

漫画：什么是动态规划？（整合版）

2017-06-21 算法爱好者

(点击上方公众号，可快速关注)

来源：微信公众号——梦见 (dreamsee321)

作者：玻璃猫

[如有好文章投稿，请点击 → 这里了解详情](#)

小灰，听说你又去面试了？
结果怎么样？



哎，别提了……



呃，面试官考你什么了，
说来听听？



当时，面试官是这么考我
的.....



小灰是吧？请简单介绍一下你自己。



好的！

blah blah blah



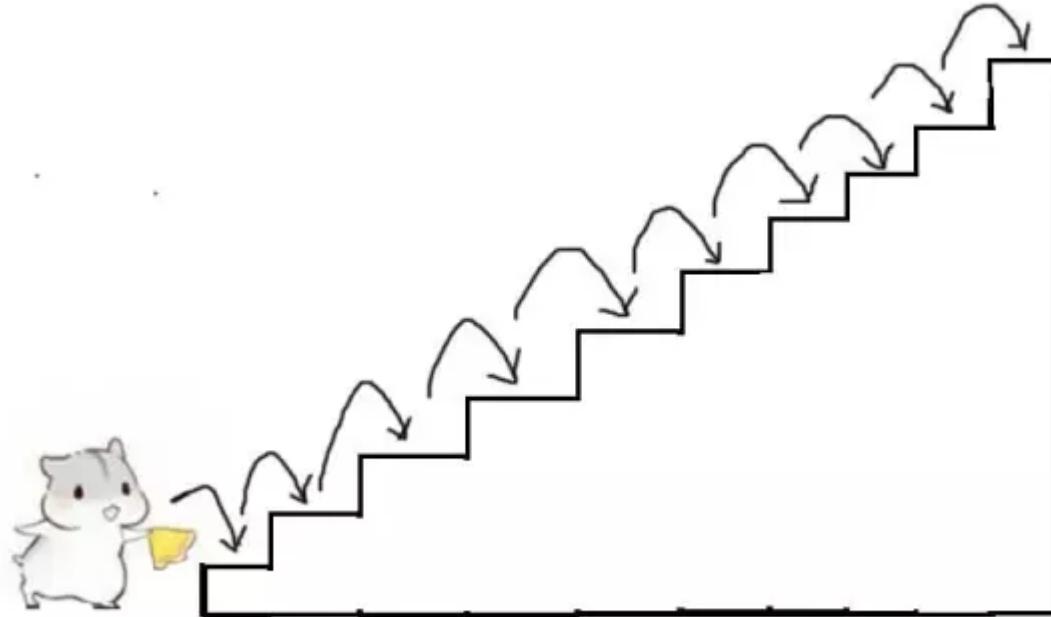
下面考你一道算法题。有一座高度是 10 级的楼梯，让你从下往上走



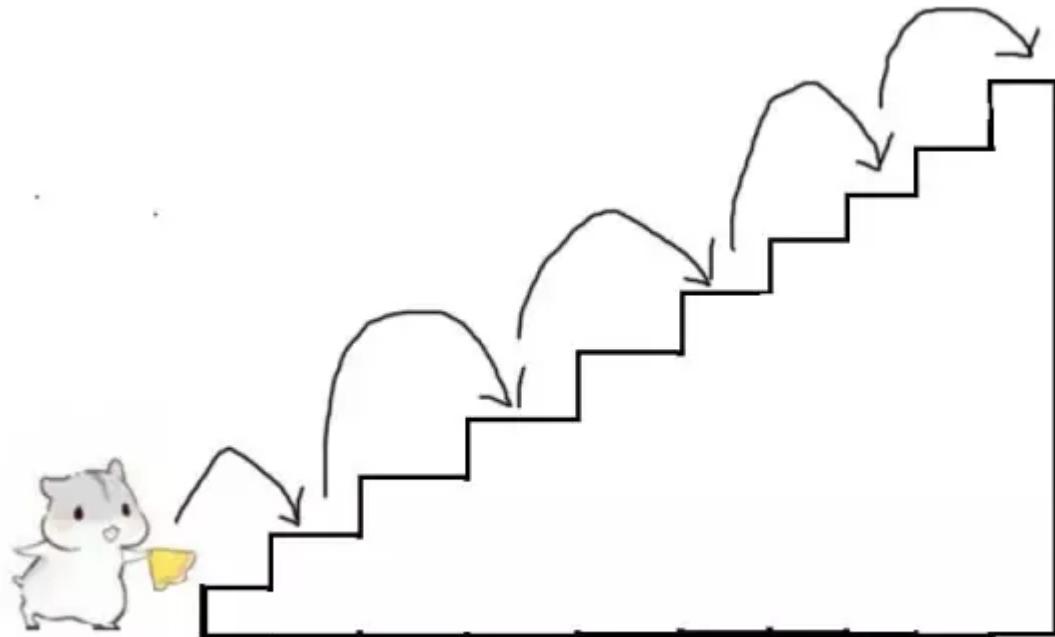
题目：

有一座高度是**10**级台阶的楼梯，从下往上走，每跨一步只能向上**1**级或者**2**级台阶。要求用程序来求出一共有多少种走法。

比如，每次走1级台阶，一共走10步，这是其中一种走法。我们可以简写成
1,1,1,1,1,1,1,1,1,1。



再比如，每次走2级台阶，一共走5步，这是另一种走法。我们可以简写成 2,2,2,2,2。



当然，除此之外，还有很多很多种走法。

额，让我想想……



有了！咱们可以利用排列组合的思想，
写一个多层次嵌套循环遍历出所有的可能
性。每遍历出一个组合，让计数器加一。



呵呵，你这个方法属于暴力枚举，时间复杂度是指数级的。
有没有更高效的解法？



额，让我想想……



要不咱们找个楼梯走一下试试吧，
正好能减肥！



呵呵，不用试啦，
直接回家等通知吧！



哈哈，小灰，你一定没学
过【动态规划】吧？



动态规划？那是什么鬼？



动态规划的英文名 Dynamic Programming，是一种分阶段求解决策问题的数学思想。它不止用于编程领域，也应用于管理学、经济学、生物学。



听起来好高大上呀，就是不怎么懂，
能不能讲的直白一点？



总结起来就是一句话：
大事化小，小事化了。



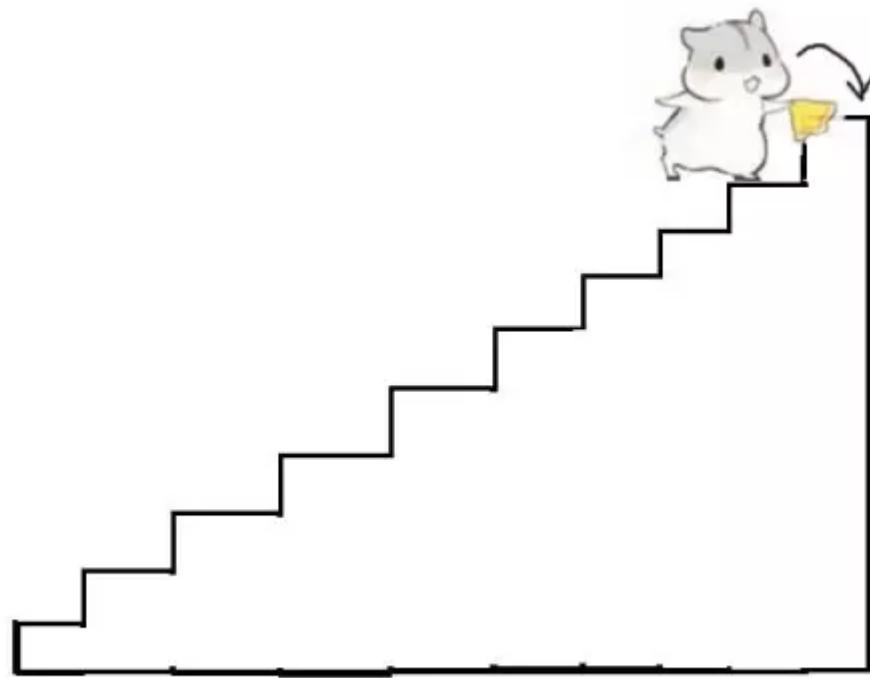
拿刚才的面试题目来说，假设你
只差最后一步就走到第 10 级台
阶，这时候会出现几种情况？



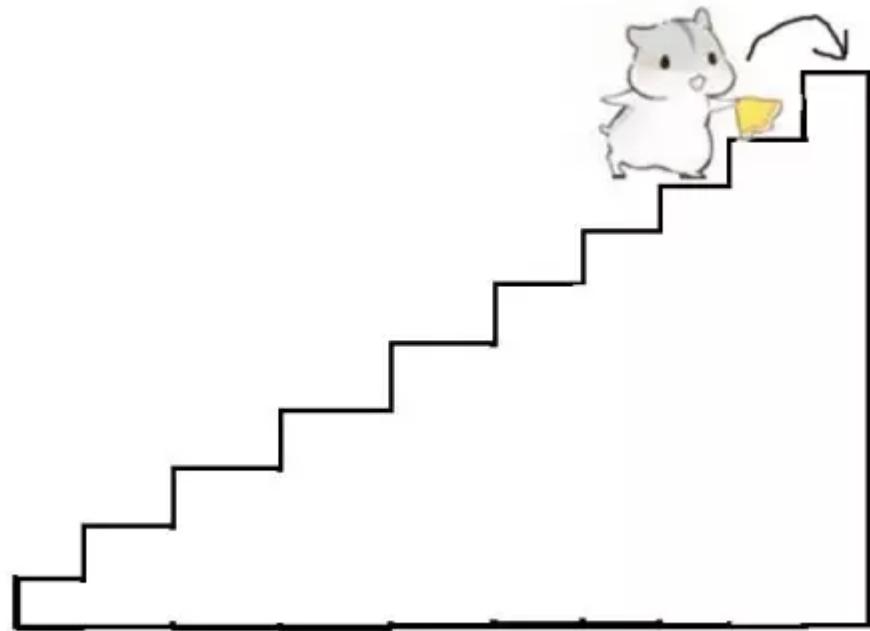
当然是两种情况喽，因为每一步只许
走 1 级或是 2 级台阶嘛。第一种是从
9 级走到 10 级，第二种是从 8 级走
到 10 级。



第一种情况：



第二种情况：



Bingo！咱们暂且不管从 0 走到 8 级台阶的过程，也不管从 0 级走到 9 级台阶的过程。想要走到第 10 级，最后一步必然是从 8 级或者 9 级开始。



接下来引申出一个新的问题：如果我们已知 0 到 9 级台阶的走法有 X 种，0 到 8 级台阶的走法有 Y 种，那么 0 到 10 级台阶的走法有多少种？



让我想想……

走到第 10 级台阶需要从第 8 级或是第 9 级开始，而 8 级和 9 级的走法数量又已经知道，那么这三者的关系是……



我想明白了！10 级台阶的所有走法可以根据最后一步的不同而分成两部分，第一部分的最后一步是从 9 级到 10 级，这部分的走法数量和 9 级台阶的走法数量是相等的，也就是 X。



第二部分的最后一步是从 8 级到 10 级，这部分的走法数量和 8 级台阶的走法数量是相等的，也就是 Y。这两部分相加，总的走法数量是 $X + Y$ ！



把思路画出来，就是这样子：

0到10级台阶的所有走法 X+Y

1->2->2->1->1->1->1

2->1->2->2->1->1

..... 0到9级台阶的
所有走法 X
.....

1->2->1->1->1->1->1

1->1->1->2->1->1->1

..... 0到8级台阶的
所有走法 Y
.....

-> 1

-> 1

-> 1

-> 2

-> 2

-> 2

-> 2

最后一步
的走法

非常好，这一步推断确实有些绕。这样
样一来就可以得出一个结论：

从 0 到 10 级台阶的走法数量 = 0 到 9
级的走法数量 + 0 到 8 级的走法数量。



为了方便表达，我们把 10 级台阶的走法数量简写为 $F(10)$ ，此时 $F(10) = F(9) + F(8)$ 。那么，我们如何计算 $F(9)$ 和 $F(8)$ 呢？



我明白了。利用刚才的思路可以很容易推断出：

$$F(9) = F(8) + F(7), \quad F(8) = F(7) + F(6)$$



没错，看到了吗？我们正在把一个复杂的问题分阶段进行简化，逐步简化成简单的问题。这就是动态规划的思想。



当只有 1 级台阶和两级台阶的时候，
有几种走法呢？显然分别是 1 和 2。
由此，我们可以归纳出如下的公式：



$$\begin{aligned}F(1) &= 1; \\F(2) &= 2; \\F(n) &= F(n-1)+F(n-2) \quad (n>=3)\end{aligned}$$

动态规划当中包含三个重要的概念：
【最优子结构】、【边界】、【状态转移公式】。



刚才我们分析出 $F(10) = F(9) + F(8)$ ，
因此 $F(9)$ 和 $F(8)$ 是 $F(10)$ 的【最优子
结构】。



当只有 1 级台阶或 2 级台阶时，我们可以直接得出结果，无需继续简化。我们称 $F(1)$ 和 $F(2)$ 是问题的【边界】。如果一个问题没有边界，将永远无法得到有限的结果。



$F(n) = F(n-1) + F(n-2)$ 是阶段与阶段之
间的【状态转移方程】。这是动态规划
的核心，决定了问题的每一个阶段和下
一阶段的关系。



哇塞，厉害了我的哥！



别高兴得太早，我们只完成了动态规划的前半部分：问题建模。下面才是真正麻烦的阶段：求解问题。



小灰，你先仔细思考一下吧。



那个啊... 我想了一下, 很简单,
不就是一个递归吗 ?



既然已经归纳出了 $F(N) = F(N-1) + F(N-2)$ ，
又知道了递归结束的条件，我们就可以直接用递
归的思路写程序。看，代码已经写好了：



方法一：递归求解

```
int getClimbingWays(int n) {  
    if(n < 1) {  
        return 0;  
    }  
  
    if(n == 1) {  
        return 1;  
    }  
  
    if(n == 2) {  
        return 2;  
    }  
  
    return getClimbingWays(n-1) + getClimbingWays(n-2);  
}
```

由于代码比较简单，这里就不做过多解释了。

OK，这样的的确可以计算出最终答案。可是，你想过这个方法的时间复杂度吗？



啊呀，这我倒是没想过...
时间复杂度该怎么计算呢？

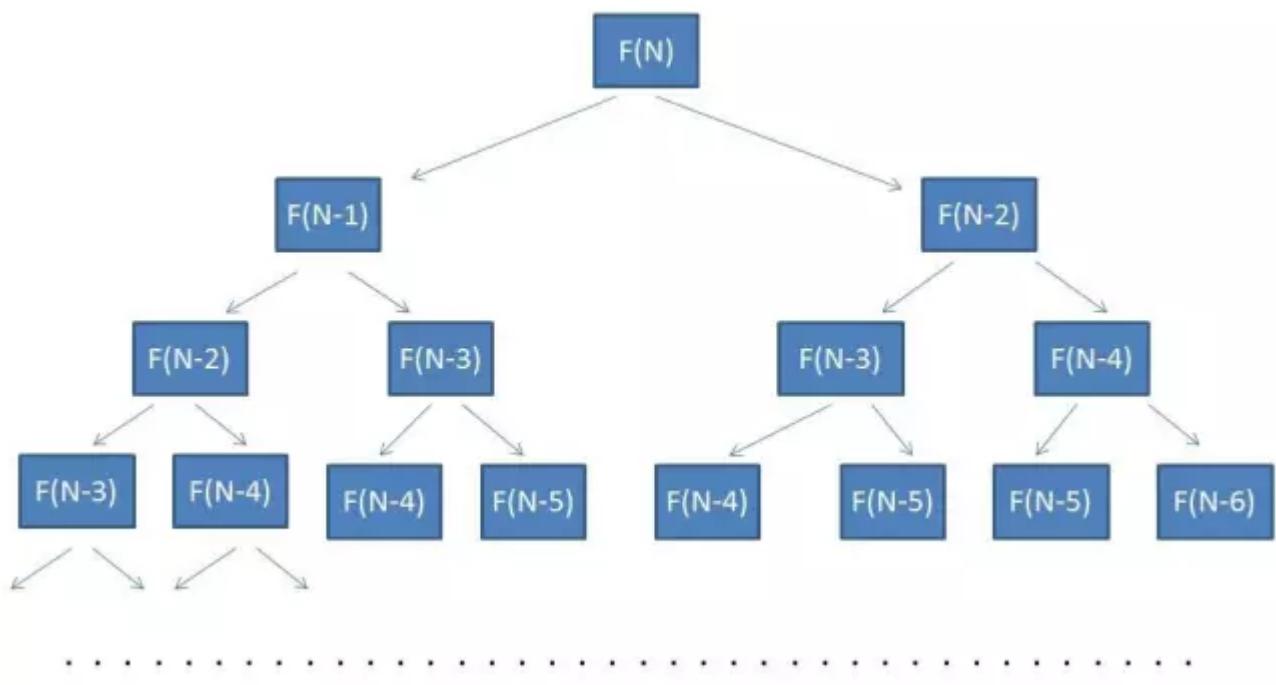


计算时间复杂度并不太难，让我们来分析一下递归方法所走过的路径吧。



要计算出 $F(N)$ ，就要先得到 $F(N-1)$ 和 $F(N-2)$ 的值。要计算 $F(N-1)$ ，就要先得到 $F(N-2)$ 和 $F(N-3)$ 的值... 以此类推，可以归纳成下面的图：





看起来还挺复杂，像是一棵
二叉树。



没错，这就是一颗二叉树，树
的节点个数就是我们的递归方
法所需要计算的次数。



不难看出，这颗二叉树的高度是 $N-1$ ，节点个数接近 2^{N-1} 。所以方法的时间复杂度可以近似地看作是 $O(2^N)$ 。



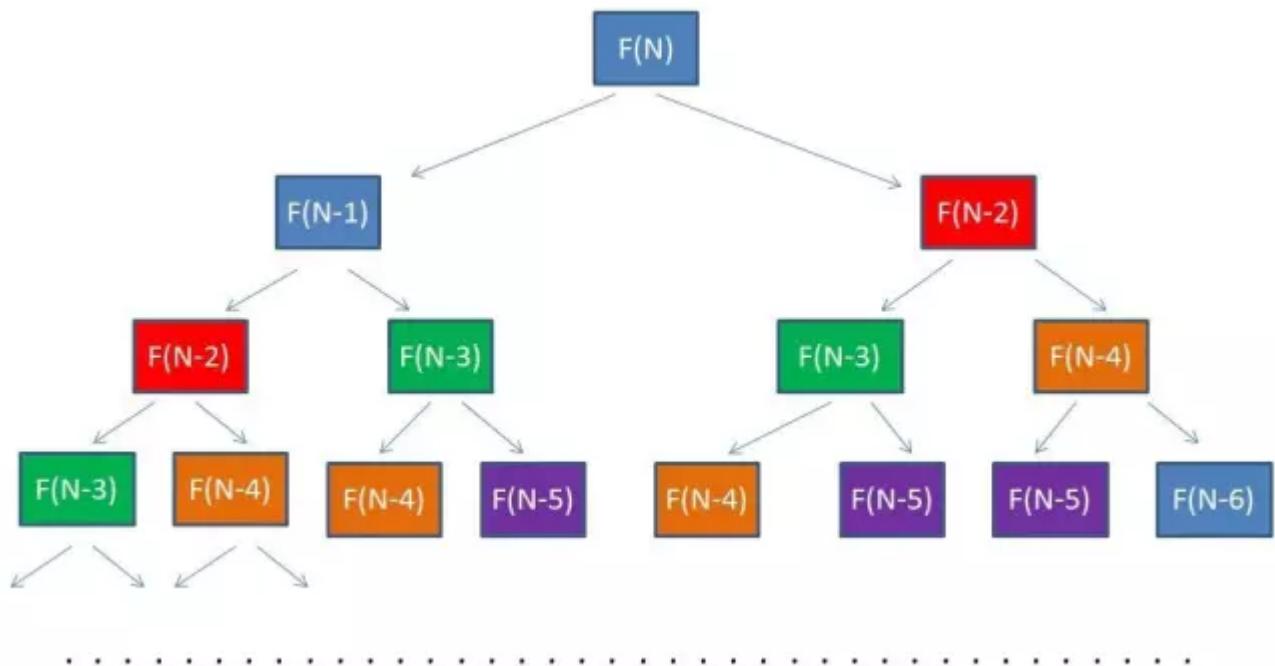
小灰，你想想有什么办法能优化一下呢？



这效率确实够低的，让我想想啊.....



回顾一下刚才的递归图，我觉得
有些相同的参数被重复计算了。
越往下走，重复的越多。



如图所示，相同颜色代表了方法被传入相同的参数。

观察的很好！对于这种重复计算的情况，应该怎么避免呢？



我知道了，用缓存！先创建一个哈希表，每次把不同参数的计算结果存入哈希。当遇到相同参数时，再从哈希表里取出，就不用重复计算了。



没错，这种暂存计算结果的方式有一个很贴切的名字，叫做【备忘录算法】。来，把刚才的思路实现一下吧。



方法二：备忘录算法

```

int getClimbingWays(int n, HashMap<Integer, Integer> map) {

    if(n < 1) {
        return 0;
    }

    if(n == 1) {
        return 1;
    }

    if(n == 2) {
        return 2;
    }

    if(map.contains(n)) {
        return map.get(n);
    }else{
        int value = getClimbingWays(n-1) + getClimbingWays(n-2);
        map.put(n,value);
        return value;
    }
}

```

在以上代码中，集合map是一个备忘录。当每次需要计算F(N)的时候，会首先从map中寻找匹配元素。如果map中存在，就直接返回结果，如果map中不存在，就计算出结果，存入备忘录中。

不错，这就是备忘录算法。你说说这个算法的时间复杂度和空间复杂度分别是多少？



这个容易算。从 $F(1)$ 到 $F(N)$ 一共有 N 个不同的输入，在哈希表里存了 $N-2$ 个结果，所以时间复杂度和空间复杂度都是 $O(N)$ 。



是的，现在我们的程序性能已经得到了明显优化，但这样还并不是真正的动态规划实现。



时间复杂度已经不能再小了。小灰，你想一想，咱们还能不能把空间复杂度进一步减小？



我的天，还要继续优化呀 ...

让我想想



不行了，想不出来



哈哈，没关系。借用鲁迅先生的一句名言：
我们不妨把思路逆转过来！





呸，鲁迅啥时候说过这句话。
怎么把思路逆转呢？



想想看，我们一定要对 $F(N)$ 自顶
向下做递归运算吗？可不可以自底
向上，用迭代的方式推导出结果？



什么自顶向上，自底向下的...

听不明白.....



好吧，让我通过一张表格，来说明一下 $F(N)$ 自底向上求解的过程。



台阶数	1	2	3	4	5	6	7	8	9
走法数	1	2							

看，表格的第一行代表了楼梯台阶的数目，第二行代表了若干级台阶对应的走法数。 $F(1)=1, F(2)=2$ ，这是之前就已经明确过的结果。



台阶数	1	2	3	4	5	6	7	8	9
走法数	1	2	3						

第一次迭代，台阶数等于 3 时，走法数量是 3。这个结果怎么来的呢？是 $F(1), F(2)$ 这两个结果相加得到的。所以 $F(3)$ 只依赖于 $F(1)$ 和 $F(2)$ 。



台阶数	1	2	3	4	5	6	7	8	9
走法数	1	2	3	5					

第二次迭代，台阶数等于 4 时，走法数量是 5。这是 $F(2), F(3)$ 这两个结果相加得到的。所以 $F(4)$ 只依赖于 $F(2)$ 和 $F(3)$ 。



台阶数	1	2	3	4	5	6	7	8	9
走法数	1	2	3	5	8				

同理，在后续的迭代中， $F(5)$ 只依赖于 $F(4), F(3)$ ； $F(6)$ 只依赖于 $F(5), F(4)$ 。



由此可见，每一次迭代过程中，只要保留之前的两个状态，就可以推导出新的状态。而不需要像备忘录算法那样保留全部的子状态。



这样才是真正的动态规划实现，
让我们一起看看代码吧。



方法三：动态规划求解

```
int getClimbingWays(int n) {  
  
    if(n < 1){  
        return 0;  
    }  
  
    if(n == 1){  
        return 1;  
    }  
  
    if(n == 2){  
        return 2;  
    }  
  
    int a = 1;  
    int b = 2;  
    int temp = 0;  
  
    for(int i=3; i<=n; i++){  
        temp = a + b;  
        a = b;  
        b = temp;  
    }  
  
    return temp;  
}
```

程序从 $i=3$ 开始迭代，一直到 $i=n$ 结束。每一次迭代，都会计算出多一级台阶的走法数量。迭代过程中只需保留两个临时变量a和b，分别代表了上一次和上上次迭代的结果。为了便于理解，我引入了temp变量。temp代表了当前迭代的结果值。

原来这才是动态规划的代码实现，
看起来好简洁，厉害了我的哥！



嘿嘿，你看看这个方法的时间复杂度和空间复杂度是多少？



时间复杂度显然是 $O(N)$ ，由于只
引入了两个或三个变量，所以空
间复杂度只有 $O(1)$ ！



是的呢，这就是动态规划，利用简洁的自底向上的递推方式，实现了时间和空间上的最优化。



不过，这道上楼梯的题目仅仅是动态规划领域中最最简单的问题，因为它只有一个变化维度。还有许多问题远比这要复杂得多。



哎呀我滴乖乖，这还只是最简单的？



哈哈，你以为呢？下面给你出一道相对复杂的题目。弄懂了这个问题，才算是真正理解了动态规划。



题目二：国王和金矿

有一个国家发现了5座金矿，每座金矿的黄金储量不同，需要参与挖掘的工人数也不同。参与挖矿工人的总数是10人。每座金矿要么全挖，要么不挖，不能派出一半人挖取一半金矿。要求用程序求解出，要想得到尽可能多的黄金，应该选择挖取哪几座金矿？



总共 10 名工人



小灰，你先仔细思考一下吧。



这个...我想了半天，还是不知道
怎么用动态规划来建模，只好依旧
采用排列组合的方式.....



用排列组合也没关系，但说无妨。



恩，我是这样想的.....



方法一：排列组合

每一座金矿都有挖与不挖两种选择，如果有 N 座金矿，排列组合起来就有 2^N 种选择。对所有可能性做遍历，排除那些使用工人数超过10的选择，在剩下的选择里找出获得金币数最多的选

代码比较简单就不展示了，时间复杂度也很明显，就是 $O(2^N)$ 。

OK，排列组合的时间复杂度是指指数级，缺点不用我多说了。让我们再次用动态规划的思路来分析一下这道题吧。



动态规划有三个核心元素：最优化结构、边界、状态转移方程式。
小灰，你想一想这个问题的最优化结构是什么？

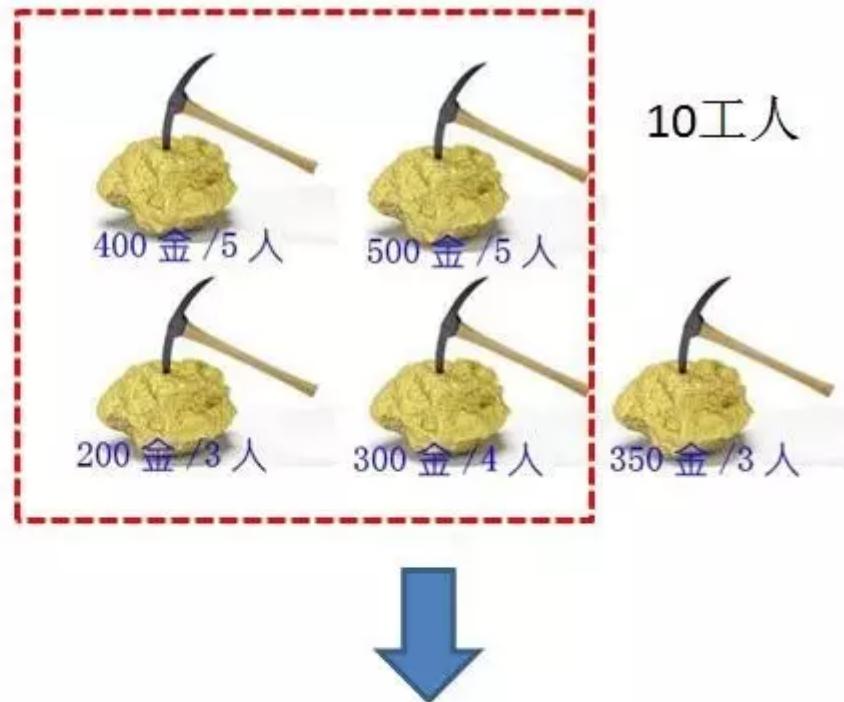


让我想想... 题目是要求出 10 个工人 5 个金矿时，挖最多黄金的选择。那么最优化结构应该是 10 个工人 4 个金矿时，挖出最多黄金的选择吧？



最优子结构：

10人4金矿的
最优选择



最终问题：

10人5金矿的
最优选择



只能说你回答对了一半。第 5 个金矿存在挖与不挖两种选择。如果选择不挖，对应的最优子结构是你所说的情况。

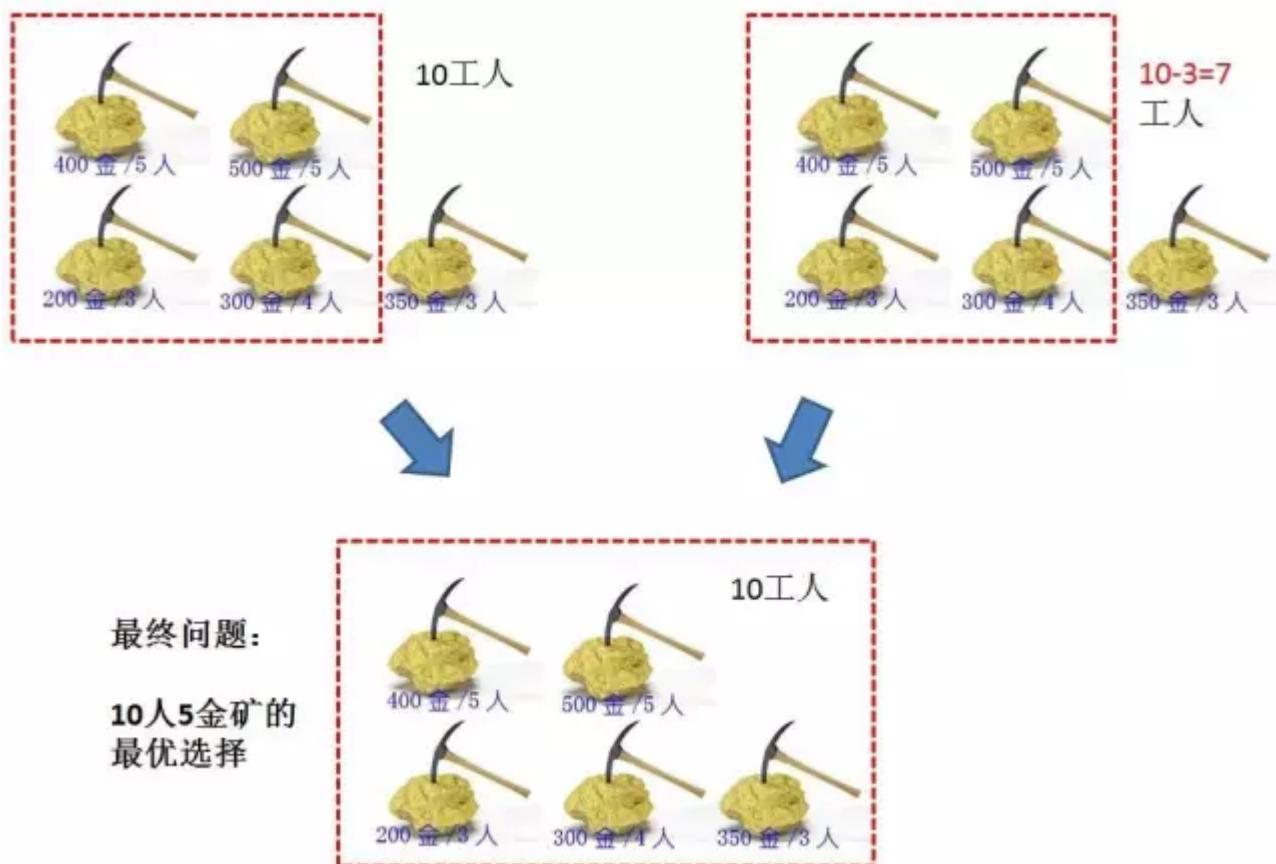


如果选择挖，会占用掉一部分工人，那么前 4 个金矿所分配的工人数量就是 $[10 - \text{第 } 5 \text{ 个金矿所需人数}]$ ，而不是 10 了。

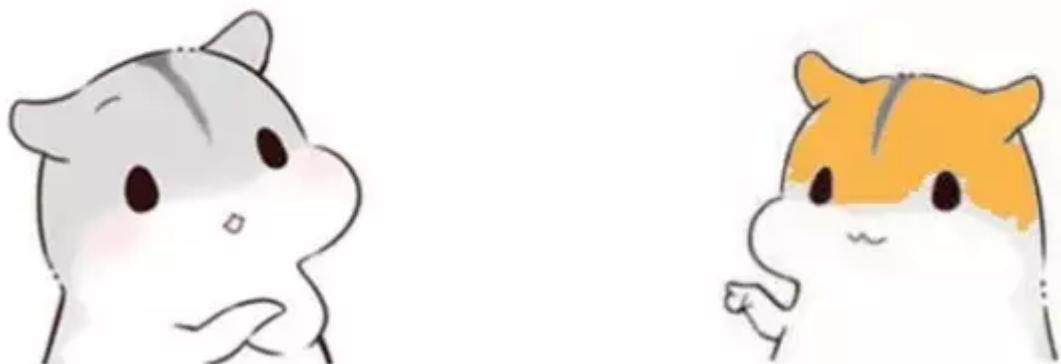


所以说，问题的最优子结构有两个，一个是 4 金矿 10 工人时的最优选择，一个是 4 金矿 10-3 工人时的最优选择。





原来如此，刚才少考虑了
一种情况。



既然已找到了最优子结构，那么我们来分析一下最优子结构和最终问题的关系。换句话说，4个金矿的最优选择和5个金矿的最优选择之间，是什么样的关系？

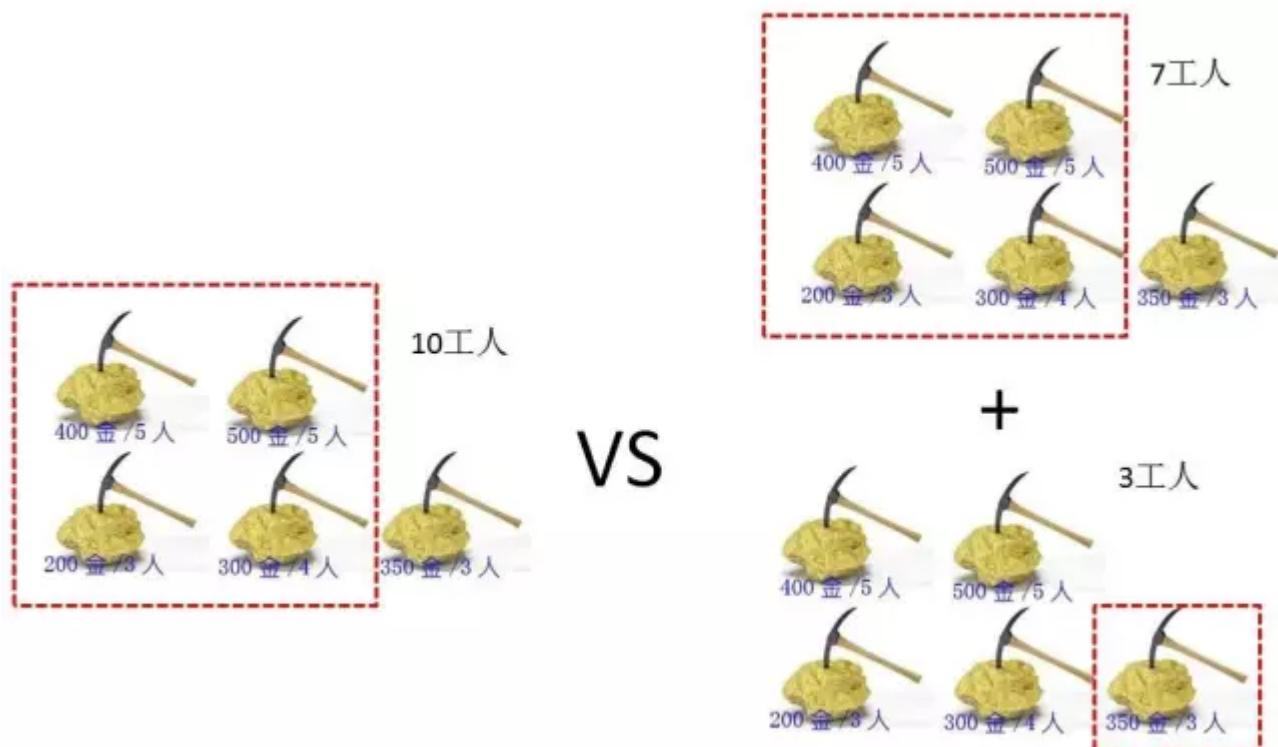


让我想想...既然4个金矿时的最优子结构有两个，它们和5个金矿时的最优选择是什么关系呢？



我想明白了！5个金矿的最优选择，就是（前4座金矿10工人的挖金数量）和（前4座金矿7工人的挖金数量+第5座金矿的挖金数量）的最大值！





Bingo！为了便于描述，我们把金矿数量设为 N ，工人数设为 W ，金矿的黄金量设为数组 $G[]$ ，金矿的用工量设为数组 $P[]$ 。



那么 5 座金矿和 4 座金矿的最优选择之间
存在这样的关系： $F(5, 10) =$
 $\max(F(4, 10), F(4, 10 - P[4]) + G[4])$



最后还需要确定一下，这个问题的
边界是什么？



边界自然是只有 1 座金矿，也就是 $N=1$
的时候。这时候没得可选，只能挖这座
唯一的金矿，得到的黄金数量就是 $G[0]$ 。



哈哈，你又遗漏了一个情况。如果给定的工人数量不够挖取第 1 座金矿，也就是 $W < P[0]$ 的时候，那么得到的黄金数就是 0 了。



用公式来表达的话，

当 $N=1, W \geq P[0]$ 时， $F(N, W) = G[0]$ ；

当 $N=1, W < P[0]$ 时， $F(N, W) = 0$ 。



哎呀，想不到需要考虑的情况
有这么多……



下面经过整理，我们就可以得到问题的状态转移方程式：



$$F(n, w) = 0 \quad (n \leq 1, w < p[0]);$$

$$F(n, w) = g[0] \quad (n == 1, w \geq p[0]);$$

$$F(n, w) = F(n-1, w) \quad (n > 1, w < p[n-1])$$

$$F(n, w) = \max(F(n-1, w), F(n-1, w-p[n-1])+g[n-1]) \quad (n > 1, w \geq p[n-1])$$

其中第三条是补充上去的，原因不难理解。

哎我去，还真挺复杂的。



好了，我们的动态规划建模工作已经完成，下面开始实现环节。小灰，还记得我之前所讲的几种实现方法吗？



当然记得哦，有简单递归、备忘录算法、动态规划这三种实现方法。



方法二：简单递归

把状态转移方程式翻译成递归程序，递归的结束的条件就是方程式当中的边界。因为每个状态有两个最优子结构，所以递归的执行流程类似于一颗高度为N的二叉树。

方法的时间复杂度是 $O(2^N)$ 。

方法三：备忘录算法

在简单递归的基础上增加一个HashMap备忘录，用来存储中间结果。HashMap的Key是一个包含金矿数N和工人数W的对象，Value是最优选择获得的黄金数。

方法的时间复杂度和空间复杂度相同，都等同于备忘录中不同Key的数量。

至于动态规划方法的实现，我是实在是没想出来... 这个问题的参数有两个，也就是存在两个输入维度，怎么能实现自底向上的递推呢？



别着急，我们先画一个表格来做分析。



	1工人	2工人	3工人	4工人	5工人	6工人	7工人	8工人	9工人	10工人
1金矿										
2金矿										
3金矿										
4金矿										
5金矿										

表格的第一列代表给定前 1-5 座金矿的情况，也就是 N 的取值。表格的第一行代表给定的工人数，也就是 W 的取值。



表格中其余的空白格，代表给定 N 和 W 值对应的黄金获得数，也就是 $F(N, W)$ 。下面让我们来逐行填写表格的空白。



还记得第 1 个金矿的信息吗？400 金，5 工人。所以前 4 个格子都是 0，因为人 数不够嘛。后面格子都是 400，因为只 有这一座金矿可挖。



	1工人	2工人	3工人	4工人	5工人	6工人	7工人	8工人	9工人	10工人
1金矿	0	0	0	0	400	400	400	400	400	400
2金矿										
3金矿										
4金矿										
5金矿										

第一行其实就是我们刚才分析的问题边界，因此可以直接得出结果。从第二行开始就复杂一些了。



第 2 座金矿有 500 黄金，需要 5 工人。
第 2 行前 4 个格子怎么计算呢？因为
 $W < 5$ ，所以 $F(N, W) = F(N-1, W) = 0$ 。



	1工人	2工人	3工人	4工人	5工人	6工人	7工人	8工人	9工人	10工人
1金矿	0	0	0	0	400	400	400	400	400	400
2金矿	0	0	0	0						
3金矿										
4金矿										
5金矿										

第 2 行后 6 个格子怎么计算呢？因为 $W \geq 5$ ，所以根据 $F(N, W) = \max(F(N-1, W), F(N-1, W-5) + 500)$ ，第 5-9 个格子的值是 500。



	1工人	2工人	3工人	4工人	5工人	6工人	7工人	8工人	9工人	10工人
1金矿	0	0	0	0	400	400	400	400	400	400
2金矿	0	0	0	0	500	500	500	500	500	
3金矿										
4金矿										
5金矿										

需要注意的是第 2 行第 10 个格子，也就是 $N=2, W=10$ 的时候， $F(N-1, W)=400$ ， $F(N-1, W-5)=400$ ， $\max(400, 400+500)=900$



	1工人	2工人	3工人	4工人	5工人	6工人	7工人	8工人	9工人	10工人
1金矿	0	0	0	0	400	400	400	400	400	400
2金矿	0	0	0	0	500	500	500	500	500	900
3金矿										
4金矿										
5金矿										

第3座金矿有200黄金，需要3工人，
第3行的计算方法如出一辙。



	1工人	2工人	3工人	4工人	5工人	6工人	7工人	8工人	9工人	10工人
1金矿	0	0	0	0	400	400	400	400	400	400
2金矿	0	0	0	0	500	500	500	500	500	900
3金矿	0	0	200	200	500	500	500	700	700	900
4金矿										
5金矿										

第4座金矿有300黄金，需要4工人，
第5座金矿有350黄金，需要3工人。
计算方法同上。



	1工人	2工人	3工人	4工人	5工人	6工人	7工人	8工人	9工人	10工人
1金矿	0	0	0	0	400	400	400	400	400	400
2金矿	0	0	0	0	500	500	500	500	500	900
3金矿	0	0	200	200	500	500	500	700	700	900
4金矿	0	0	200	300	500	500	500	700	800	900
5金矿										

	1工人	2工人	3工人	4工人	5工人	6工人	7工人	8工人	9工人	10工人
1金矿	0	0	0	0	400	400	400	400	400	400
2金矿	0	0	0	0	500	500	500	500	500	900
3金矿	0	0	200	200	500	500	500	700	700	900
4金矿	0	0	200	300	500	500	500	700	800	900
5金矿	0	0	350	350	500	550	650	850	850	900

小灰，对于每个格子数值的推导，发现什么规律没？



似乎除了第1行以外，每个格子都是前一行的一个或两个格子推导而来。



比如 3 金矿 8 工人的结果，就来自于 2 金矿 5 工人和 2 金矿 8 工人，
 $\text{Max}(500, 500+200)=700。$



	1工人	2工人	3工人	4工人	5工人	6工人	7工人	8工人	9工人	10工人
1金矿	0	0	0	0	400	400	400	400	400	400
2金矿	0	0	0	0	500	500	500	500	500	900
3金矿	0	0	200	200	500	500	500	700	700	900
4金矿	0	0	200	300	500	500	500	700	800	900
5金矿	0	0	350	350	500	550	650	850	850	900

再比如 5 金矿 10 工人的结果，就来自于 4 金矿 7 工人和 4 金矿 10 工人，
 $\text{Max}(900, 500+350)=900。$



	1工人	2工人	3工人	4工人	5工人	6工人	7工人	8工人	9工人	10工人
1金矿	0	0	0	0	400	400	400	400	400	400
2金矿	0	0	0	0	500	500	500	500	500	900
3金矿	0	0	200	200	500	500	500	700	700	900
4金矿	0	0	200	300	500	500	500	700	800	900
5金矿	0	0	350	350	500	550	650	850	850	900

说的没错！我们使用程序实现的时候，也可以像这样从左至右，从上到下一格一格推导出最终结果。



而且我们并不需要存储整个表格，只需要存储前一行的结果，就可以推导出新的一行。我们来写一下代码看看吧。



方法四：动态规划

```

int getMostGold(int n, int w, int[] g, int[] p) {
    int[] preResults = new int[p.length];
    int[] results = new int[p.length];

    //填充边界格子的值
    for(int i=0; i<=n; i++) {
        if(i < p[0]){
            preResults[i] = 0;
        }else{
            preResults[i] = g[0];
        }
    }

    //填充其余格子的值，外层循环是金矿数量，内层循环是工人数
    for(int i=0; i<n; i++) {

        for(int j=0; j<=w; j++) {
            if(j < p[i]){
                results[j] = preResults[j];
            }else{
                results[j] = Math.max(preResults[j], preResults[j-p[i]] + g[i]);
            }
        }

        preResults = results;
    }

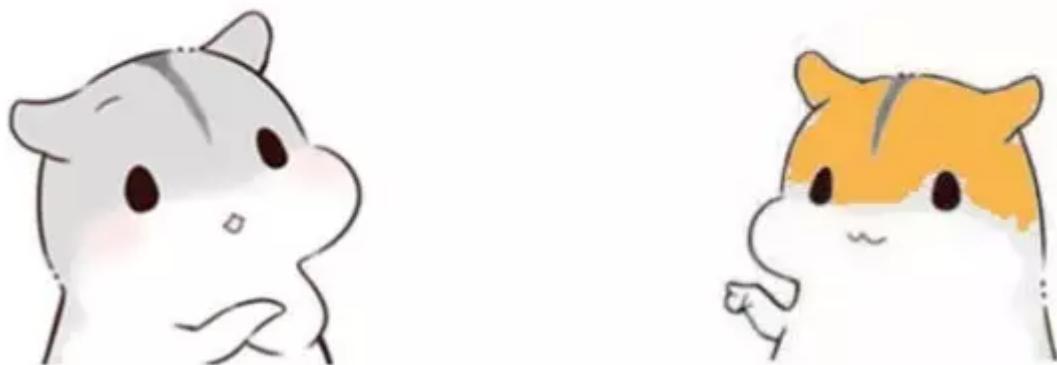
    return results[n];
}

```

方法利用两层迭代，来逐步推导出最终结果。在外层的每一次迭代，也就是对表格每一行的迭代过程中，都会保留上一行的结果数组 preResults，并循环计算当前行的结果数组 results。

方法的时间复杂度是 $O(n * w)$ ，空间复杂度是 (w) 。需要注意的是，当金矿只有5座的时候，动态规划的性能优势还没有体现出来。当金矿有10座，甚至更多的时候，动态规划就明显具备了优势。

原来如此，题目终于完美解决啦！



先别高兴的太早，你还需要再思考一个问题。

我们把题目改变一下，总工人数变成 1000 人，每个金矿的用工数也相应增加，这时候如何实现最优选择呢？



切，这还不是一样道理吗？继续用动态规划的方法，两层循环搞定问题！



哈哈，你想一想，1000工人5金矿的时候，总共需要计算多少次，开辟多少空间？



动态规划的时间复杂度是 $O(n \times w)$ ，空间复杂度是 $O(w)$ 。在 $n=5, w=1000$ 的时候，显然要计算 5000 次，开辟 1000 单位的空间。



如果我们用简单递归呢？需要计算多少次，开辟多少空间？



简单递归的时间复杂度是 $O(2^n)$ ，需要计算 32 次，开辟 5 单位（递归深度）的空间。



啊，我好像明白了... 1000 工人的时候，动态规划方法的性能反而不如简单递归呢。



是的，由于动态规划方法的时间和空间都和 W 成正比，而简单递归却和 W 无关，所以当工人数量很多的时候，动态规划反而不如递归。



所以说，每一种算法都没有绝对的好与坏，关键看应用场景。



好了，关于动态规划算法就介绍到这里。我所讲的这些，也仅仅是其中最最基础的部分。



要想更进一步提升，大家可以去研究更多更深的算法问题，例如多重背包算法、迪杰特斯拉算法，这些都是以动态规划为基础。



希望通过这篇文章，大家能在算法的理解方面更上一层楼。
感谢支持！



-----END-----

推荐阅读

- 漫画算法：最小栈的实现
- 漫画算法：判断 2 的乘方
- 漫画算法：找出缺失的整数
- 漫画算法：辗转相除法是什么鬼？

觉得本文有帮助？请分享给更多人

关注「算法爱好者」，修炼编程内功

算法爱好者

专注算法相关内容



微信号：AlgorithmFans



长按识别二维码关注

伯乐在线旗下微信公众号

商务合作QQ：2302462408