递归(recursion)是所有 CS 专业的学生不能避免的一个知识点，但凡提及算法的课，总会提到递归。撇去知识性的介绍，我希望能帮助大家从根本上理解如何写递归，将自己的一些理解和经验分享给大家。

递归是一个函数在其定义中调用自身的一种方法，比如下面计算 n! 的算法就是一种简单的递归：

```
factorial (n)
{
        if(n==1)
                {return 1;}
        else
                {return n*factorial(n-1); }
}
```

在此函数中，在 n 不为 1 的情况下，都会调用自身并传入 n-1 作为参数，直到 n 为 1 则开始返回。比如当第一次传入的参数 n=3 的时候：

第一次递归：判断 n=3 不为 1，调用函数自身传入参数 n-1 = 2,

第二次递归：判断 n = 2 不为 1，调用函数自身传入参数 n-1 =1,

第三次递归：此时传入的参数为 1，因此函数判断 n==1，返回 1,

返回第二次递归：上一轮函数中的 n 为 2，因此返回 2*1,

返回第一轮递归：第一轮传入的 n=3，因此最终返回 3*2*1, 达到了计算 3! 的效果。

看到这里，大家可能会觉得，这不是挺简单的吗，那么再来看看递归中的一道经典问题，棋盘覆盖问题：

在一个 $2^n \times 2^n$ 的棋盘上（n 为非 0 整数），随机放置一个 1×1 的小方块，如何用以下四种 L 型拼图将整个棋盘放满并不留空格：

分享一个大佬在 github 上制作的棋盘放置模拟器，大家可以上去体验一下并思考如何用递归来实现：https://taesungh.github.io/triomino-tiling-visualizer/

是不是觉得有些无从下手呢，不要着急，先将这道题放在一边，让我们来一起深入了解一下递归算法。

上过或正在上 ICS 6D 的小伙伴一定知道，在学习递归前，我们先学习了数学归纳（Induction）。我认为了解数学归纳法确实可以让我们更加容易理解递归，因此，让我们先来聊一聊数学归纳法。

大家应该都知道数学归纳法的基本步骤：

假如我要证明 Theorem T 在 domain 为任意非负整数中为真，则需要：

1. 证明 base case (T(1)) 为真。
2. 假设 domain 中的任意 k，T(k)为真。
3. 证明 T(k+1) 为真。

那么它的原理是什么呢，当我们同时证明了这三点之后，意味着所有 domain 中的数，都可以用它的上一步来证明为真（参照 2，3 步骤），为了证明上一步为真，我们就需要不断地返回上一步，直到第一步，也就是 base case，而我们已经证明了他为真，因此，domain 中的任何值都为真。

这听起来是不是有点像递归呢？在计算 n! 的函数中，我们的 base case 设置为 n == 1 的时候，返回 1，而剩余步骤则是不断地将 n 减少直到 n 为 1，也就是到达 base case 的时候，结束递归开始返回。

也许大家还是有些模糊，那让我们来看看这一道经典的买果汁问题：

商店有 4 瓶一盒和 5 瓶一盒的果汁，顾客无法将果汁拆开单独购买，只能一盒一盒购买，用数学归纳法证明所有 12 瓶以上的果汁都能够按盒购买。

Base case:

12 瓶果汁可以通过购买 3 盒 4 瓶一盒的果汁得到，

13 瓶果汁可以通过购买 2 盒 4 瓶一盒的果汁以及 1 盒 5 瓶一盒的果汁得到，

14 瓶果汁可以通过购买 1 盒 4 瓶一盒的果汁以及 2 盒 5 瓶一盒的果汁得到，

15 瓶果汁可以通过购买 3 盒 5 瓶一盒的果汁得到。

Induction step：

假设对任意 $k \geqslant 15$，存在一个正整数 j，$12 \leqslant j \leqslant k$，顾客能够按盒购买 j 瓶果汁。

因为 $k \geqslant 15$，我们可以得到 $k - 3 \geqslant 12$，

由此得到 $12 \leqslant k - 3 \leqslant k$，

根据假设（存在一个正整数 j，$12 \leqslant j \leqslant k$，顾客能够按盒购买 j 瓶果汁），顾客能够按盒购买 $k - 3$ 瓶果汁。

因此，当顾客想要购买 k + 1 瓶果汁的时候，只需要在购买 k – 3 瓶果汁的基础上，再购买 1 盒 4 瓶一盒的果汁就可以了。

简单来说，当我们想要购买 15 瓶以上的果汁，比如 17 瓶，我们只需要先买一盒 4 瓶一盒的果汁，17 - 4 = 13，然后根据 base case 购买 13 瓶果汁的方法购买剩余果汁即可，就算是更大的数量，比如 100，我们可以不断的增加 4 瓶一盒果汁的数量，将 100 不断-4，最终达到 4 种 base case 中的任意一种。

而这正是递归解题的思路：

```
void BuyJuice(unsigned& num_4_pack, unsigned & num_5_pack, unsigned n)
{
	if (n == 12)                        //首先定义四种base case
	{
		num_4_pack = 3;
		return;
	}
	else if (n == 13)
	{
		num_4_pack = 2;
		num_5_pack = 1;
		return;
	}
	else if (n == 14)
	{
		num_4_pack = 1;
		num_5_pack = 2;
		return;
	}
	else if (n == 15)
	{
		num_5_pack = 3;
		return;
	}
	else
	{
		BuyJuice(num_4_pack, num_5_pack, n - 4); //总数减去4瓶，传入下一次递归，
                                                 //由于base case会分配4瓶1盒果汁的
                                                 //量，增加盒数应该写在函数调用之后。
		num_4_pack++;//增加1盒4瓶一盒的果汁
		return;
	}
}
```

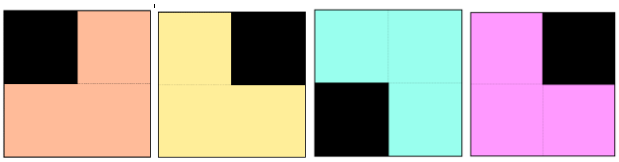
看到这里，我想各位对数学归纳和递归都有了一定的了解了，并且可以感觉到这两者之间存在着某种联系。

用我的话来讲：数学归纳是扩展 base case，而递归则是将复杂的问题不断简化直到到达 base case，因此如何写好递归函数，最重要的就是找到 base case 已经如何简化并将参数传入下一次递归。

现在让我们回过头来看看一看棋盘放置问题。

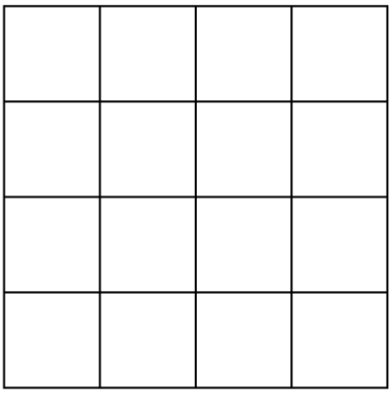
各位可以先花时间想一想，这道题的 base case 是什么。

没错，这道题的 base case 就是在 2 × 2 的棋盘上放置一个黑色方块后分别根据情况放置四种拼图：

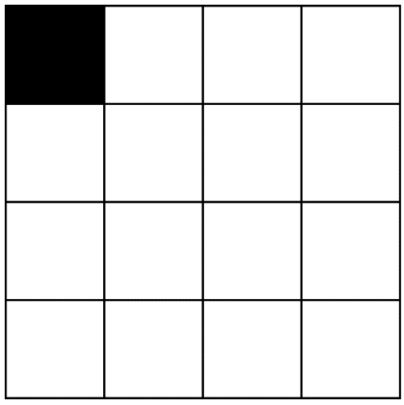


那么下一步，如何将棋盘逐步递归为 base case 呢？

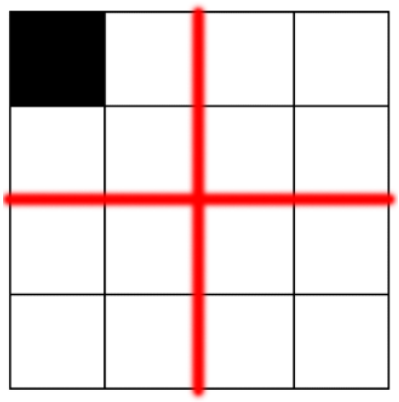
接下来拿 4 × 4 方格举例：



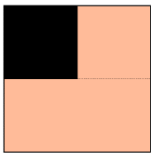
当我们在左上角放置黑色方块的时候：



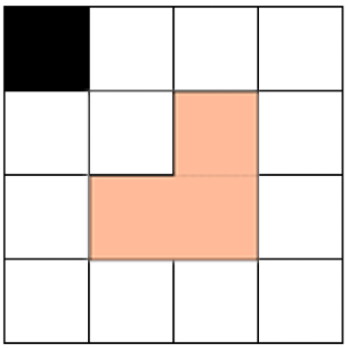
我们得到了这样的棋盘，再将 4 × 4 棋盘划分为四个 2 × 2 棋盘：



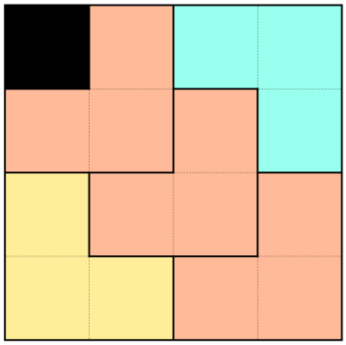
这时，我们可以观察到，除了第二象限的棋盘，也就是左上角的棋盘，其余三个象限都是空白，因此我们可以在棋盘中心放置一个



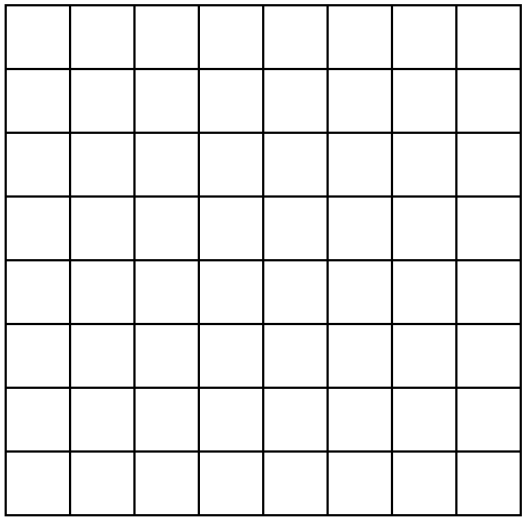
类型的拼图，来使得四个象限的棋盘都缺失一块：



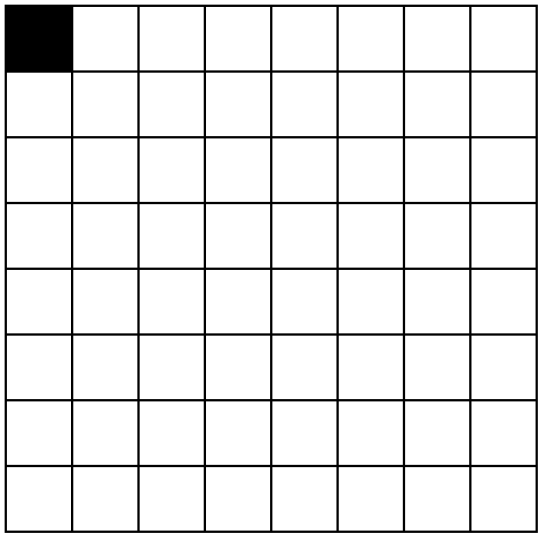
再次观察四个象限，我们得到了四个 base case，只需要依次放置拼图即可：



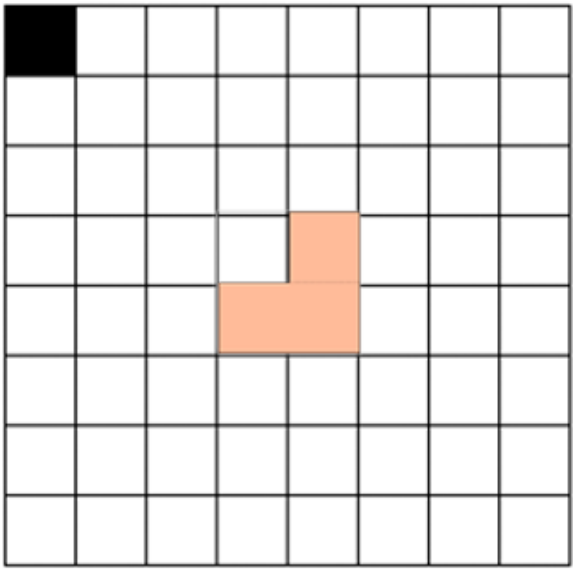
让我们更进一步，来看看 8 × 8 的棋盘：



此时我们再次在左上角放置黑色方块：

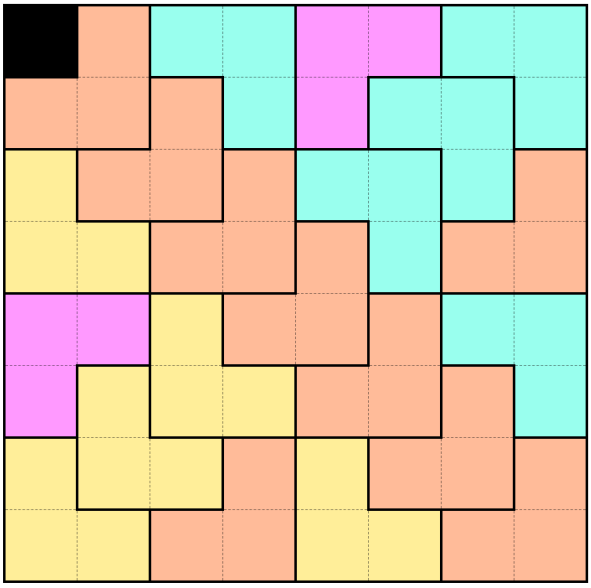


根据四个象限的空白情况，在中心放置拼图：



如果将棋盘再次按照象限划分，我们得到了 4 个 4 × 4 的棋盘，而正是我们刚刚所遇到的情况。

最终可以得到：



因此，对于棋盘放置问题，我们只需要将每次将棋盘分成 4 个小棋盘，

进行四次循环分别判断四个小棋盘：

1. 如果存在黑色方格，则再将其分为四个小棋盘进行递归。
2. 如果不存在黑色方格，则假定最靠近中心方格为黑色方格，进行递归。

直到棋盘可以用 base case 放置拼图为止。

代码略长，篇幅有限，若是有兴趣可以自行搜索棋盘覆盖问题。

总而言之，递归是一种将复杂问题分解成同类型小问题直到 base case 的方法，因此想要写好递归，需要考虑在哪些情况下需要停止递归，也就是 base case，以及如何分解大问题并进行下一次递归。

感谢各位能看到这里，希望能对你们理解递归有一些帮助!