，并且子问题与原问题具有相同的结构。 这样， 我们可以通过递归地解决子问题来解决原问题。
2. 子问题规模更小：递归算法要求子问题的规模比原问题的规模更小，这样才能通过递归不断地将问题分解，直到某个子问题可以直接求解
3. 存在递归终正条件：递归算法需要定义递归的终正条件，当满足终正条件时，递归终止，直接返回结果。
4. 存在合并子问题的操作：递归算法在递归求解子问题之后， 需要将子问题的解合并起来，得到原问题的解。

- 递归算法的分类：

1. 直接递归：即在函数中调用函数本身
2. 间接递归：即间接地调用一个函数，如funa调用fun b， funb又调用fun a。

- 示例：
  仍然是上面的阶乘程序

3. 递归算法总结

1. 递归算法将复杂的问题分解为更小、更简单的子问题。 通过递归调用自身，在每一层中处理当前子问题，并继续递归处理子问题的子问题，最终得到问题的解决方案

2. 递归算法的代码通常比选代算法更简洁、 易于理解。递归通过函数自身的调用来实现循环结构代码结构更接近问题的本质，从而更容易表达问题的解决思路
3. 递归算法可能导致函数调用的套层级过深，从而消耗较多的堆栈空间。 在处理大规模问题时,递归深度可能会导致栈溢出的问题，需要注意设置递归终止条件或改用选代算法
4. 递归算法在处理问题时可能会重复计算相同的子问题，导致性能下降。 可以通过使用记忆化的方式 （如缓存中间结果） 来避免重复计算，提高算法效率
5. 递归算法法必须设置递归终止条件，以防正无限递归调用 总之，递归算法是一种强大且灵活的问题解决方法，适用于处理递归结构、 深度优先搜索等问题.