Week3

递归模板

形成机械记忆，上来就写递归终止条件，否则死循环

python

```
def recursion(level, param1, param2, ...):
  # recursion terminator
  #递归终止条件
  if level > MAX_LEVEL:
    process_result
  return

  # process logic in current level
  #处理当前层逻辑
  process(level, data...)

  # drill down
  # 下探到下一层
  self.recursion(level + 1, p1, ...)

  # reverse the current level status if needed
  # 清理当前层
```

Java

```
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
}
```

要理解递归的思路并且熟练的使用它，就是要 <u>想清楚你想做什么，什么时候停止。</u>

如前序遍历：我想先打印头节点对吧？那我打印完了头节点，我现在想打印左边节点了，<u>我只是告诉计算机我想打印左边结点，之后打印右边结点</u>。

那么后序遍历呢？这个时候你应该知道了，我就是想操作左边然后右边，最后打印中间的元素。

我们并不需要太过于在意具体的递归过程，而是要想清楚让计算机干什么。

计算机都可能溢出，用人脑去遍历就不现实了吧，看开点。

分治模板

```python
# Python
def divide_conquer(problem, param1, param2, ...):
  # recursion terminator
  # 本质上是递归到了最下面的层级，即到达叶子节点，对分治来说，没有子问题了
  if problem is None:
    print_result
    return
  # prepare data
  # 处理当前层逻辑，把大问题划分成小问题
  data = prepare_data(problem)
  subproblems = split_problem(problem, data)
  # conquer subproblems
  # 下探到下一层
  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
  # 合并子结果
  result = process_result(subresult1, subresult2, subresult3, …)

  # revert the current level states
```