Lab03 & HW03 补充

pingpong

解法一

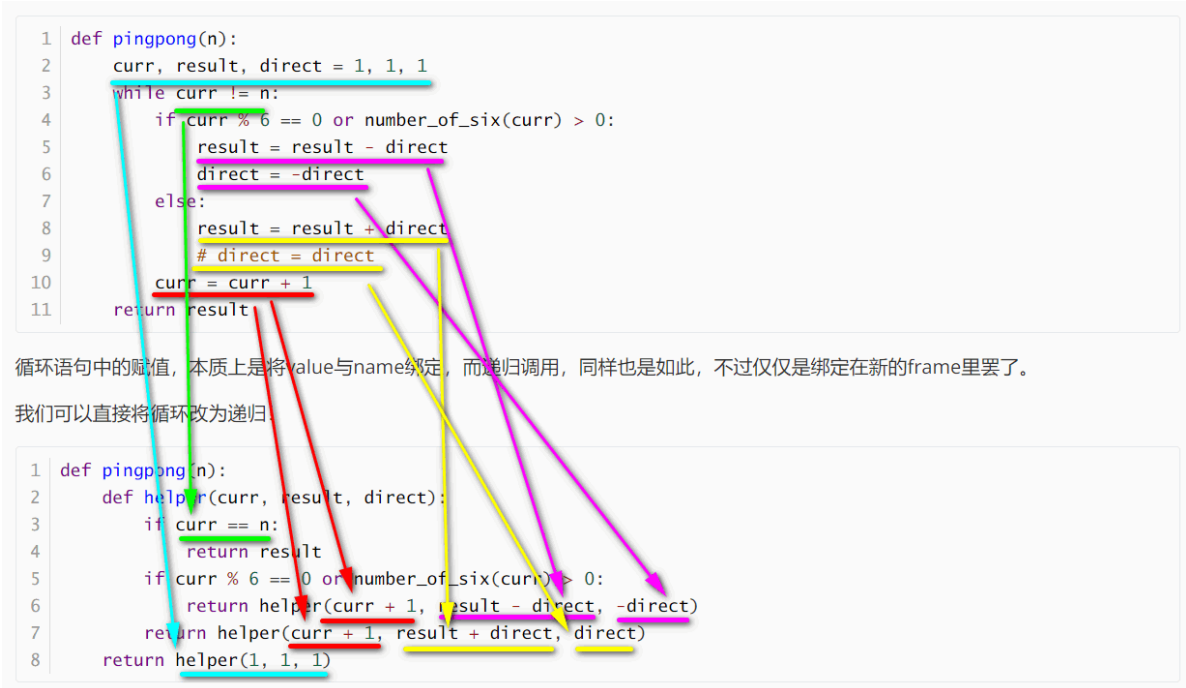
题目提示了先写循环，那就先写一个吧。

```
def pingpong(n):
    curr, result, direct = 1, 1, 1
    while curr != n:
        if curr % 6 == 0 or number_of_six(curr) > 0:
            result = result - direct
            direct = -direct
        else:
            result = result + direct
            # direct = direct
        curr = curr + 1
    return result
```

循环语句中的赋值，本质上是将value与name绑定，而递归调用，同样也是如此，不过仅仅是绑定在新的frame里罢了。

我们可以直接将循环改为递归！

```
def pingpong(n):
    def helper(curr, result, direct):
        if curr == n:
            return result
        if curr % 6 == 0 or number_of_six(curr) > 0:
            return helper(curr + 1, result - direct, -direct)
        return helper(curr + 1, result + direct, direct)
    return helper(1, 1, 1)
```



如此，我们的循环版本与递归版本的pingpong就是完全一致的。

至于helper函数的意义？与循环一样。

`result`是`n = curr`时的pingpong值，`direct`是`n = curr`时的+1/-1方向，返回值就是循环跳出时的`result`值。

解法二

首先确定`n`时的方向，再递归向下求解。

```
def pingpong(n):
    def ping_pong(k, f):
        if k <= 6:
            return k
        elif k % 6 == 0 or number_of_six(k) > 0:
            return ping_pong(k - 1, -f) - f
        else:
            return ping_pong(k - 1, f) + f
    def add_or_sub(k):
        if k < 6:
            return 1
        elif k % 6 == 0 or number_of_six(k) > 0:
            return -1 * add_or_sub(k - 1)
        else:
            return add_or_sub(k - 1)
    return ping_pong(n, add_or_sub(n))
```

解法三

不妨假设`n`时方向是加1（减1也行，无所谓），递归到1，如果1的时候方向是加1，就说明结果正确；如果是减1，根据对称性来调整结果。

什么对称性？如果1的时候的pingpong值是0的话，那么对于从1开始加1的pingpong和从1开始减1的pingpong，其值正好为相反数。

```
def pingpong(n):
    def ping_pong(k, f):
        if k == 1:
            return 0, f # 0: result, f: is_result_right (1: right, -1: wrong)
        elif k % 6 == 0 or number_of_six(k) > 0:
            return adjust(ping_pong(k - 1, -f), -f)
        else:
            return adjust(ping_pong(k - 1, f), f)
    def adjust(r, f): # adjust only result
        return r[0] + f, r[1]
    def final(r):
        return r[0] * r[1]
    return final(ping_pong(n, 1)) + 1
```

为什么我的代码会超时?

简单画一下递归调用栈就明白了。（为方便忽略`number_of_six`产生的栈）

150分TLE的代码：

```
pingpong(n) -> pingpong(n-1) ->  ...  -> pingpong(1)
↓               ↓
add_or_sub(n)   add_or_sub(n-1)
↓               ↓
add_or_sub(n-1) add_or_sub(n-2)
↓               ↓
...             ...
↓               ↓
add_or_sun(1)   add_or_sun(1)

n+1     +       n       +       ...   +  1 数量级大概是 n^2
```

而上面的参考代码的栈数量的数量级都是`n`，时间的差别自然就体现出来了。

multiadder

解法一

降order，ppt中已有

解法二

self-reference!

self-reference的本质是什么？我个人觉得是通过函数调用的frame来保存信息。

那么，这个函数要保存什么信息？还要加的数的数量`n`和已相加的数的和`curr_sum`。

这个函数的参数够保存这些信息吗？不够！所以要怎么办——helper function。

注意self-reference中，向外示人的反而是内层的`inner`函数，`helper`函数只是为了保存信息的中转站。

先搭框架。

```
def multiadder(n):
    def helper(n, curr_sum):
        def inner(x):
            ...
            return helper(...)
        return inner
    return helper(...)
```

再填细节。

```python
def multiadder(n):
    def helper(n, curr_sum):
        def inner(x):
            if n == 1:
                return curr_sum + x
            return helper(n - 1, curr_sum + x)
        return inner
    return helper(n, 0)
```

其他

可以把Maximum Subsequence、count change和分巧克力放在一起类比，给他们简单提一下分治法。