函数执行流程

C语言中，函数的活动和栈有关。

栈是后进先出的数据结构。栈是由底端向顶端生长，栈顶加入数据称为压栈、入栈，栈顶弹出数据称为出栈。

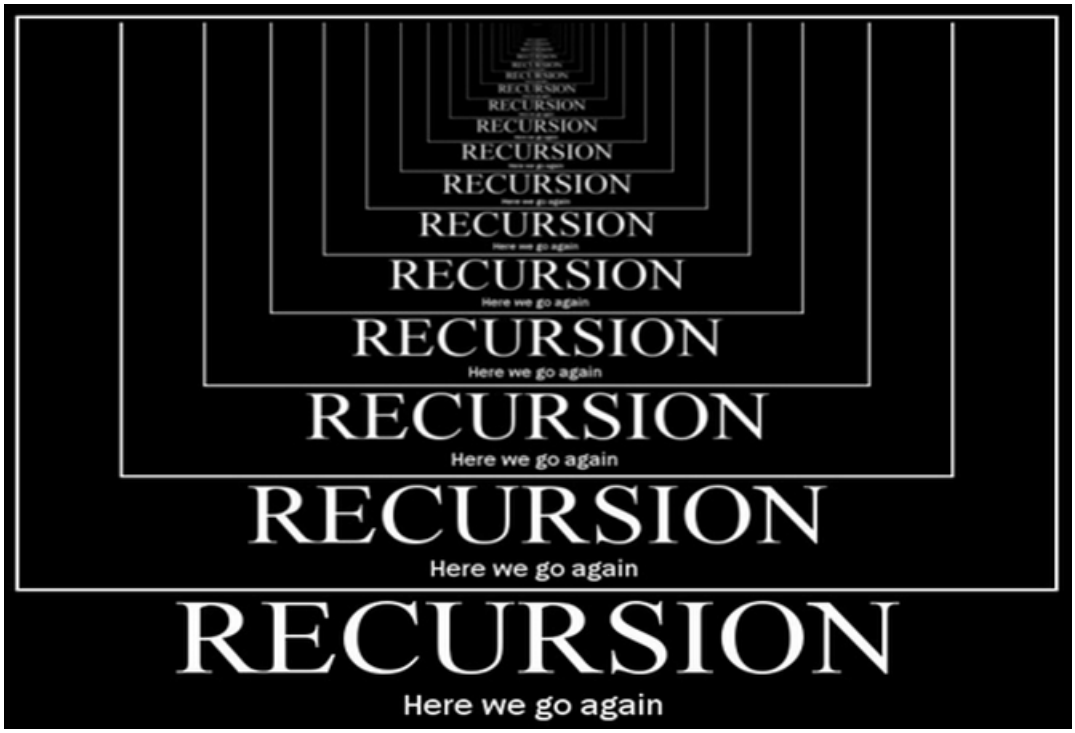
```
def add(x, y):
    r = x + y
    print(r)
    return r

def main():
    a = 1
    b = add(a, 2)
    return b


main()
```

- main调用，在栈顶创建栈帧
- a = 1，在main栈帧中增加a，堆里增加1，a指向这个1
- b = add(a, 2)，等式右边先执行，add函数调用
- add调用，在栈顶创建栈帧，压在main栈帧上面
- add栈帧中增加2个变量，x变量指向1，y指向堆中新的对象2
- 在堆中保存计算结果3，并在add栈帧中增加r指向3
- print函数创建栈帧，实参r被压入print的栈帧中
- print函数执行完毕，函数返回，移除栈帧
- add函数返回，移除栈帧
- main栈帧中增加b指向add函数的返回值对象
- main函数返回，移除栈帧

问题：如果再次调用main函数，和刚才的main函数调用，有什么关系？

每一次函数调用都会创建一个独立的栈帧入栈。
因此，可以得到这样一句**不准确**的话：哪怕是同一个函数两次调用，每一次调用都是独立的，这两次调用没什么关系。

递归



- 函数直接或者间接调用自身就是 递归
- 递归需要有边界条件、递归前进段、递归返回段
- 递归一定要有边界条件
- 当边界条件不满足的时候，递归前进
- 当边界条件满足的时候，递归返回

斐波那契数列递归

斐波那契数列Fibonacci number：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

如果设F(n）为该数列的第n项（n∈N*），那么这句话可以写成如下形式：:F(n)=F(n-1)+F(n-2)

有F(0)=0，F(1)=1, F(n)=F(n-1)+F(n-2)

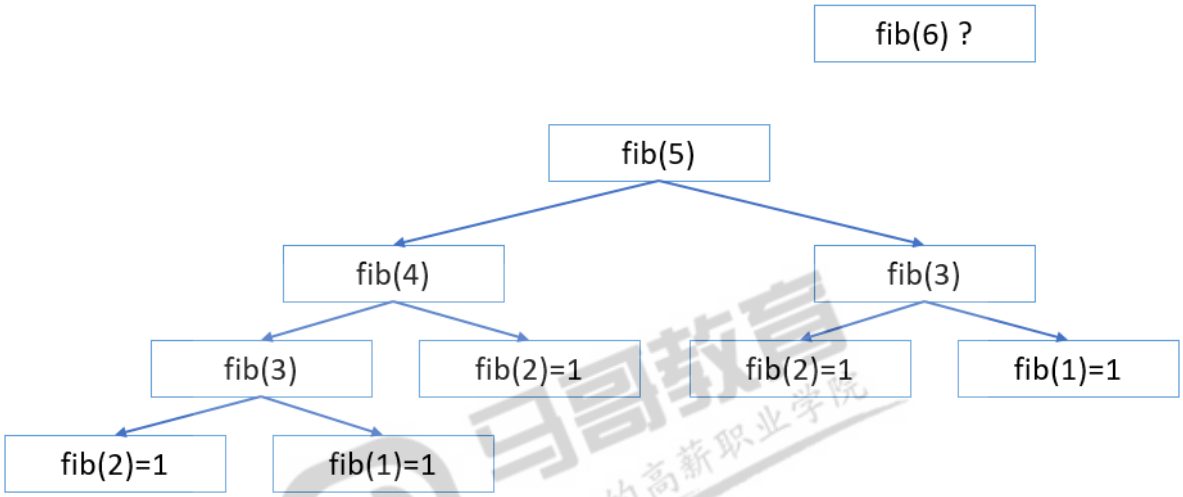
```
# 循环实现
def fib_v1(n): # n>=3
    a = b = 1
    for i in range(n-2):
        a, b = b, a + b
    return b

fib_v1(101)
fib_v1(35)
```

使用递归实现，需要使用上面的递推公式

```python
# 递归
def fib_v2(n):
    if n < 3:
        return 1
    return fib_v2(n-1) + fib_v2(n-2)

# 递归
def fib_v2(n):
    return 1 if n < 3 else fib_v2(n-1) + fib_v2(n-2)

fib_v2(35) # 9227465
```

递归实现很美，但是执行fib(35)就已经非常慢了，为什么?

以fib(5)为例。看了下图后，fib(6)是怎样计算的呢?



这个函数进行了大量的重复计算，所以慢。

递归要求

- 递归一定要有退出条件，递归调用一定要执行到这个退出条件。没有退出条件的递归调用，就是无限调用
- 递归调用的深度不宜过深
- Python对递归调用的深度做了限制，以保护解释器
  - 超过递归深度限制，抛出RecursionError： maxinum recursion depth exceeded 超出最大深度
  - sys.getrecursionlimit()

递归效率

使用时间差或者%%timeit来测试一下这两个版本斐波那契数列的效率。很明显循环版效率高。

难道递归函数实现，就意味着效率低吗?

能否改进一下fib_v2函数?

```python
# 递归
def fib_v3(n, a=1, b=1):
    if n < 3:
        return b
    a, b = b, a + b
    #print(n, a, b)
    return fib_v3(n-1, a, b) # 函数调用次数就成了循环次数，将上次的计算结果代入下次函数调用

fib_v3(101) # fib_v3(35)
# 提示：用fib_v3(3)代入思考递归后计算了几次
```

思考时，也比较简单，思考fib_v3(3)来编写递归版本代码。

经过比较，发现fib_v3性能不错，和fib_v1循环版接近。但是递归函数有深度限制，函数调用开销较大。

间接递归

```python
def foo1():
    foo2()

def foo2():
    foo1()

foo1()
```

间接递归调用，是函数通过别的函数调用了自己，这一样是递归。

只要是递归调用，不管是直接还是间接，都要注意边界返回问题。但是间接递归调用有时候是非常不明显，代码调用复杂时，很难发现出现了递归调用，这是非常危险的。

所有，使用良好的代码规范来避免这种递归的发生。

总结

- 递归是一种很自然的表达，符合逻辑思维
- 递归相对运行效率低，每一次调用函数都要开辟栈帧
- 递归有深度限制，如果递归层次太深，函数连续压栈，栈内存很快就溢出了
- 如果是有限次数的递归，可以使用递归调用，或者使用循环代替，循环代码稍微复杂一些，但是只要不是死循环，可以多次迭代直至算出结果
- 绝大多数递归，都可以使用循环实现
- 即使递归代码很简洁，但是能不用则不用递归

递归作业

递归是面试常见题目。请使用递归实现下面的作业题

- 求n的阶乘
- 解决猴子吃桃问题

- 猴子第一天摘下若干个桃子，当即吃了一半，还不过瘾，又多吃了一个。第二天早上又将剩下的桃子吃掉一半，又多吃了一个。以后每天早上都吃了前一天剩下的一半零一个。到第10天早上想吃时，只剩下一个桃子了。求第一天共摘多少个桃子