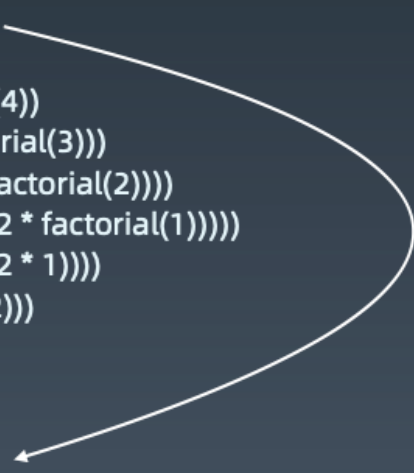


week3

## 1) 递归

### 递归 Recursion

```
factorial(6)
6 * factorial(5)
6 * (5 * factorial(4))
6 * (5 * (4 * factorial(3)))
6 * (5 * (4 * (3 * factorial(2))))
6 * (5 * (4 * (3 * (2 * factorial(1)))))
6 * (5 * (4 * (3 * (2 * 1))))
6 * (5 * (4 * (3 * 2)))
6 * (5 * (4 * 6))
6 * (5 * 24)
6 * 120
720
```



- 模版

-

### Python 代码模版

```
def recursion(level, param1, param2, ...):
    # recursion terminator
    if level > MAX_LEVEL:
        process_result
        return

    # process logic in current level
    process(level, data...)

    # drill down
    self.recursion(level + 1, p1, ...)

    # reverse the current level status if needed
```

递归终结条件

处理当前层逻辑

下探到下一层

清理当前层

```

public void recur(int level, int param) {
    // terminator
    if (level > MAX_LEVEL) {
        // process result
        return;
    }

    // process current logic
    process(level, param);

    // drill down
    recur( level: level + 1, newParam);

    // restore current status
}

```

-注意：

利用数学归纳法

不要人肉递归

找最近最简单方法，将其拆解成可重复解决的问题（重复子问题）

## 2) 分治，回溯

-分治：大问题由细的子问题组成

### 分治代码模板

```

def divide_conquer(problem, param1, param2, ...):
    # recursion terminator
    if problem is None:
        print_result
        return
    # prepare data
    data = prepare_data(problem)
    subproblems = split_problem(problem, data)
    # conquer subproblems
    subresult1 = self.divide_conquer(subproblems[0], p1, ...)
    subresult2 = self.divide_conquer(subproblems[1], p1, ...)
    subresult3 = self.divide_conquer(subproblems[2], p1, ...)
    ...
    # process and generate the final result
    result = process_result(subresult1, subresult2, subresult3, ...)

    # revert the current level states

```

-回溯：试错思想

回溯法采用试错的思想，它尝试分步的去解决一个问题。在分步解决问题的过程中，当它通过尝试发现现有的分步答案不能得到有效的正确的解答的时候，它将取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问题的答案。

回溯法通常用最简单的递归方法来实现，在反复重复上述的步骤后可能出现两种情况：

- 找到一个可能存在的正确的答案；
- 在尝试了所有可能的分步方法后宣告该问题没有答案。

在最坏的情况下，回溯法会导致一次复杂度为指数时间的计算。