138 个回答 默认排序

阮行止
上海洛谷网络科技有限公司 讲师

专业 已有 5 人赞与了专业徽章

7,809 人赞同了该回答

0. intro

很有意思的问题。以往见过许多教材，对动态规划（DP）的引入属于“奉天承运，皇帝诏曰”式：不给出一点引入，见面即拿出一大堆公式吓人；学生则死啃书本，然后突然顿悟。针对入门者的教材不应该是这样的。恰好我给入门者讲过四次DP入门，迭代出了一套比较靠谱的教学方法，所以今天跑过来献丑。

现在，我们试着自己来一步步“重新发明”DP。

1. 从一个生活问题谈起

先来看看生活中经常遇到的事吧——假设您是个土豪，身上带了足够的1、5、10、20、50、100元面值的钞票。现在您的目标是凑出某个金额w，**需要用到尽量少的钞票。**

依据生活经验，我们显然可以采取这样的策略：能用100的就尽量用100的，否则尽量用50的……依次类推。在这种策略下，666=6×100+1×50+1×10+1×5+1×1，共使用了10张钞票。

这种策略称为“**贪心**”：假设我们面对的局面是“需要凑出w”，**贪心策略会尽快让w变得更小。**能让w少100就尽量让它少100，这样我们接下来面对的局面就是凑出w-100。长期的生活经验表明，贪心策略是正确的。

但是，如果我们换一组钞票的面值，贪心策略就也许不成立了。如果一个奇葩国家的钞票面额分别是1、5、11，那么我们在凑出15的时候，贪心策略会出错：

15=1×11+4×1（贪心策略使用了5张钞票）

15=3×5（正确的策略，只用3张钞票）

为什么会这样呢？贪心策略错在了哪里？

鼠目寸光。

刚刚已经说过，贪心策略的纲领是：“尽量使接下来面对的w更小”。这样，贪心策略在w=15的局面时，会优先使用11来把w降到4；但是在这个问题中，凑出4的代价是很高的，必须使用4×1。如果使用了5，w会降为10，虽然没有4那么小，但是凑出10只需要两张5元。

在这里我们发现，贪心是一种**只考虑眼前情况**的策略。

那么，现在我们怎样才能避免鼠目寸光呢？

如果直接暴力枚举凑出w的方案，明显复杂度过高。太多种方法可以凑出w了，枚举它们的时间是不可承受的。我们现在来尝试找一下性质。

重新分析刚刚的例子。w=15时，我们如果取11，接下来就面对w=4的情况；如果取5，则接下来面对w=10的情况。我们发现这些问题都有相同的形式：“给定w，凑出w所用的最少钞票是多少张？”接下来，我们用f(n)来表示“凑出n所需的最少钞票数量”。

那么，如果我们取了11，最后的代价（用掉的钞票总数）是多少呢？

明显 $\mathbf{cost} = f(4) + 1 = 4 + 1 = 5$ ，它的意义是：利用11来凑出15，付出的代价等于f(4)加上自己这一张钞票。现在我们暂时不管f(4)怎么求出来。

依次类推，马上可以知道：如果我们用5来凑出15，cost就是 $f(10) + 1 = 2 + 1 = 3$ 。

那么，现在w=15的时候，我们该取那种钞票呢？**当然是各种方案中，cost值最低的那一个**！

- 取11： $\mathbf{cost} = f(4) + 1 = 4 + 1 = 5$
- 取5： $\mathbf{cost} = f(10) + 1 = 2 + 1 = 3$
- 取1： $\mathbf{cost} = f(14) + 1 = 4 + 1 = 5$

显而易见，cost值最低的是取5的方案。**我们通过上面三个式子，做出了正确的决策**！

这给了我们一个**至关重要**的启示—— $f(n)$ 只与 $f(n-1), f(n-5), f(n-11)$ 相关；更确切地说：

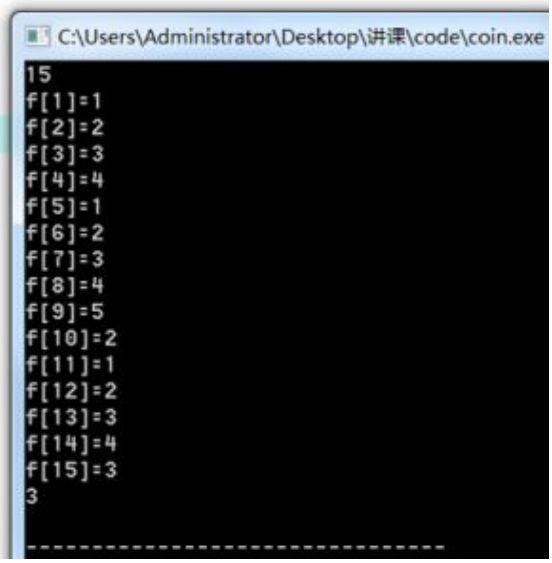
$$f(n) = \min\{f(n-1), f(n-5), f(n-11)\} + 1$$

这个式子是非常激动人心的。我们要求出f(n)，只需要求出几个更小的f值；既然如此，我们从小到大把所有的f(i)求出来不就好了？注意一下边界情况即可。代码如下：

```
int f[105],i,n,cost;
scanf("%d",&n);

f[0]=0;

for(i=1;i<=n;i++)
{
    cost=INF;
    if(i-1>=0) cost=min(cost,f[i-1]+1);
    if(i-5>=0) cost=min(cost,f[i-5]+1);
    if(i-11>=0) cost=min(cost,f[i-11]+1);
    f[i]=cost;
    printf("f[%d]=%d\n",i,f[i]);
}
```



我们以 $O(n)$ 的复杂度解决了这个问题。现在回过头来，我们看看它的原理：

- $f(n)$ 只与 $f(n-1), f(n-5), f(n-11)$ 的**值**相关。
- 我们只关心 $f(w)$ 的**值**，不关心是怎么凑出w的。

这两个事实，保证了我们做法的正确性。它比起贪心策略，会分别算出取1、5、11的代价，从而做出一个正确决策，这样就避免掉了“鼠目寸光”！

它与暴力的区别在哪里？我们的暴力枚举了“使用的硬币”，然而这属于冗余信息。我们要的是答案，根本不关心这个答案是怎么凑出来的。譬如，要求出f(15)，只需要知道f(14),f(10),f(4)的值。**其他信息并不需要。**我们舍弃了冗余信息。我们只记录了对解决问题有帮助的信息——f(n).

我们能这样干，取决于问题的性质：求出f(n)，只需要知道几个更小的f(c)。**我们将求解f(c)称作求解f(n)的“子问题”。**

这就是DP（动态规划，dynamic programming）.

将一个问题拆成几个子问题，分别求解这些子问题，即可推断出大问题的解。

> 思考题：请稍微修改代码，输出我们凑出w的**方案**。

2. 几个简单的概念

【无后效性】

一旦f(n)确定，“我们如何凑出f(n)”就再也用不着了。

要求出f(15)，只需要知道f(14),f(10),f(4)的值，而f(14),f(10),f(4)是如何算出来的，对之后的问题没有影响。

“未来与过去无关”，这就是**无后效性。**

（严格定义：如果给定某一阶段的状态，则在这一阶段以后过程的发展不受这阶段以前各段状态的影响。）

【最优子结构】

回顾我们对f(n)的定义：我们记“凑出n所需的**最少**钞票数量”为f(n).

f(n)的定义就已经蕴含了“最优”。利用w=14,10,4的**最优**解，我们即可算出w=15的**最优**解。

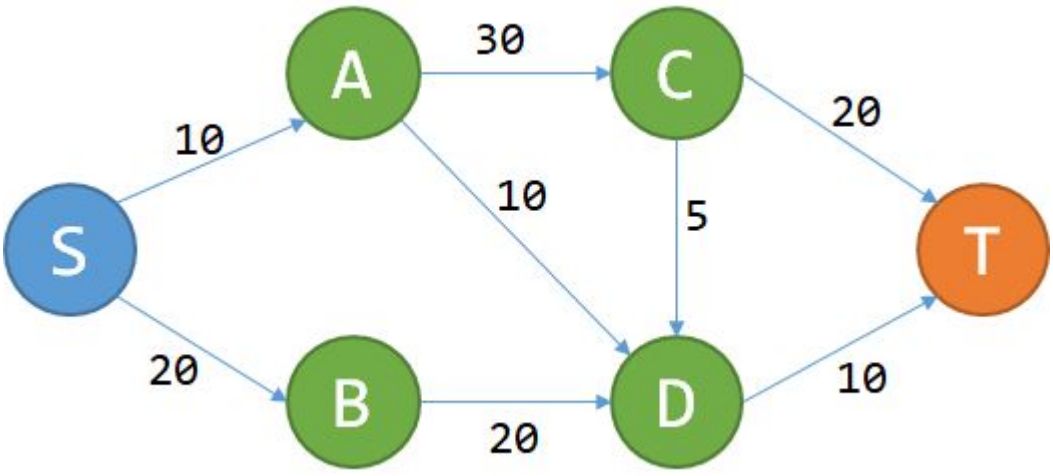
大问题的**最优解**可以由小问题的**最优解**推出，这个性质叫做“最优子结构性质”。

引入这两个概念之后，我们如何判断一个问题能否使用DP解决呢？

能将大问题拆成几个小问题，且满足无后效性、最优子结构性质。

3. DP的典型应用：DAG最短路

问题很简单：给定一个城市的地图，所有的道路都是单行道，而且不会构成环。每条道路都有过路费，问您从S点到T点花费的最少费用。



一张地图。边上的数字表示过路费。

这个问题能用DP解决吗？我们先试着记从S到P的最少费用为f(P).
想要到T，要么经过C，要么经过D。从而 $f(T) = \min\{f(C) + 20, f(D) + 10\}$.

好像看起来可以DP。现在我们检验刚刚那两个性质：
- 无后效性：对于点P，一旦f(P)确定，以后就只关心f(P)的值，不关心怎么去的。
- 最优子结构：对于P，我们当然只关心到P的最小费用，即f(P)。如果我们从S走到T是 $S \to P \to Q \to T$ ，那肯定S走到Q的最优路径是 $S \to P \to Q$ 。对一条最优的路径而言，从S走到**沿途上所有的点（子问题）**的最优路径，都是这条大路的一部分。这个问题的最优子结构性质是显然的。

既然这两个性质都满足，那么本题可以DP。式子明显为：

$$f(P) = \min\{f(R) + w_{R \to P}\}$$

其中R为有路通到P的所有的点， $w_{R \to P}$ 为R到P的过路费。

代码实现也很简单，拓扑排序即可。

4. 对DP原理的一点讨论

【DP的核心思想】

DP为什么会快？
无论是DP还是暴力，我们的算法都是在**可能解空间**内，寻找**最优解**。

来看钞票问题。暴力做法是枚举所有的可能解，这是最大的可能解空间。
DP是枚举**有希望成为答案的解**。这个空间比暴力的小得多。

也就是说：DP自带剪枝。

DP舍弃了一大堆不可能成为最优解的答案。譬如：
15 = 5+5+5 被考虑了。
15 = 5+5+1+1+1+1+1 从来没有考虑过，因为这不可能成为最优解。

从而我们可以得到DP的核心思想：**尽量缩小可能解空间。**

在暴力算法中，可能解空间往往是指数级的大小；如果我们采用DP，那么有可能把解空间的大小降到多项式级。

一般来说，解空间越小，寻找解就越快。这样就完成了优化。

【DP的操作过程】

一言以蔽之：**大事化小，小事化了。**

将一个大问题转化成几个小问题；
求解小问题；
推出大问题的解。

【如何设计DP算法】

下面介绍比较通用的设计DP算法的步骤。

首先，把我们面对的**局面**表示为x。这一步称为**设计状态。**
对于状态x，记我们要求出的答案(e.g. 最小费用)为f(x).我们的目标是求出f(T).
找出f(x)与哪些局面有关（记为p），写出一个式子（称为**状态转移方程**），通过f(p)来推出f(x).

【DP三连】

设计DP算法，往往可以遵循DP三连：

我是谁？ ——设计状态，表示局面
我从哪里来？
我要到哪里去？ ——设计转移

设计状态是DP的基础。接下来的设计转移，有两种方式：一种是考虑我从哪里来（本文之前提到的两个例子，都是在考虑“我从哪里来”）；另一种是考虑我到哪里去，这常见于求出f(x)之后，**更新能从x走到的一些解**。这种DP也是不少的，我们以后会遇到。

总而言之，“我从哪里来”和“我要到哪里去”只需要考虑清楚其中一个，就能设计出状态转移方程，从而写代码求解问题。前者又称pull型的转移，后者又称push型的转移。（这两个词是

@阮止雨 妹妹告诉我的，不知道源出处在哪）

> 思考题：如何把钞票问题的代码改写成“我到哪里去”的形式？
> 提示：求出f(x)之后，更新f(x+1),f(x+5),f(x+11).

5. 例题：最长上升子序列

扯了这么多形而上的内容，还是做一道例题吧。

最长上升子序列（LIS）问题：给定长度为n的序列a，从a中抽取出一个子序列，这个子序列需要单调递增。问最长的上升子序列（LIS）的长度。
e.g. 1,5,3,4,6,9,7,8的LIS为1,3,4,6,7,8，长度为6。

如何设计状态（我是谁）？

我们记 $f(x)$ 为以 a_x 结尾的LIS长度，那么答案就是 $\max\{f(x)\}$.

状态x从哪里推过来（我从哪里来）？

考虑比x小的每一个p：如果 $a_x > a_p$ ，那么f(x)可以取f(p)+1.

解释：我们把 a_x 接在 a_p 的后面，肯定能构造一个以 a_x 结尾的上升子序列，长度比以 a_p 结尾的LIS大1.那么，我们可以写出状态转移方程了：

$$f(x) = \max_{p<x, a_p<a_x} \{f(p)\} + 1$$

至此解决问题。两层for循环，复杂度 $O(n^2)$.

```
int main(void)
{
    int f[105]={0},a[105]={0},i,x,p,n,ans=0;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
        scanf("%d",&a[i]),f[i]=1;

    for(x=1;x<=n;x++)
    {
        for(p=1;p<x;p++)
            if(a[p]<a[x]) f[x]=max(f[x],f[p]+1);
        printf("f[%d]=%d\n",x,f[x]);
    }

    for(x=1;x<=n;x++)
        ans=max(ans,f[x]);

    printf("%d\n",ans);
```

```
C:\Users\Administrator\Desktop\讲课\code\lcs.
8
1 5 3 4 6 9 7 8
f[1]=1
f[2]=2
f[3]=2
f[4]=3
f[5]=4
f[6]=5
f[7]=5
f[8]=6
6

--------------------------------
Process exited after 4.065 seconds w
请按任意键继续. . .
```

从这三个例题中可以看出，DP是一种思想，一种“大事化小，小事化了”的思想。带着这种思想，DP将会成为我们解决问题的利器。

最后，我们一起念一遍DP三连吧——我是谁？我从哪里来？我要到哪里去？

6. 习题

如果读者有兴趣，可以试着完成下面几个习题：

一、请采取一些优化手段，以 $O(n \log n)$ 的复杂度解决LIS问题。

提示：可以参考这篇博客 Junior Dynamic Programming--动态规划初步·各种子序列问题

二、“按顺序递推”和“记忆化搜索”是实现DP的两种方式。请查阅资料，简单描述“记忆化搜索”是什么。并采用记忆化搜索写出钞票问题的代码，然后完成P1541 乌龟棋 - 洛谷 。

三、01背包问题是一种常见的DP模型。请完成P1048 采药 - 洛谷。

谢谢您看完本文 /(/// ω ///)/

2019.3.3

编辑于 2019-03-03

王勐

4,297 人赞同了该回答

动态规划的本质不在于是递推或是递归，也不需要纠结是不是内存换时间。

理解动态规划并不需要数学公式介入，只是完全解释清楚需要点篇幅...首先需要明白哪些问题不是动态规划可以解决的，才能明白为神马需要动态规划。不过好处时顺便也就搞明白了递推贪心搜索和动规之间有什么关系，以及帮助那些总是把动规当成搜索解的同学建立动规的思路。当然熟悉了之后可以直接根据问题的描述得到思路，如果有需要的话再补充吧。

动态规划是对于 某一类问题 的解决方法！！重点在于如何鉴定“某一类问题”是动态规划可解的而不是纠结解决方法是递归还是递推！

怎么鉴定dp可解的一类问题需要从计算机是怎么工作的说起...计算机的本质是一个状态机，内存里存储的所有数据构成了当前的状态，CPU只能利用当前的状态计算出下一个状态（不要纠结硬盘之类的外部存储，就算考虑他们也只是扩大了状态的存储容量而已，并不能改变下一个状态只能从当前状态计算出来这一条铁律）

当你企图使用计算机解决一个问题是，其实就是在思考如何将这个问题表达成状态（用哪些变量存储哪些数据）以及如何在状态中转移（怎样根据一些变量计算出另一些变量）。所以所谓的空间复杂度就是为了支持你的计算所必需存储的状态最多有多少，所谓时间复杂度就是从初始状态到达最终状态中间需要多少步！

太抽象了还是举个例子吧：

比如说我想计算第100个非波那契数，每一个非波那契数就是这个问题的一个状态，每求一个新数字只需要之前的两个状态。所以同一个时刻，最多只需要保存两个状态，空间复杂度就是常数；每计算一个新状态所需要的时间也是常数且状态是线性递增的，所以时间复杂度也是线性的。

上面这种状态计算很直接，只需要依照固定的模式从旧状态计算出新状态就行（a[i]=a[i-1]+a[i-2]），不需要考虑是不是需要更多的状态，也不需要选择哪些旧状态来计算新状态。对于这样的解法，我们叫递推。

非波那契那个例子过于简单，以至于让人忽视了阶段的概念，所谓阶段是指随着问题的解决，在同一个时刻可能会得到的不同状态的集合。非波那契数列中，每一步会计算得到一个新数字，所以每个阶段只有一个状态。想象另外一个问题情景，假如把你放在一个围棋棋盘上的某一点，你每一步只能走一格，因为你可以东南西北随便走，所以你当你同样走四步可能会处于很多个不同的位置。从头开始走了几步就是第几个阶段，走了n步可能处于的位置称为一个状态，走了这n步所有可能到达的位置的集合就是这个阶段下所有可能的状态。

现在问题来了，有了阶段之后，计算新状态可能会遇到各种奇葩的情况，针对不同的情况，就需要不同的算法，下面就分情况来说明一下：

假如问题有n个阶段，每个阶段都有多个状态，不同阶段的状态数不必相同，一个阶段的一个状态可以得到下个阶段的所有状态中的几个。那我们要计算出最终阶段的状态数自然要经历之前每个阶段的某些状态。

好消息是，有时候我们并不需要真的计算所有状态，比如这样一个弱智的棋盘问题：从棋盘的左上角到达右下角最短需要几步。答案很显然，用这样一个弱智的问题是为了帮助我们理解阶段和状态。某个阶段确实可以有多个状态，正如这个问题中走n步可以走到很多位置一样。但是同样n步中，有哪些位置可以让我们在第n+1步中走的最远呢？没错，正是第n步中走的最远的位置。换成一句熟悉话叫做“下一步最优是从当前最优得到的”。所以为了计算最终的最优值，只需要存储每一步的最优值即可，解决符合这种性质的问题的算法就叫贪心。如果只看最优状态之间的计算过程是不是和非波那契数列的计算很像？所以计算的方法是递推。

既然问题都是可以划分成阶段和状态的。这样一来我们一下子解决了一大类问题：一个阶段的最优可以由前一个阶段的最优得到。

如果一个阶段的最优无法用前一个阶段的最优得到呢？

什么你说只需要之前两个阶段就可以得到当前最优？那跟只用之前一个阶段并没有本质区别。最麻烦的情况在于你需要之前所有的情况才行。

再来一个迷宫的例子。在计算从起点到终点的最短路线时，你不能只保存当前阶段的状态，因为题目要求你最短，所以你必须知道之前走过的所有位置。因为即便你当前再的位置不变，之前的路线不同会影响你的之后走的路线。这时你需要保存的是之前每个阶段所经历的那个状态，根据这些信息才能计算出下一个状态！

每个阶段的状态或许不多，但是每个状态都可以转移到下一阶段的多个状态，所以解的复杂度就是指数的，因此时间复杂度也是指数的。哦哦，刚刚提到的之前的路线会影响到下一步的选择，这个令人不开心的情况就叫做有后效性。

刚刚的情况实在太普遍，解决方法实在太暴力，有没有哪些情况可以避免如此的暴力呢？

契机就在于后效性。

有一类问题，看似需要之前所有的状态，其实不用。不妨也是拿最长上升子序列的例子来说明为什么他不必需要暴力搜索，进而引出动态规划的思路。

假装我们年幼无知想用搜索去寻找最长上升子序列。怎么搜索呢？需要从头到尾依次枚举是否选择当前的数字，每选定一个数字就要去看看是不是满足“上升”的性质，这里第i个阶段就是去思考是否要选择第i个数，第i个阶段有两个状态，分别是选和不选。哈哈，依稀出现了刚刚迷宫找路的影子！咦慢着，每次当我决定要选择当前数字的时候，只需要和之前选定的一个数字比较就行了！这是和之前迷宫问题的本质不同！这就可以纵容我们不需要记录之前所有的状态啊！既然我们的选择已经不受之前状态的组合的影响了，那时间复杂度自然也不是指数的了啊！虽然我们不在乎某序列之前都是什么元素，但我们还是需要这个序列的长度的。所以我们只需要记录以某个元素结尾的LIS长度就好！因此第i个阶段的最优解只是由前i-1个阶段的最优解得到的，然后就得到了DP方程（感谢

@韩曦 指正）

$$LIS(i) = max\{LIS(j) + 1\} \quad j < i \ and \ a[j] < a[i]$$

所以一个问题是该用递推、贪心、搜索还是动态规划，完全是由这个问题本身阶段间状态的转移方式决定的！

每个阶段只有一个状态->递推；
每个阶段的最优状态都是由上一个阶段的最优状态得到的->贪心；
每个阶段的最优状态是由之前所有阶段的状态的组合得到的->搜索；
每个阶段的最优状态可以从之前某个阶段的某个或某些状态直接得到而不管之前这个状态是如何得到的->动态规划。

> 每个阶段的最优状态可以从之前某个阶段的某个或某些状态直接得到

这个性质叫做最优子结构；

> 而不管之前这个状态是如何得到的

这个性质叫做无后效性。
另：其实动态规划中的最优状态的说法容易产生误导，以为只需要计算最优状态就好，LIS问题确实如此，转移时只用到了每个阶段“选”的状态。但实际上有的问题往往需要对每个阶段的所有状态都算出一个最优值，然后根据这些最优值再来找最优状态。比如背包问题就需要对前i个包（阶段）容量为j时（状态）计算出最大价值。然后在最后一个阶段中的所有状态种找到最优值。

编辑于 2015-03-26

覃含章
机器学习 话题的优秀回答者

编辑推荐 共 2 项收录

872 人赞同了该回答

看了看高赞回答们，大多还是出于直觉出发。题主主要问了两个问题，关于第一个问题，什么是动态规划，我这里简单讲一个**关于动态规划的抽象模型**（本回答的第一~第四节），其实对一定数学基础的同学来说也会很简明扼要。

这个思路来源于搞了动态规划几十年（也算是理论奠基人之一了）的Dimitri Bertsekas，他所称之为Abstract Dynamic Programming Models。在回答最后的第五节，我再基于个人的观点谈谈学习研究动态规划的意义。

一、定义：两个集合 S, A ;策略映射 μ ；两个算子 T_μ, T

本节我们定义为了准确描述动态规划模型必不可少的5个符号。

考虑 S 和 A 为两个集合，前者我们认为是包含所有状态（state）的集合，后者我们认为是包含所有控制（control，或者action）的集合。对任意 $s \in S$ ，定义 $A(s) \subset A$ 为针对状态 s 的可行控制集。然后我们再对任意 $s \in S$ 定义函数映射 $\mu : S \to A, \mu(s) \in A(s)$. 如果你对动态规划一无所知，也无妨，那么你只需要把 S, A 当成两个任意的集合就可以。

那么我们现在定义了两组集合和一组函数。我们记无穷序列 $\pi = \{\mu_0, \mu_1, \ldots\}$ （满足任意 $\mu_k \in \mathcal{M}$ ）为非稳态的策略（nonstationary policies），因为如果我们把下标 k 代表不同的时间段，相当于每个时间段都在用不同的策略在应对。与之相对的，我们叫这样的序列 $\{\mu, \mu, \ldots\}$ 为稳态策略（stationary policies），因为每个时间段都用一样的策略（映射）。

坚持住，接下来就是定义最后最关键的符号了！我们把函数 $V: S \to \mathbb{R}$ 的集合叫做 $\mathcal{R}(S)$ ，并且定义一个映射 $U: S \times A \times \mathcal{R}(S) \to \mathbb{R}$ 。（如果你已经熟悉动态规划，这里其实就是所谓的值函数。然后比如强化学习里面所谓Q learning就是对这里的 U 映射进行近似的一种算法。但对没有这些知识的同学，同样的，你只要当成 U, V 是两个在对应空间上的定义的映射就好）

在定义了这些符号后，抽象动态规划便主要是考虑如下两组递推映射 T_μ, T ：

$$(T_\mu V)(s) = U(s, \mu(s), V),\ \forall s \in S, V \in \mathcal{R}(S)$$

$$(TV)(s) = \inf_{a \in A(s)} U(s, a, V), \forall\ s \in S, V \in \mathcal{R}(S)$$

也就是说， T_μ 是将函数 V 结合 μ 赋值给 U 的算子， T 是将函数 V 赋值给 U ，并对 U 在控制集合上最优化的算子。

二、抽象动态规划模型

一般来说，一个动态规划问题会有 N 个“阶段”，那么对一个最终阶段成本 $\bar{V} \in \mathcal{R}(S)$ 来说，在使用策略 $\pi = \{\mu_0, \mu_1, \ldots\}$ 的情况下，我们从不同状态 x 起始的N阶段总成本就可以写成：(算子的好处就体现出来了！)

$$V_{\pi,N}(s) = (T_{\mu_0} T_{\mu_1} \ldots T_{\mu_{N-1}} \bar{V})(s), \forall\ s \in S.$$

对不熟悉上边算子"连乘"记号的，实际上这个N个算子“连乘”可以分解成

$$T_{\mu_0} T_{\mu_1} \ldots T_{\mu_{N-1}} \bar{V} = (T_{\mu_0} (T_{\mu_1} (\ldots (T_{\mu_{N-2}} (T_{\mu_{N-1}} \bar{V}))) \ldots)).$$

也就是说，实际上我们是先对 $\bar{V}$ 作用算子 $T_{\mu_{N-1}}$ ，然后再对得到的 $T_{\mu_{N-1}} \bar{V}$ 作用算子 $T_{\mu_{N-2}}$ ，以此类推，最后作用算子 T_{μ_0} 。这也就是动态规划模型所谓的逆向归纳（backward induction）式。在每一步归纳（induction）中，实际上我们就是把后一回合的值函数传递到前一回合（根据当前回合使用的策略）。

当 N 很大的时候，也即所谓的无限时间动态规划模型，我们可以考虑一个更为简洁的问题结构。即我们本来主要考虑的 N 阶段总成本可以和 N 无关，记作

$$V_\pi(s) = \limsup_{N \to \infty} V_{\pi,N}(x) = \limsup_{N \to \infty} (T_{\mu_0} T_{\mu_1} \ldots T_{\mu_{N-1}} \bar{V})(s), \forall\ s \in S.$$

即为原来成本函数的极限。动态规划模型的目标便是要找到一组最好的策略 π^* ，最小化这个总成本（如果最小值存在）。也即我们要找到“最小可能的”成本

$$V^*(s) = \inf_\pi V_\pi(s).$$

在合理的条件下，这个问题等价于一个求不动点问题： V^* 应当满足

$$V^*(s) = \inf_{a \in A(s)} U(s, a, V^*), \ \forall \, s \in S.$$

即问题变成了求T的不动点（希望你已经熟悉 T 的定义了:)）！ 且如果 π^* 确实存在，一般来说我们在无限时间动态规划模型里只需要考虑稳态策略，因为一般来说存在一个最优的 π^* 是稳态的。那么这种情况下我们就得到问题求解等价于所谓的Bellman's equation：（找到对应的 V^* ）

$$(T_{\mu^*} V^*)(s) = (TV^*)(s), \ \forall \, s \in S.$$

也就是说，抽象的来看，动态规划需要解决的问题就是计算由5个符号定义的，这个等式的解！

三、两个重要性质：单调性与压缩性

本节我们介绍常见动态规划模型需要满足的两个重要性质，也即前一段末尾所说的“合理的条件”。只有满足了这两个条件，我们的动态规划模型才可以化为上述唯一确定的基于Bellman's equation的不动点求解问题。

先说**单调性**（Monotonicity）：

$$若 V, V' \in \mathcal{R}(S) 且 V \leq V', 那么 U(s, a, V) \leq U(s, a, V'), \ \forall \, s \in S : a \in A(s).$$

也就是说算子U 对于取值 V 是单调的。这也很好理解，这个意思就是说当我们在时间 k 的时候，如果从未来时间 $k+1$ 到无穷的成本 V 对于所有状态 s 增加（单调上升）了，那么我们在时间 k ，对相同的状态 s 和控制 a 的总成本也应当上升。因此，不难想象，任何一个“合理”的动态规划模型都会满足这里的单调性。

再说**压缩性**（Contraction）：

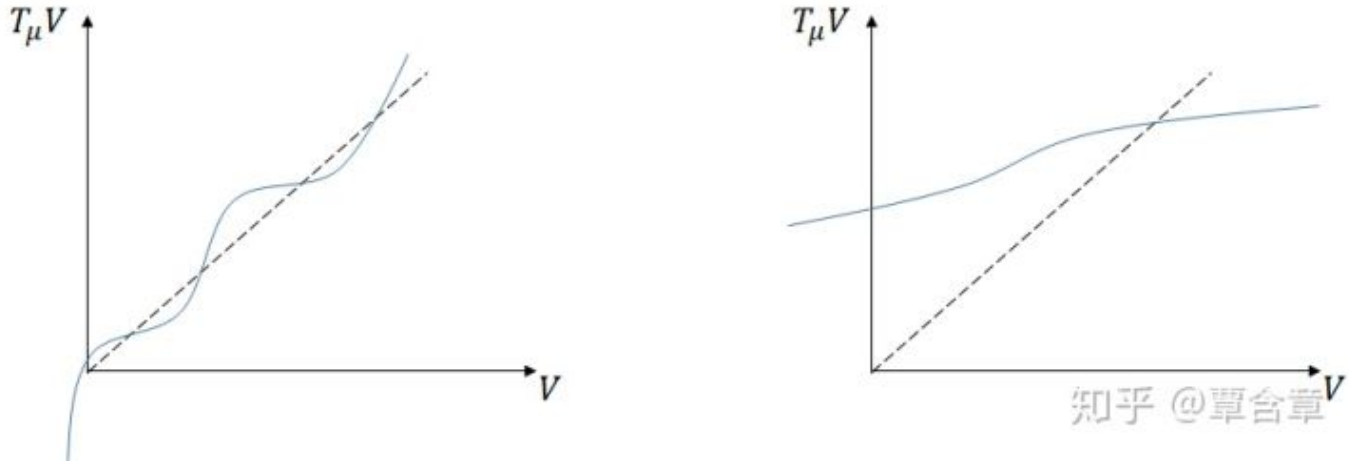
假设 $\mathcal{R}(S)$ 是赋范空间，则对任意 $V \in \mathcal{R}(X), \mu \in \mathcal{M}$ ，函数 $T_\mu V, TV \in \mathcal{R}(X).$ 且对某个 $\alpha \in (0, 1)$ 有

$$\|T_\mu V - T_\mu V'\| \leq \alpha \|V - V'\|, \ \forall \, V, V' \in \mathcal{R}(X), \mu \in \mathcal{M}.$$

也就是说 T_μ, T 是压缩映射(contraction mapping)！注意这里的范数一般来说取weighted sup-norm，不过本回答不再展开（包括赋范空间的严格定义云云）。

不过这个压缩性的话可能就不如前面那个单调性，没什么特别直观的道理。然而如果你认同“折旧”（discount）的会计概念，比如未来的单位成本相比今天的单位成本要按照日期以固定比例 α 打个折扣 --- 放到我们的模型来说就是在从时间 k 到 $k+1$ 的过程当中我们产生的额外成本都要乘以一个属于0,1之间的系数 α ，那么容易验证在这种情况下我们的模型一般就会满足这里的压缩性了。

好那么有没有什么直观的解释为什么我们的动态规划模型最好要满足这两个性质呢？



回忆：Bellman's equation的本质便是寻找 T 的不动点。而上图左，我们的 T_μ 虽然单调但并不具有压缩性，因此实际上存在不止一个不动点（蓝线与黑虚线的交点）。相比之下，上图右的 T_μ 既单调又有压缩性，我们便很容易知道这个时候Bellman's equation有唯一的不动点。因此，可以认为问题的性质就要好很多了。

四、马尔可夫决策过程（Markov Decision Process）

上面的讨论其实完全没有涉及具有实际意义的问题，因此如果你觉得太抽象了，本节便讨论如何将MDP用抽象动态规划的框架写出来。那么因为是MDP，我们有所谓的离散时间状态转移方程：

$$s_{k+1} = f(s_k, a_k, w_k),\ k = 0, 1, \dots,$$

也就是说，如果时间k的时候我们处于状态 s_k ，并采取了控制 a_k ，那么 f 就决定了我们下一个时间点 $k+1$ 的状态 s_{k+1} 。当然，这里还有一点噪音/随机性 w_k ，满足分布 $\mathbb{P}[\cdot|s_k, a_k]$ ，所以也跟当前状态和控制有关。那么我们的MDP问题就可以看成是要对任意的初始状态 s_0 ，找到策略 π 最小化总的期望成本 $V_\pi(s_0)$ ：

$$\inf_\pi V_\pi(s_0) = \inf_\pi \left(\limsup_{N\to\infty} \mathbb{E}_{\substack{w_k; \\ k=0,1,\dots}} \left[\sum_{k=0}^{N-1} \alpha^k g(x_k, \mu_k(s_k), w_k) \right] \right).$$

注意这里我们使用了折旧系数 α ，且 g 就是每个时间段 k 的成本，最后的总成本就是所有成本的总和。利用我们抽象动态规划的框架，实际上这个问题等价于定义

$$U(s, a, V) = \mathbb{E}_w[g(s, a, w) + \alpha V(f(s, a, w))].$$

也就是说我们考虑的两个算子定义为

$$(T_\mu V)(s) = \mathbb{E}_w[g(s, \mu(s), w) + \alpha V(f(s, \mu(s), w))].$$

$$(TV)(s) = \inf_{a \in A(s)} \mathbb{E}_w[g(s,a,w) + \alpha V(f(s,a,w))].$$

如果你已经接触过动态规划，这可能就会是你更加熟悉的递推方程。

容易验证， T_μ, T 是单调的，且如果 g 有界（对任意 $s \in S : a \in U(s), \mathbb{E}_w[|g(s,a,w)|] < \infty$ ），T_μ, T 是压缩映射。即，前一节的两个性质的确是对普遍意义上的动态规划问题都成立的。

五、动态规划的意义

动态规划是非常强大的建模工具。基本上，一个多阶段的决策问题，如果你无法把他至少写成一个动态规划模型，那很可能你也无法找到办法求解它。

比如在运营管理界，所谓的动态库存控制问题就是一个十分经典的动态规划问题。而在物流运输界，作为核心问题之一的最短路径问题也是一个经典的动态规划问题。事实上，任何一个有限状态空间的MDP问题都可以写成最短路问题，反之亦然（不过本回答不再讨论这些抽象概念之间的联系了，打住打住）。

作为强大建模能力的代价便是，往往你很容易写出各种眼花缭乱的动态规划递推，但是可能花了很久也找不到一个“高效”的算法求解之。这是因为在我看来，动态规划是一种数学规划的建模思想，可以对各种各样的多阶段决策问题进行建模，但本身却只蕴含了一个和暴力枚举差不多的基本算法。尤其在决策空间维度很大时，动态规划算法会遭受著名的维数诅咒（curse of dimensionality），即算法求解时间随着问题规模指数级增长。

因此为了真正求解复杂的动态规划问题，我们实际上只能近似求解，这也是所谓的近似动态规划。这几年，伴随着机器学习的热潮，数据驱动的近似动态规划也渐渐再度被人熟悉，尤其是其中的一类近似算法，所谓的强化学习（reinforcement learning）算法。当然，一些CS背景的同学，可能会觉得RL和动态规划已经不是一类问题了，这里我们也不多展开了。

总之，作为研究领域动态规划已经存在了超过半个世纪，在目前的当下又受到了一波关注，而它的研究难点就在于作为泛用的建模工具，针对高维问题的算法设计和求解。

参考文献：

Bertsekas, Dimitri P. Abstract dynamic programming. Belmont, MA: Athena Scientific, 2013.

编辑于 2019-07-24

Abstract DP

赞赏

还没有人赞赏，快来当第一个赞赏的人吧！

徐凯强 Andy
永远好奇

3,335 人赞同了该回答

动态规划中递推式的求解方法不是动态规划的本质。

我曾经作为省队成员参加过NOI，保送之后也给学校参加NOIP的同学多次讲过动态规划，我试着讲一下我理解的**动态规划**，争取深入浅出。希望你看了我的答案，能够喜欢上动态规划。

0. 动态规划的本质，是对问题***状态的定义***和***状态转移方程***的**定义**。
引自维基百科

> **dynamic programming** is a method for solving a complex problem by **breaking it down into a collection of simpler subproblems**.

动态规划是通过**拆分问题，**定义问题状态和状态之间的关系，使得问题能够以递推（或者说分治）的方式去解决。
本题下的其他答案，大多都是在说递推的求解方法，但**如何拆分问题**，才是动态规划的核心。
而**拆分问题**，靠的就是***状态的定义***和***状态转移方程***的**定义**。

1. 什么是**状态的定义？**

首先想说大家千万不要被下面的数学式吓到，这里只涉及到了函数相关的知识。
我们先来看一个动态规划的教学必备题：

> 给定一个数列，长度为N，
> 求这个数列的最长上升（递增）子数列（LIS）的长度.
> 以
> 1 7 2 8 3 4
> 为例。
> 这个数列的最长递增子数列是 1 2 3 4，长度为4；
> 次长的长度为3， 包括 1 7 8; 1 2 3 等.

要解决这个问题，我们首先要**定义这个问题**和这个问题的子问题。
有人可能会问了，题目都已经在这了，我们还需定义这个问题吗？需要，原因就是这个问题在字面上看，找不出子问题，而没有子问题，这个题目就没办法解决。

所以我们来重新定义这个问题：

> 给定一个数列，长度为N，
> 设 F_k 为：以数列中第k项结尾的最长递增子序列的长度.
> 求 $F_1..F_N$ 中的最大值.

显然，这个新问题与原问题等价。
而对于 F_k 来讲， $F_1..F_{k-1}$ 都是 F_k 的子问题：因为以第k项结尾的最长递增子序列（下称LIS），包含着以第 $1..k-1$ 中某项结尾的LIS。

上述的新问题 F_k 也可以叫做状态，定义中的“ F_k 为数列中第k项结尾的LIS的长度”，就叫做对状态的定义。
之所以把 F_k 做“状态”而不是“问题”，一是因为避免跟原问题中“问题”混淆，二是因为这个新问题是数学化定义的。

对状态的定义只有一种吗？*当然不是*。
我们甚至可以二维的，以完全不同的视角定义这个问题：

> 给定一个数列，长度为N，
> 设 $F_{i,k}$ 为：
> 在前i项中的，长度为k的最长递增子序列中，最后一位的最小值. $1 \le k \le N$.
> 若在前i项中，不存在长度为k的最长递增子序列，则 $F_{i,k}$ 为正无穷.
> 求最大的x，使得 $F_{N,x}$ 不为正无穷。

这个新定义与原问题的等价性也不难证明，请读者体会一下。
上述的 $F_{i,k}$ 就是状态，定义中的“ $F_{i,k}$ 为：在前i项中，长度为k的最长递增子序列中，最后一位的最小值”就是对状态的定义。

2. 什么是**状态转移方程**？
上述状态定义好之后，状态和状态之间的关系式，就叫做**状态转移方程。**

比如，对于LIS问题，我们的第一种定义：

> 设 F_k 为：以数列中第k项结尾的最长递增子序列的长度.

设A为题中数列，状态转移方程为：

> $F_1 = 1$ （根据状态定义导出边界情况）
> $F_k = max(F_i + 1 | A_k > A_i, i \in (1..k-1))\ (k > 1)$

用文字解释一下是：
以第k项结尾的LIS的长度是：保证第i项比第k项小的情况下，以第i项结尾的LIS长度加一的最大值，取遍i的所有值（i小于k）。

第二种定义：

> 设 $F_{i,k}$ 为：在数列前i项中，长度为k的递增子序列中，最后一位的最小值

设A为题中数列，状态转移方程为：

> 若 $A_i > F_{i-1,k-1}$ 则 $F_{i,k} = min(A_i, F_{i-1,k})$
> 否则： $F_{i,k} = F_{i-1,k}$

（边界情况需要分类讨论较多，在此不列出，需要根据状态定义导出边界情况。）
大家套着定义读一下公式就可以了，应该不难理解，就是有点绕。

这里可以看出，这里的状态转移方程，就是定义了问题和子问题之间的关系。
可以看出，状态转移方程就是带有条件的递推式。

3. 动态规划迷思
本题下其他用户的回答跟动态规划都有或多或少的联系，我也讲一下与本答案的联系。

a. “缓存”，“重叠子问题”，“记忆化”：
这三个名词，都是在阐述递推式求解的技巧。以Fibonacci数列为例，计算第100项的时候，需要计算第99项和98项；在计算第101项的时候，需要第100项和第99项，这时候你还需要重新计算第99项吗？不需要，你只需要在第一次计算的时候把它记下来就可以了。
上述的需要再次计算的“第99项”，就叫“重叠子问题”。如果没有计算过，就按照递推式计算，如果计算过，直接使用，就像“缓存”一样，这种方法，叫做“记忆化”，这是递推式求解的技巧。这种技巧，通俗的说叫“花费空间来节省时间”。**都不是动态规划的本质，不是动态规划的核心。**

b. “递归”：
递归是递推式求解的方法，连技巧都算不上。

c. "无后效性"，“最优子结构”：
上述的状态转移方程中，等式右边不会用到下标大于左边i或者k的值，这是"无后效性"的通俗上的数学定义，符合这种定义的状态定义，我们可以说它具有“最优子结构”的性质，在动态规划中我们要做的，就是找到这种“最优子结构”。

在对状态和状态转移方程的定义过程中，满足“最优子结构”是一个隐含的条件（否则根本定义不出来）。对状态和“最优子结构”的关系的进一步解释，什么是动态规划？动态规划的意义是什么？ - 王勐的回答 写的很好，大家可以去读一下。

需要注意的是，一个问题可能有多种不同的状态定义和状态转移方程定义，存在一个有后效性的定义，***不代表该问题不适用动态规划***。这也是其他几个答案中出现的逻辑误区：
动态规划方法要寻找符合“最优子结构“的状态和状态转移方程的定义，在找到之后，这个问题就可以以“记忆化地求解递推式”的方法来解决。而寻找到的定义，才是动态规划的本质。

有位答主说：

> 分治在求解每个子问题的时候，都要进行一遍计算
> 动态规划则存储了子问题的结果，查表时间为常数

这就像说多加辣椒的菜就叫川菜，多加酱油的菜就叫鲁菜一样，是存在误解的。
文艺的说，动态规划是寻找一种对问题的观察角度，让问题能够以递推（或者说分治）的方式去解决。寻找看问题的角度，才是动态规划中最耀眼的宝石！（大雾）

编辑于 2014-12-19

Coldwings
Python 话题的优秀回答者

160 人赞同了该回答

首先明确：动态规划是用来求解最优化问题的一种方法。常规算法书上强调的是无后效性和最优子结构描述，这套理论是正确的，但是适用与否与你的状态表述有关。至于划分阶段什么的就有些扯淡了：动态规划不一定有所谓的阶段。其实质是状态空间的状态转移。

下面的理解为我个人十年竞赛之总结。基本上在oi和acm中我没有因为动态规划而失手过。

所有的决策类求最优解的问题都是在状态空间内找一个可以到达的最佳状态。搜索的方式是去遍历每一个点，而动态规划则是把状态空间变形，由此变成从初始到目标状态的最短路问题。

依照这种描述：假若你的问题的结论包含若干决策，则可以认为从初始状态（边界条件）到解中间的决策流程是一个决策状态空间中的转移路线。前提是：你的状态描述可以完整且唯一地覆盖所有有效的状态空间中的点，且转移路线包含所有可能的路径。

这个描述是包含动态规划两大条件的。所谓无后效性，指状态间的转移与如何到达某状态无关。如果有关，意味着你的状态描述不能完整而唯一地包括每一个状态。如果你发现一个状态转移有后效性，很简

单，把会引起后效性的参数作为状态描述的一部分放进去将其区分开来就可以了；最优子结构说明转移路线包含了所有可能的路径，如果不具备最优子结构，意味着有部分情况没有在转移中充分体现，增加转移的描述就可以了。最终所有的搜索问题都可以描述成状态空间内的状态转移方程，只是有可能状态数量是指数阶的，有可能不满足计算要求罢了。

这样的描述下，所有的动态规划问题都可以转变为状态空间内大量可行状态点和有效转移构成的图的从初始状态到最终状态的最短路问题。
于是乎，对于动态规划，他的本质就是图论中的最短路；阶段可以去除，因为不一定有明确的阶段划分。

发布于 2015-04-07

winter
程序员，前ACMer

52 人赞同了该回答

动态规划有很多等价的描述，我相信最优子结构和无后效什么的大家都听吐了。

我理解题主想要的是一个感性的认识，算法里面讲到动态规划的时候确实特别懵逼，觉得这玩意没头没脑的。

实际上动态规划这个东西是来自数学，它是运筹学的分支数学规划里面的一种方法。

运筹学是数学里很实用的部分，主要为管理学和经济学服务，数学规划，也叫规划论，最早就是为了交通运输和生产管理服务的。

数学规划研究的问题是最优化问题，最优化问题是在一定约束条件下找到一个最优解。

举个例子你去麦当劳吃饭，有各种套餐，怎么才能省钱又吃饱呢？这就是数学规划研究的东西。省钱就是你最优解表达式，吃饱就是约束方程，建个模型给麦当劳各种食品设定一个饱腹度，就可以求解了。当然你可以说不能光喝可乐灌个水饱啊，咱还得有个搭配，那模型就复杂了但是无非是最优解表达式和约束方程的变化。

根据这个最优解表达式和约束方程是线性还是非线性，规划问题分成线性规划和非线性规划，各自有不少数学方法。

动态规划是在这之上的一种方法，动态两个字体现在它有一定的时序性，有些问题天然跟时间有关系，有些问题可以通过建模分成“阶段”，比如咱们的0-1背包问题，你东西得一件一件往里放吧？这就是一种时序性。

有了时序性就有了子问题，如果你建模建得好，前置的子问题跟后续子问题就有一种逻辑关系或者数学关系，比如最大子段和问题，就跟子问题的含尾最大子段和跟不含尾最大子段和两者强相关，建模建错了，子问题最优不最优就毫无意义，那就不能动态规划了。

我觉得计算机算法里面的动态规划，用的初等数学方法比较多，如果想要再进一步，可以看看运筹学。

编辑于 2019-06-22

八汰
祝所有的人获得自洽的归途

44 人赞同了该回答

@熊大大 的回答真好，理解的准确又深刻。
我来试试看看能能不能能多创造一点价值 ：）

1. 动态规划是解决问题的一种方法。
是的，它只是问题的一种解决方法。一个问题完全可能有多种解法。

2. 动态规划的本质是它试图将问题定义成这样的形式：
“原问题的解如何由子问题的解组合而成。”

比如求解斐波那契数列的第n个数f(n)，用动态规划解决这个问题就是要找出f(n)实际上是等于f(n-1) 加上 f(n-2)得到的。
当然这是最简单的例子，很多问题在描述的时候并不能很显然的看出原问题是否有子问题，原问题的解如何由子问题的解组合而成，这时试图用动态规划求解就需要变化看待原问题的角度，来获得动态规划的问题形式。

如果怎么都无法将问题定义成这样的形式，那么很抱歉说明这个问题无法用动态规划求解。
我们常常说的状态转移方程，其实就是“原问题的解如何由子问题的解组合而成”这句话的数学表达。
所以

@熊大大 说“动态规划的本质，是对问题***状态的定义***和***状态转移方程***的定义。”，准确！（不过没学过的同学可能紧接着要问这里的状态指什么）

3. 对于有经验的人来说，摸索出原问题的解如何由子问题组合而成，就能把状态转移方程写出来，该问题就已经解决了。
因为动态规划的实现有很成熟的两类方法：
（1）“自顶向下”（top-down dynamic programming:）的实现方式
（2）“自低向上”（bottom-up dynamic programming：）的实现方式

有经验的人由状态转移方程随便选一种实现方式，就跟大部分高中试题一样遵从套路即可。
所以，解动态规划问题本质还是找出“原问题的解如何由子问题组合而成”，这是动态规划的本质。

4. 到这里就可以说明一下动态规划与递归、caching的关系，
（1）递归只是动态规划的一种实现方式，属于“自顶向下”的实现方式。一个动态规划问题在实现的时候完全是可以用非递归的方式来编码实现的。
所以，递归不是动态规划的本质。
（2）Caching是动态规划实现过程中提升计算效率的一种方法，它的想法就是把计算过的值存起来，下次同样的计算过程直接使用保存的结果即可。
Caching既可以用于递归实现，也可以用在非递归实现。
所以，Caching也不是动态规划的本质。

4. “自顶向下”的实现方式 和 “自低向上”的实现方式各有什么优缺点，我的理解如下。
两种方法的取舍我个人的喜好是——优先选择Top-Down dynamic programming，除非不容易得到递归公式或空间复杂度无法接受。

“自顶向下”（top-down dynamic programming）：
1.能方便的得到递归公式，并用递归函数实现
2.保持了递归实现的代码结构，逻辑上容易理解。
3.过程中只计算需要计算的子结果。
4.当采用了caching技术时多次被调用时天然的复用之前被调用时的子结果。（比如连续两次计算fibonacci数F(4), F(5),则计算F(5)时已知F(3)和F(4)，直接相加即可）

“自低向上”（bottom-up dynamic programming）：
1.需要设计数据结构来完成自底向上的计算过程。逻辑上相对不那么直观。
2.常常可以进行空间复杂度的优化。比如Fibonacci数列可以优化为只使用两个变量的额外存储空间，0-1背包问题可以优化为O(n)的空间复杂度。
3.若进行了空间优化，则连续多次被调用时无法复用之前被调用的子结果。

最后，如果想要更全面完整的理解动态规划，我的学习路线上只有两步：
1. 读一下Algrithm in C这本书的相应章节
2. 把几个经典动态规划问题分别用“自顶向下”和“自低向上”两种方法实现一遍，用任何你喜欢的语言都行
：）祝开心

编辑于 2015-07-01

Mingqi
Professional Stranger

512 人赞同了该回答

维基百科上对于动态规划是这么定义的：

> **dynamic programming** is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions.

Ref: Dynamic programming

也就是说，动态规划一定是具备了以下三个特点：

把原来的问题分解成了几个**相似的子问题**。（强调“相似子问题”）
所有的子问题都**只需要解决一次**。（强调“只解决一次”）
储存子问题的解。（强调“储存”）

直接读这个定义还是比较玄乎抽象的。这里，我想结合两个实际的例子，来分别回答题主提出的两个问题：

什么是动态规划？
动态规划的意义是什么？

什么是动态规划？

我们来看看最经典的——斐波那契数列(Fibonacci)的例子！

1，1，2，3，5，8，13，21 ...

如果我们把第n个斐波拉契数记作 Fibonacci(n)，那么怎样利用动态规划来计算这个数呢？

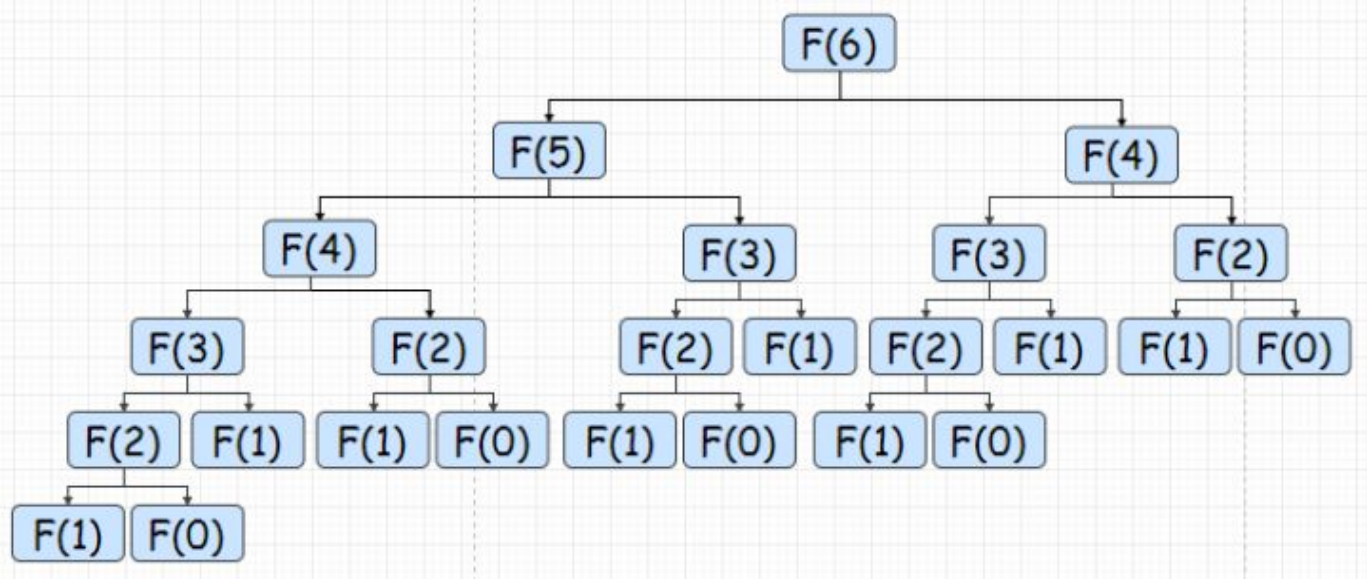
根据动态规划的三个特点，首先，**我们需要把原问题分解成几个相似的子问题**。这里还蛮清晰的啦，子问题就是这两个：Fibonacci(n-1) 和 Fibonacci(n-2)。

而原问题和子问题之间的关系是：

$$Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)$$

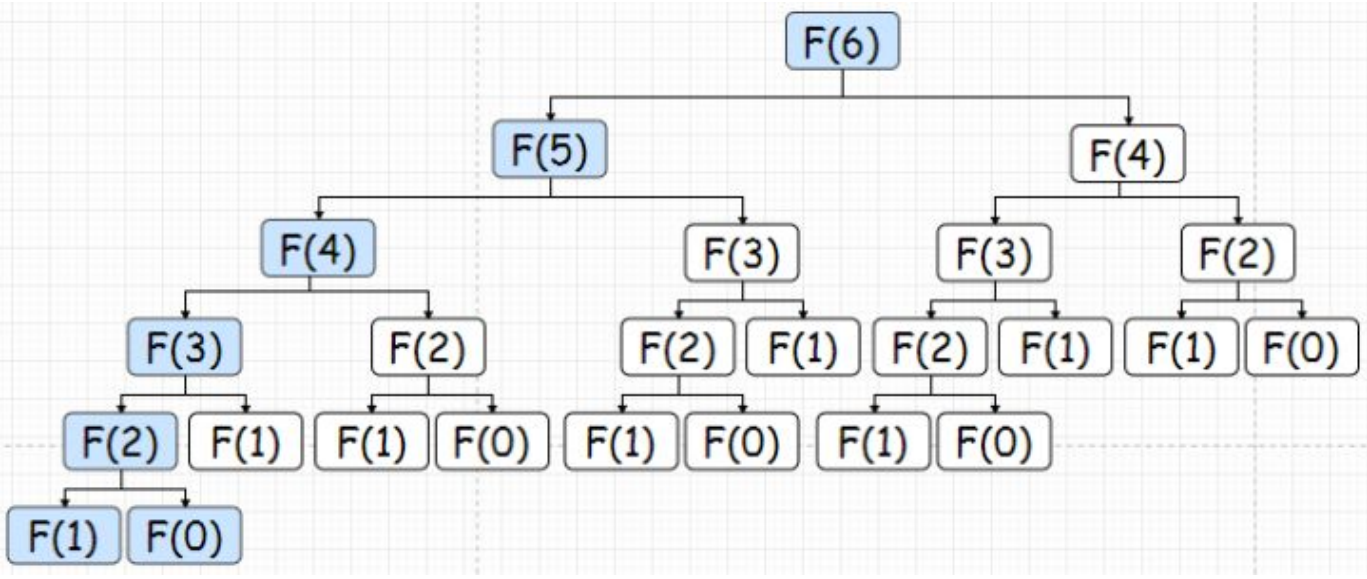
(其中Fibonacci(0)=Fibonacci(1)=1)

假设我们现在需要计算n=6的时候，斐波拉契的值，那么我们就需要计算他的子问题Fibonacci(5)和Fibonacci(4)。同理对于Fibonacci(5)我们需要计算Fibonacci(4)和Fibonacci(3)，对于Fibonacci(4)我们需要计算Fibonacci(3)和Fibonacci(2)。这样的话，最后我们需要计算的东西如下图——由最顶向下不停的分解问题，最后往上返回结果。

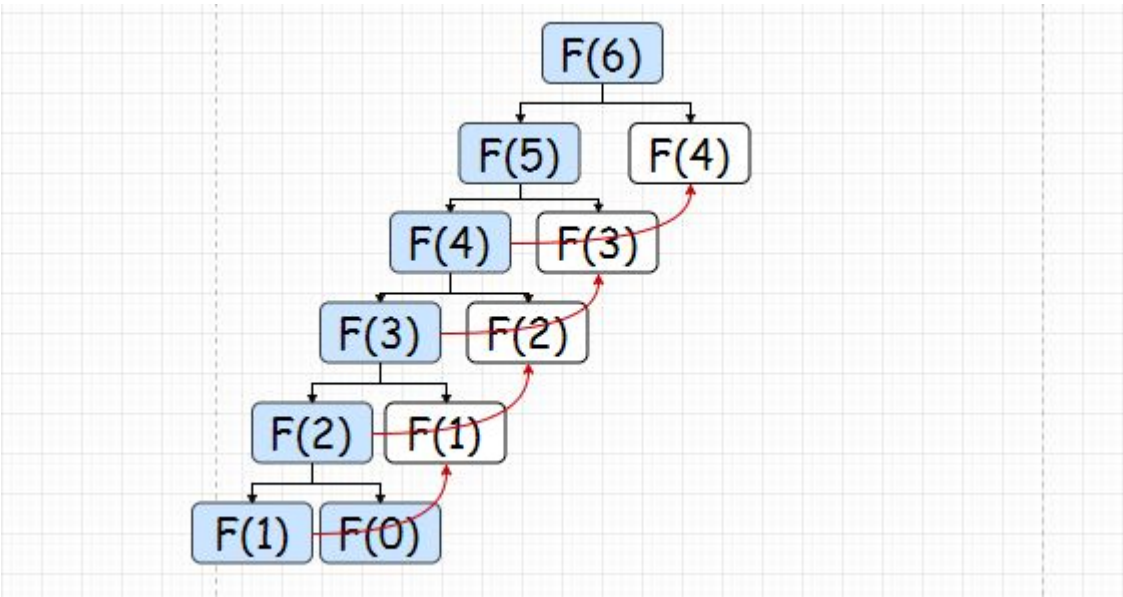


显然，这是一个非常低效的方法，因为其中有大量的重复计算。并且不满足动态规划的第二个特点：**“所有的子问题都只需要解决一次“。**

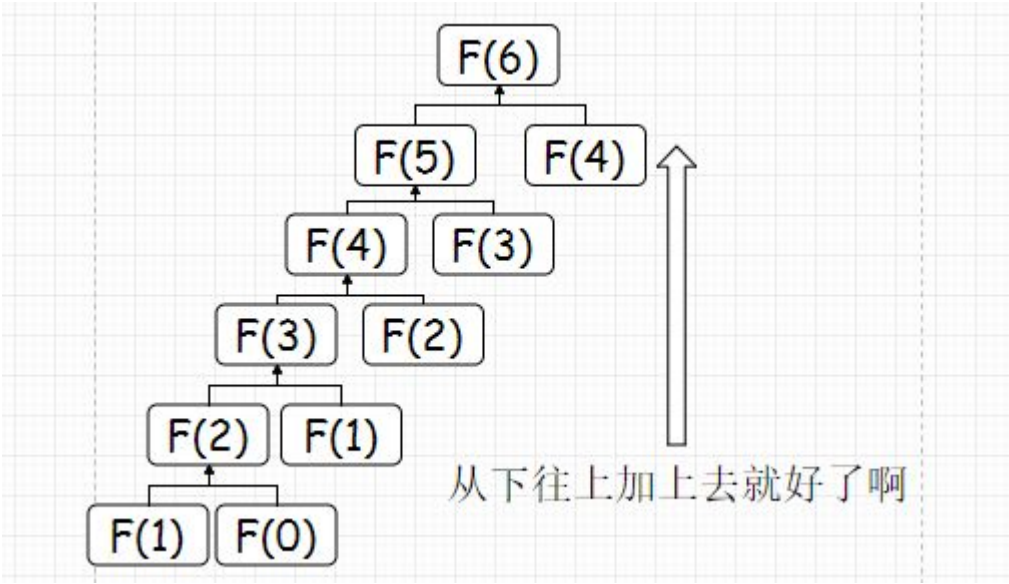
这个问题很好解决，我们只需要利用动态规划的第三个特点——**”储存子问题的解“**就可以了。比如，如果已经计算过Fibonacci(3)，那么把结果储存起来，其他地方碰到需要求Fibonacci(3)的时候，就不需要计算，直接调用结果就行。这样做之后，蓝色表示需要进行计算的，白色表示可以直接从存储中取得结果的：



也就是如下的计算（红色箭头表示调用了储存的数据，并未进行计算）：



看到这里，你可能会问这样一个问题，如果要计算Fibonacci(6)的话，何必这样大费周章的又是分解问题，又是储存的呢？直接这样从Fibonacci(0)和Fibonacci(1)开始一直往上一步步加就是了啊，看：



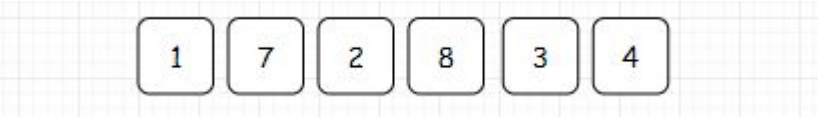
又直观又简洁，为啥要用动态规划呢？又要分解问题又要储存啥的。似乎看起来，动态规划的作用并不大的样子。

动态规划的意义是什么？

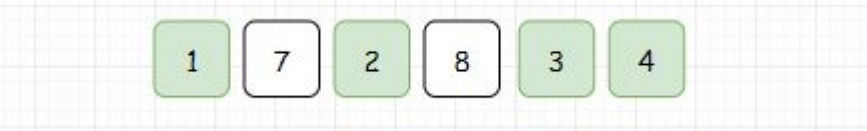
对于上面斐波拉契数列的例子，其实还真就是这样。动态规划至顶向下的算法和暴力的从底向上的算法其实没有太大区别。

我们再来看另外一个列子：求一个数列中最长上升子数列的长度（LIS）的问题。

比如，给一个数列：

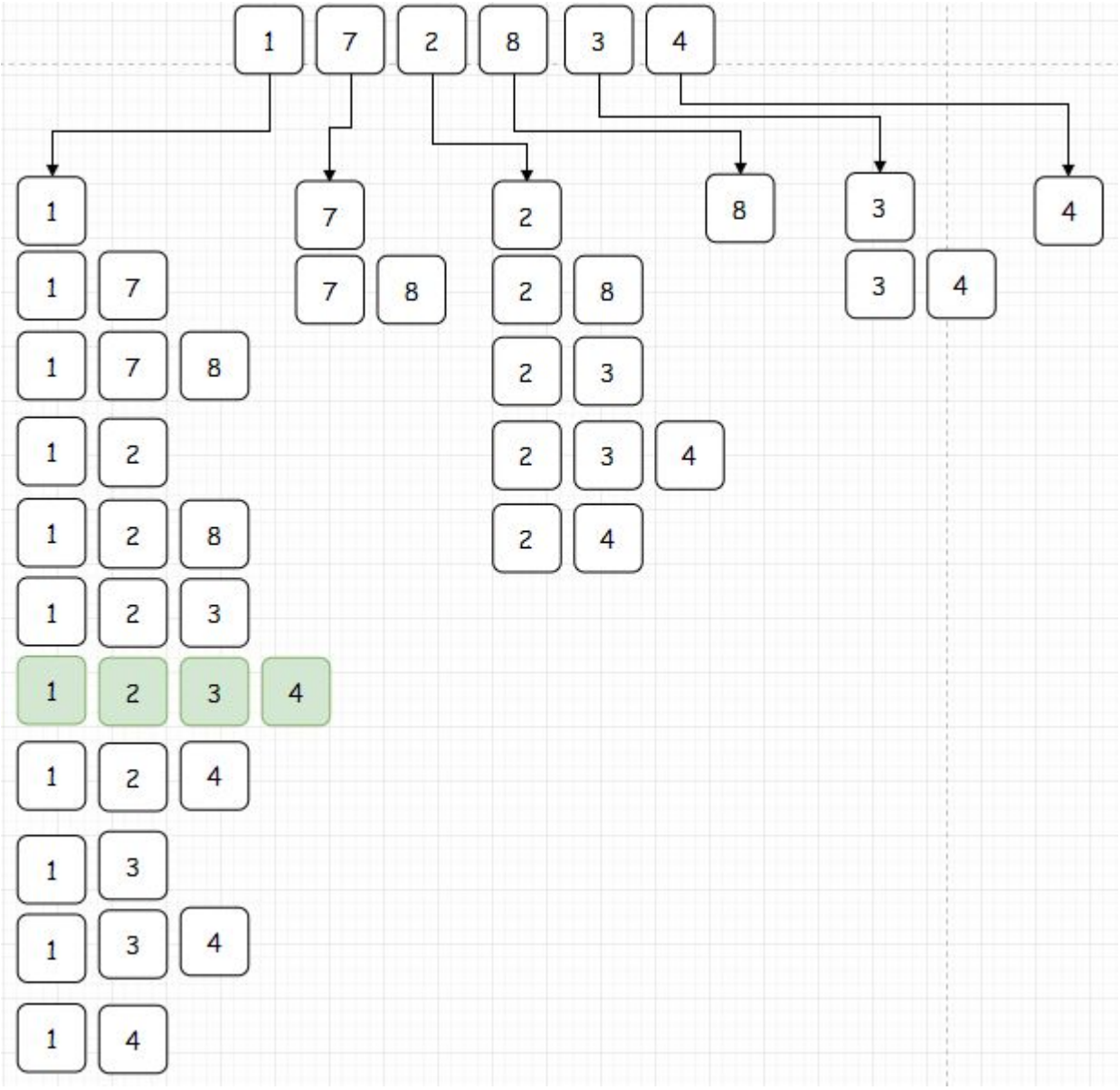


他的最长上升子数列就是：



[1,2,3,4], 长度为4，所以这个数列的最长上升子数列长度就是4。

对于这类问题我们如何求解呢？我们这次先用暴力来解一下试试，还是用上面那个数列作为例子：



古老的穷举法！直接这样找出所有的上升子序列，然后用肉眼观察哪个是最长的。显然，1,2,3,4是最长的，所以最长上升子序列的长度是4。

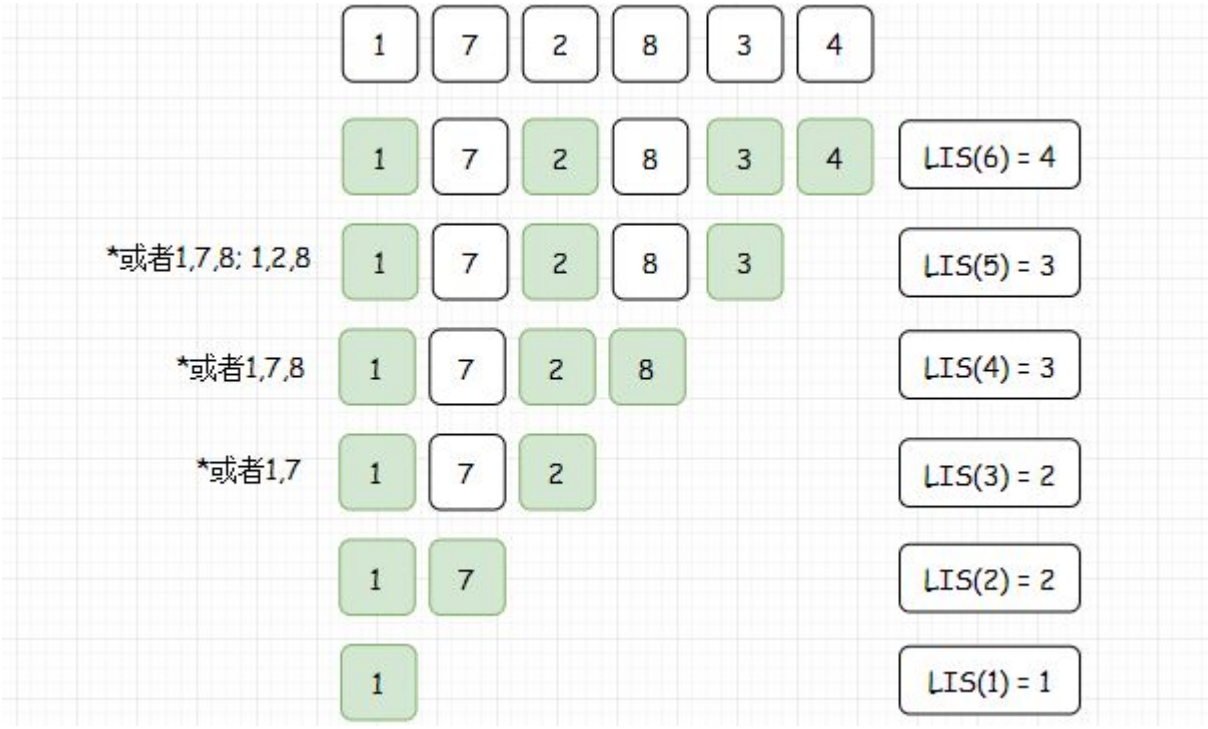
我们来看看这个方法的时间复杂度：

$$O(C_n^1 + C_n^2 + C_n^3 + \ldots + C_n^n) = O(C_n^{n/2}) = O(n!)$$

这就太消耗时间了。我们现在用动态规划试一下，看看有什么惊喜。

根据动态规划的定义，首先我们需要把原来的问题分解成了几个相似的子问题。但是，不同于斐波拉契数列的例子，这个如何分解原问题并不是那么一目了然。

原来的问题是求LIS(n)，现在我们需要找的就是LIS(n)和LIS(k)之间的关系1<=k<=n。通过肉眼观察一下：

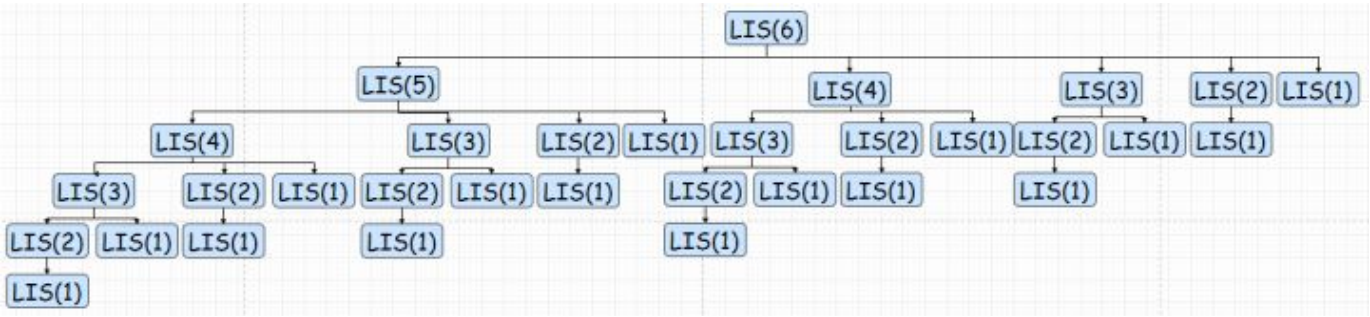


这里我们可以看到，LIS(K+1)要么等于LIS(K)，要么加了一。其实也很好理解，基本上就是，在前面所有的LIS种找到一个最长的LIS(i)，如果A(K)比这个找到LIS(i)的尾项A(i)要大，则LIS(K)=LIS(i)+1，否则LIS(K)=LIS(i)。

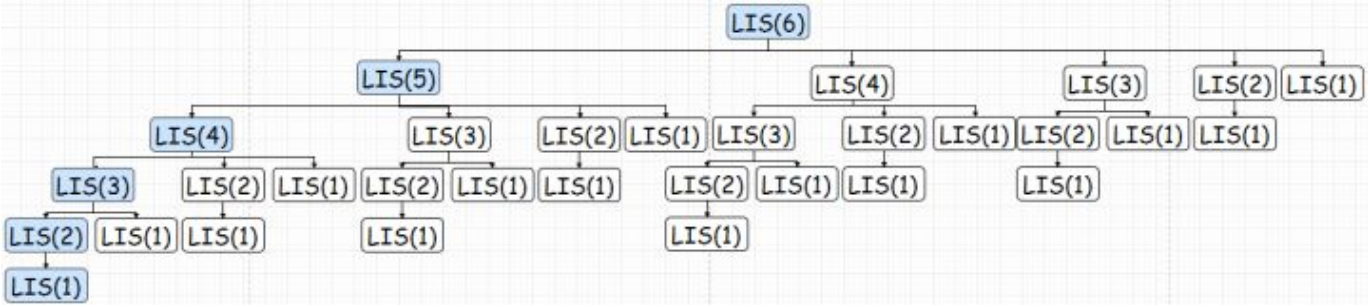
这样的话，我们就分解了原问题，并且找到了原问题和子问题之间的关系：

$$LIS(n) = max(LIS(1), LIS(2), LIS(3) \dots LIS(n-1)) + (A(n) > A(i)?1:0)$$

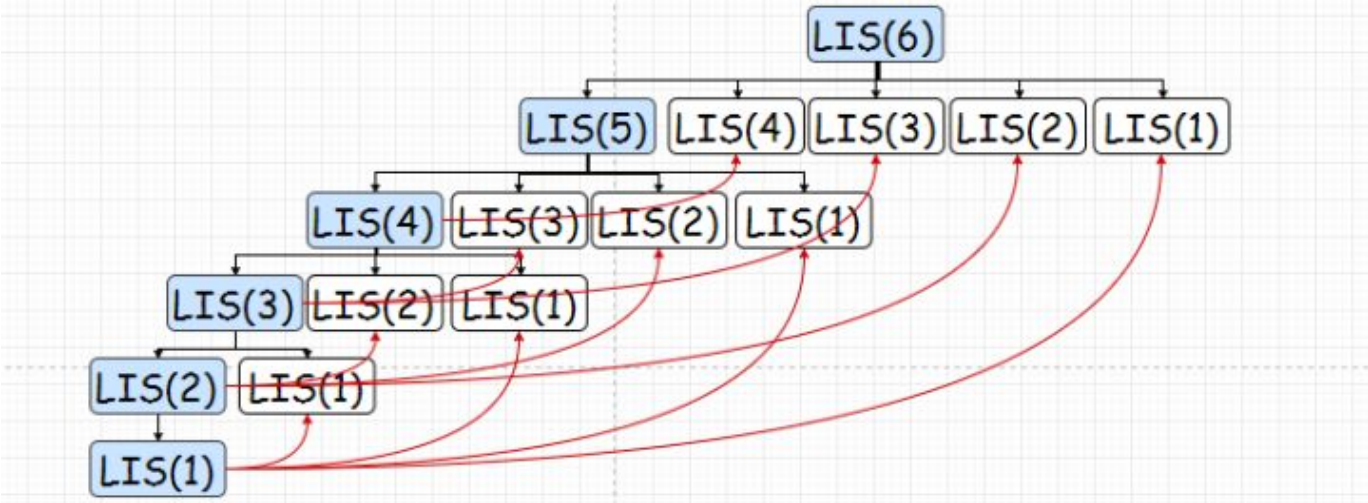
i是对应的最大LIS(i)。也就是说，计算LIS(n)需要计算之前所有的LIS(K)：



同理，我们可以储存子问题的结果，让每个子问题只被计算一次。需要计算的子问题就只剩下蓝色标出的部分了：



也就是（红色箭头表示调用了储存的数据，并未进行计算）：



我们可以看到，采用了动态规划之后，时间复杂度大大降低了：纵轴方向的递归计算返回时间复杂度是O(n)，横轴方向每行求Max的时间复杂度是O(logn)，所以总共的时间复杂度就是O(nlogn)，远远小于暴力穷举法的O(n!)。屌的一匹！

这就是动态规划的意义！在某些类型的问题中，动态规划通过巧妙的分治和记录子问题的解，可以给原本时间开销巨大的问题提供时间效率上大大优化的解决方案。

动态规划的三个特点都很重要，少了任何一个，都会有可能导致动态规划发挥不了应有的威力。比如在第一个斐波拉契数列的例子中，如果我们只是分解成了相似的子问题，但是并不储存子问题的解，那么最后那个算法的时间复杂度其实是指数级别的O(2^n)，反而比直观解法更慢了。同样的，**动态规划只能优化一部分问题。**比如最长递增子序列的暴力解法VS动态规划解法是O(n!) vs O(nlogn)，有很强的优化效果，但是斐波拉契数列的暴力解法VS动态规划解法是O(n) vs O(n)，动态规划并不能起到优化的效果。

在学生时代的时候，我一开始也一直对于动态规划这一部分比较困惑，主要大多数教材和讲解都运用了大量抽象的公式和不太容易理解的数学语言。这里，我希望借助视觉化算法的过程，以及尽量通俗的语

言和例子，能让你大概了解到，动态规划到底是个啥？就Ok了。本题下很多答案都回答的很好，看了很多，受益匪浅。就像很多答主提到过的，动态规划之所以比较难以掌握，在于**如何把原问题分解为相似的子问题**，以及**如何找到原问题和子问题之间的关系**。这两点需要大量的练习和总结才能熟练运用。

最后，说一点题外话。

@徐凯强 Andy 的最高票答案，回答的很好，也很有参考价值。但是他有一点是我不太喜欢的，就是他的答案里不停的强调**动态规划的本质对问题状态的定义和状态转移方程的定义，而储存重叠子问题的解只是一种“技巧”，与动态规划的本质无关。**这对很多人是有一定误导性的（我自己曾经就困扰了很久动态规划和分治法的区别到底是啥）。他的答案中，引用维基百科对于动态规划的定义的时候，不知是有心还是无意，他只节选了前半句：

> dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems.

但是，实际上，维基百科对于动规的定义里，还有更重要的后半句他漏掉了：

> dynamic programming is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, **solving each of those subproblems just once, and storing their solutions.**

Ref: Dynamic programming

也就是说，没有后面半句，只有前面半句，根本不能算是动态规划，只能算是分治法。**储存重叠子问题的解，并不只是一个“花费空间来节省时间的技巧”，而就是动态规划的定义和核心之一。**我不知道最高票答主是有意为了支持自己观点而故意断章取义了维基百科的对于动态规划的定义，还是无心而为之。我希望是后者。

编辑于 2018-01-31

端泽
杂学进行中

162 人赞同了该回答

搞过ACM的水货答一下。
排名第一的答案本身已足够好了，但还是太过专业，不能传教于大众，故试着来个**通俗的答案**。

首先，**动态规划**是一种**算法**。那么，何谓算法？计算机书籍中不难找到其严谨的学术定义，大众可以简单理解为“**解决某一类问题的核心思想**”。

先谈动态规划的意义——望文生义，“动态”规划对应“动态”的问题：你并不知道问题的规模会有多大，而不论是个位数还是百万级，都能以**较快速度(**动态规划是一种泛用性算法，而泛用性算法与特定算法相比往往存在性能差距**)**将结果正确计算出来。这是对于计算机科学最直观的意义，当然我认为其**对人生亦有一定指导意义**，但那是见仁见智的事了。

动态规划这一思想的实质其实是以下两点：
1.分析问题，构造状态转移方程
2.以空间换时间

让我们结合一个简单例子来理解一下：
以乘法计算为例，乘法的定义其实是做n次加法，请先忘掉九九乘法表，让你计算9*9，如何得到81这个解？计算9*10呢？9*999……以及9*n呢？
1.分析问题，构造状态转移方程
“状态转移方程”的学术定义亦可简单找到（比如置顶答案），略去不表。光看“方程”二字，可以明白它是一个式子。
针对以上问题，我们构造它的状态转移方程。
问题规模小的时候，我们可以容易得到以下式子：
9*0=0；
9*1=0+9；
9*2=0+9+9；
……
可以得到：9*n=0+9+…+9(总共加了n个9)。严谨的证明可以使用数学归纳法，略去不表。
现在，定义dp(n)=9*n,改写以上式子：
dp(0)=9*0=0；
dp(1)=9*1=dp(0)+9；
dp(2)=9*2=dp(1)+9；
……
作差易得：dp(n)=dp(n-1)+9；这就是**状态转移方程**了。
可以看到，有了状态转移方程，我们现在可以顺利求解9*n（n为任意正整数）这一问题。
2.以空间换时间
虽然能解，但当n很大时，计算耗时过大，看不出状态转移方程dp(n)=dp(n-1)+9与普通方程9*n=0+9+…+9(总共加了n个9)相比没有任何优势。
这时，如果dp(n-1)的结果已知，dp(n)=dp(n-1)+9只需计算一次加法，而9*n=0+9+…+9(总共加了n个9)则需计算n-1次加法，**效率差异一望即知**。
存储计算结果，可令状态转移方程加速，而对普通方程没有意义。
以空间换时间，是令动态规划具有实用价值的必备举措。

发布于 2014-12-18

空明流转
计算机图形学、C++、编程 话题的优秀回答者

101 人赞同了该回答

从实现的角度上来说，动态规划可以说是记忆化搜索的一种更加特化，或者也可以说更加优化的形式。

记忆化搜索很好理解，就是把之前计算过的保存下来，下次查询到的时候不再做重复计算。而且它的约束其实和动态规划是非常接近的，就是要求无后效性 —— 如果有后效性，那就考虑把“后效”也加入到状态中，直到可以把问题变成无后效的 —— 如果这样做也解决不了，那就不适合记忆化搜索，更不适合动态规划了。
所以很多动态规划问题的难点就在于，如何判断问题的某一个属性是否是状态的一部分，使得整个状态呈现无后效性。

这里给大家举个例子。考虑驾车从A到B点，中间选择不同的驾驶策略（加速，减速，匀速），在不同的路况上（上坡，下坡，平路）会有不同的油耗。这里我们要选择最小油耗的驾驶方案，使得它能满足：
1. A和B点的速度为0。
2. A到B点的最高速度不大于S。
3. 油耗最低。

把这个问题离散化考虑，比如一共有10000个段，因为每个段都有加速，匀速，减速三种选择，在不考虑约束的情况下，就有3^n种可能的策略。即便考虑约束，整个搜索状态空间也是非常大的。所以有必要把这个问题分而治之。对于这个问题，一个最基本的分治的策略就是，选取某个中间的位置C，显然，
MIN(F(AB)) = MIN(F(AC)) + MIN(F(CB))。
对于C保存什么样的状态，最简单的想法肯定是AC的里程以及油耗。但是仔细思考一下时候你会发现，如果C仅仅保留里程信息其实是不对的，因为C点的速度，也会影响到最终的状态。如果AC路程相同油耗相同，C点时的车速越快，CB段所需要的燃油就会越少（当然要注意的是还有最高速度和首末点速度为0的约束。所以此时，这里的C不仅要保留里程信息，更加要保留当前的速度。这时候，你就可以说，AC段和CB段彼此是独立的，C点所选取的状态无后效性。

回到记忆化搜索和动态规划的关系上。记忆化搜索它有一个问题，就是你并不知道你之前解过的子问题会不会再次出现，所以需要保留几乎所有的结果。这样会导致很高的空间复杂度。而动态规划对问题做了进一步约束之后，会告诉我们如何安排计算顺序以尽可能最少保存中间结果；以及哪些子问题是需要保留的，又在怎样的计算完成之后就可以抛掉了。

这样不管是递归还是递推，都极大的减少了中间存储。

编辑于 2019-04-10

Tim Shen
Twitter: @timshen_

167 人赞同了该回答

// 发现上面已经有人提到了最短路了。。但是动态规划真的不喜欢有环的图啊QAQ

动态规划不是个具体的算法，是一个框架。框架是死的，但框架里填什么全看人。

这个框架就是有向无环图。图中每个点都存着一个值。你去把图论刷一遍，弄弄拓扑排序之后，就会明白，动态规划就是对图中所有点执行拓扑排序，再依照拓扑序让每个点更新自己指向的节点的值的过程。

至于图中点怎么定义，边怎么定义，值怎么定义，“更新”怎么定义，那就是考验建图论模型的能力了，我也没能力讲出个系统规律来。

这时候动态规划那些概念都能一一对应了。所谓最优子结构就是我可以证明只要按着拓扑序来更新，保证能得到最优解。无后效性呢，就是图不能有环。至于记忆化嘛，就是在每个点上存一个值了。递推嘛就是把拓扑序正着来一遍，递归嘛就是把拓扑序反着来一遍。都没什么新鲜东西的。

值得一提的是，根据上述的定义，动态规划不如最短路径强劲（觉得不好理解的请看后面的更新）。因为最短路径可以反复迭代，而动态规划必须一遍扫完。

也就是说，不要管什么动态规划了，只要是动态规划，最短路径都能解：）动态规划点里的值嘛就是最短距离了，一个点对另一个点的更新嘛就是最短路径的更新了。

若是连最短路径也不能活学活使，那还是背背书让老师开心就行了。

---------- 更新 ----------

对于有向无环图，按照拓扑序更新后继节点就是求最短路。通用的最短路算法在此依然适用。另外，通用的最短路算法仍然能解部分带环的图，所以在此可以把拓扑排序规约到最短路。当然有效率上的差别，但是我提这整个一大段只是希望用此模型辅助理解，降低学习曲线斜率而已，所以从“新的知识点越少越好”的角度，规约到最短路是最省力的了。

真写还是要注意效率的，能用循环就循环，能递归+记忆化就行。别傻乎乎地建图就行了。至于真正如何用代码实现，循环用什么姿势展开能达成拓扑序，有向无环图有没有分层关系从而能导致用滚动数组优化空间，都是一些后面要掌握的技巧了，但是应该机械得很。

拿“背包九讲”来说吧，可以尝试给0/1背包和无限背包问题画图，就明白为什么二者命令式语言实现中的循环是那么的似是而非。其实他俩状态图完全不一样，而且0/1背包的状态更复杂。但是通过对0/1背包进行滚动数组（严格来讲比滚动数组还变态，因为已经不是两个数组来回滚了，是同一个数组后面覆盖前面的）的优化，可以把存状态的数组减少到一维，结果代码就长得和完全背包只差一点点了。

具体点说，0/1背包的状态是二元组(i, j)表示花费为i时，前j个元素。状态中存值为最大能得到的利益。每个状态出两条边，即第j+1个物品选还是不选。这个图画出来是个二维的节点阵列，有明显的层次关系（j到j+1）。

而完全背包更简单，状态就是当前花费，每个状态的出边数和物品数相当。

顺带扯一句，这也是正则表达式中"a*"比"a{5, 81}“要好实现的原因。
至于树状动态规划，那就更贴近有向无环图的解法了，对树的后序遍历和拓扑排序如出一辙。

编辑于 2015-07-25

程序员吴师兄
公众号：五分钟学算法

23 人赞同了该回答

写了一篇新手向，新手可看看：）

在学习「数据结构和算法」的过程中，因为人习惯了平铺直叙的思维方式，所以「递归」与「动态规划」这种带循环概念（绕来绕去）的往往是相对比较难以理解的两个抽象知识点。

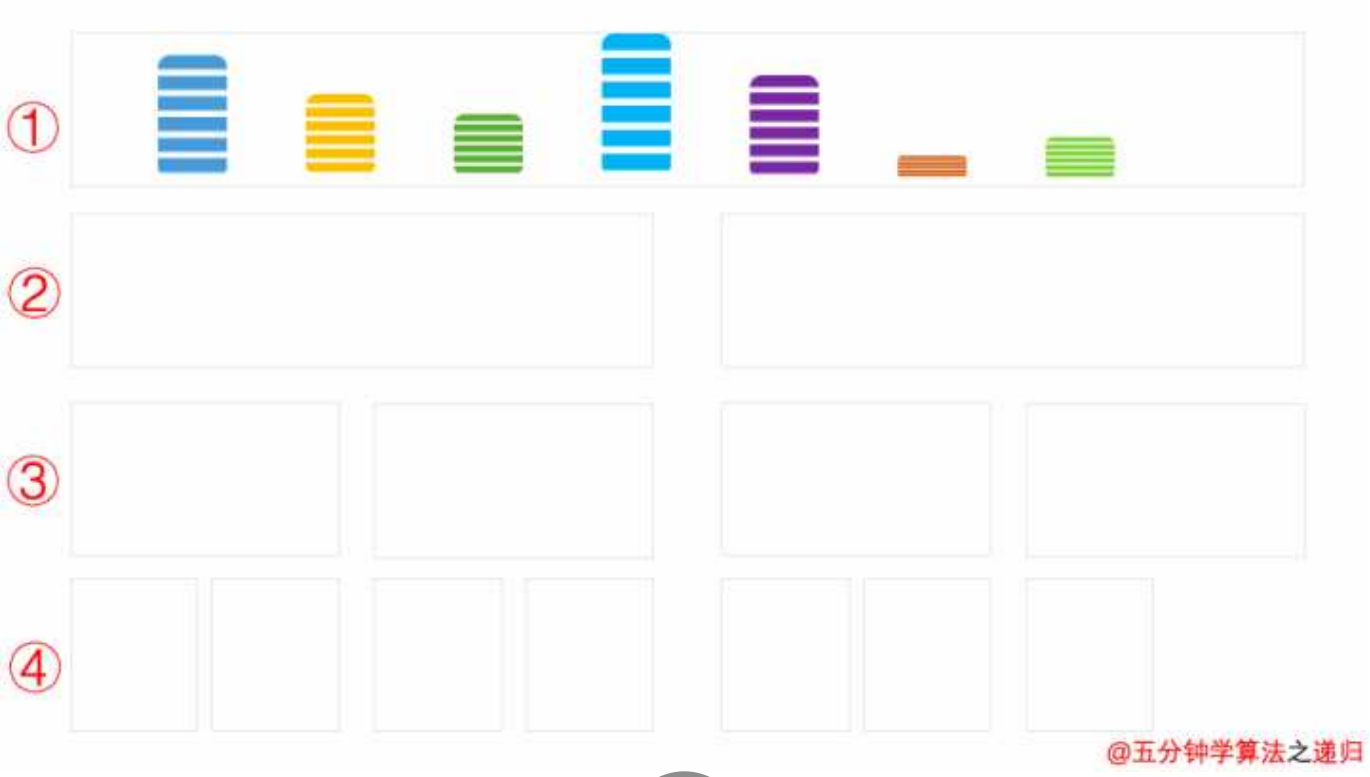
程序员小吴打算使用动画的形式来帮助理解「递归」，然后通过「递归」的概念延伸至理解「动态规划」算法思想。

什么是递归

先下定义：**递归算法是一种直接或者间接调用自身函数或者方法的算法。**

通俗来说，递归算法的实质是把问题分解成规模缩小的同类问题的子问题，然后递归调用方法来表示问题的解。它有如下特点：

1. 一个问题的解可以分解为几个子问题的解
2. 这个问题与分解之后的子问题，除了数据规模不同，求解思路完全一样
3. 存在递归终止条件，即必须有一个明确的递归结束条件，称之为递归出口



GIF

通过动画一个一个特点来进行分析。

1.一个问题的解可以分解为几个子问题的解

子问题就是相对与其前面的问题数据规模更小的问题。

在动图中①号问题（一块大区域）划分为②号问题，②号问题由两个子问题（两块中区域）组成。

2. 这个问题与分解之后的子问题，除了数据规模不同，求解思路完全一样

「①号划分为②号」与「②号划分为③号」的逻辑是一致的，求解思路是一样的。

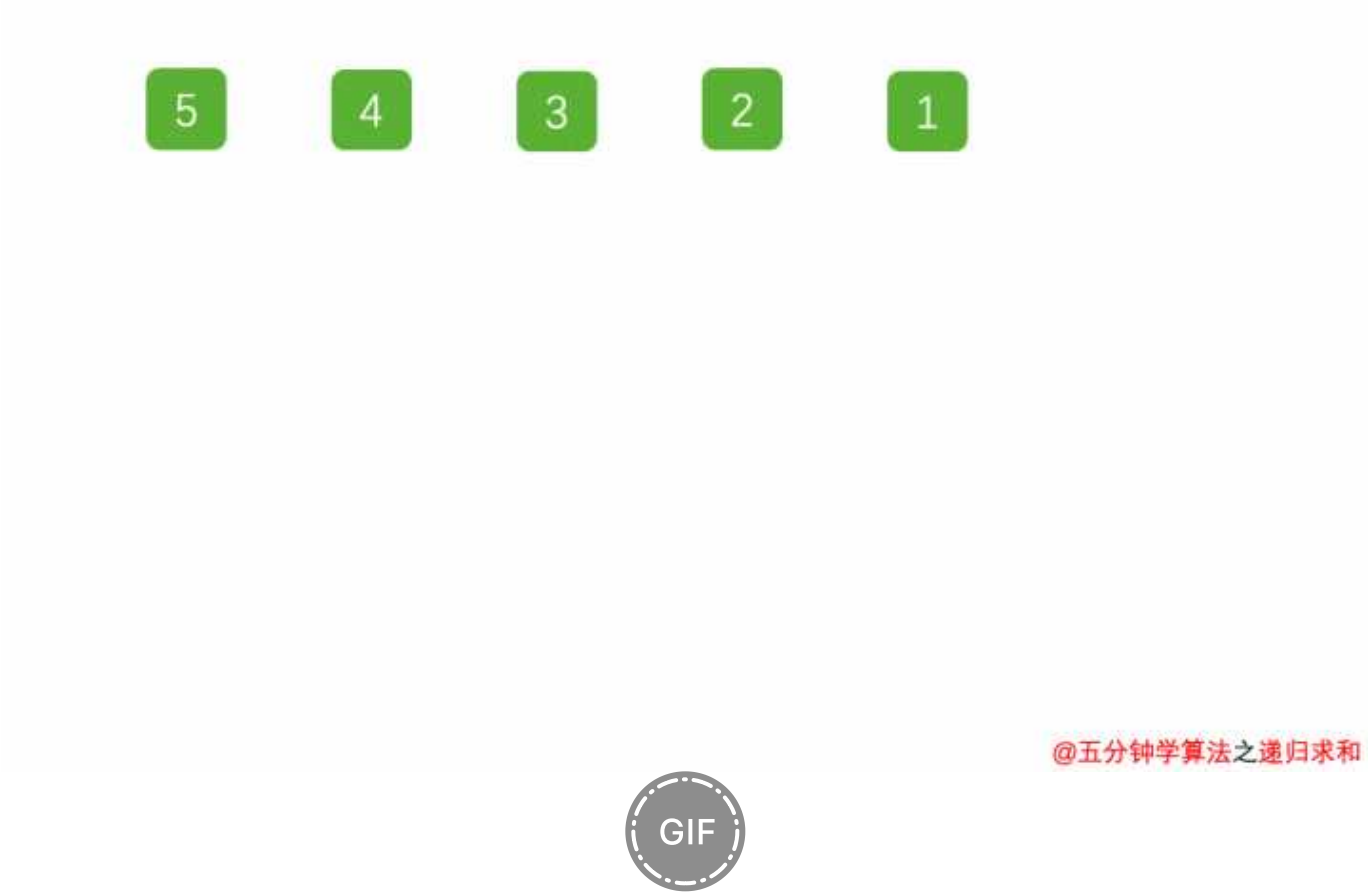
3. 存在递归终止条件，即存在递归出口

把问题分解为子问题，把子问题再分解为子子问题，一层一层分解下去，不能存在无限循环，这就需要有终止条件。

①号划分为②号，②号划分为③号，③号划分为④号，划分到④号的时候每个区域只有一个不能划分的问题，这就表明存在递归终止条件。

从递归的经典示例开始

一.数组求和




```
1Sum(arr[0...n-1]) = arr[0] + Sum(arr[1...n-1])
```

后面的 Sum 函数要解决的就是比前一个 Sum 更小的同一问题。

```
1Sum(arr[1...n-1]) = arr[1] + Sum(arr[2...n-1])
```

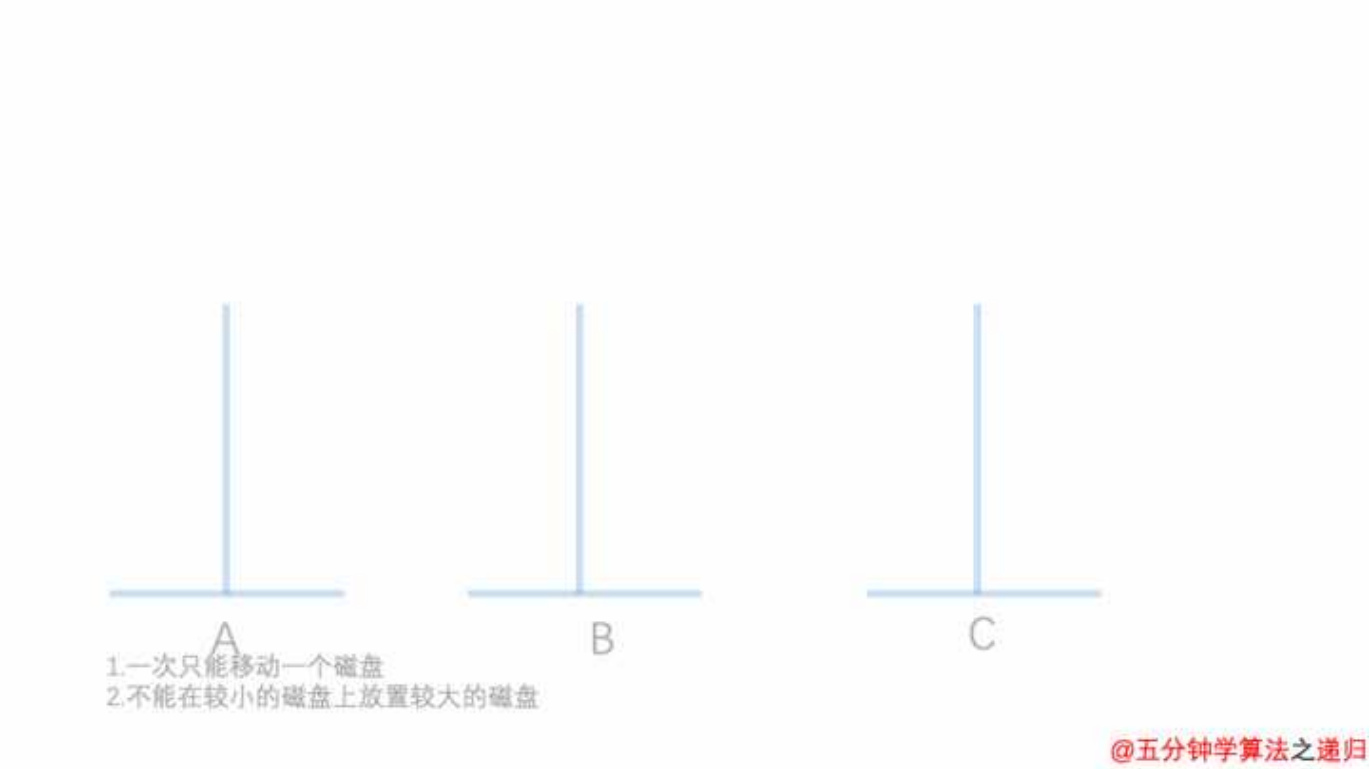
以此类推，直到对一个空数组求和，空数组和为 0 ，此时变成了最基本的问题。

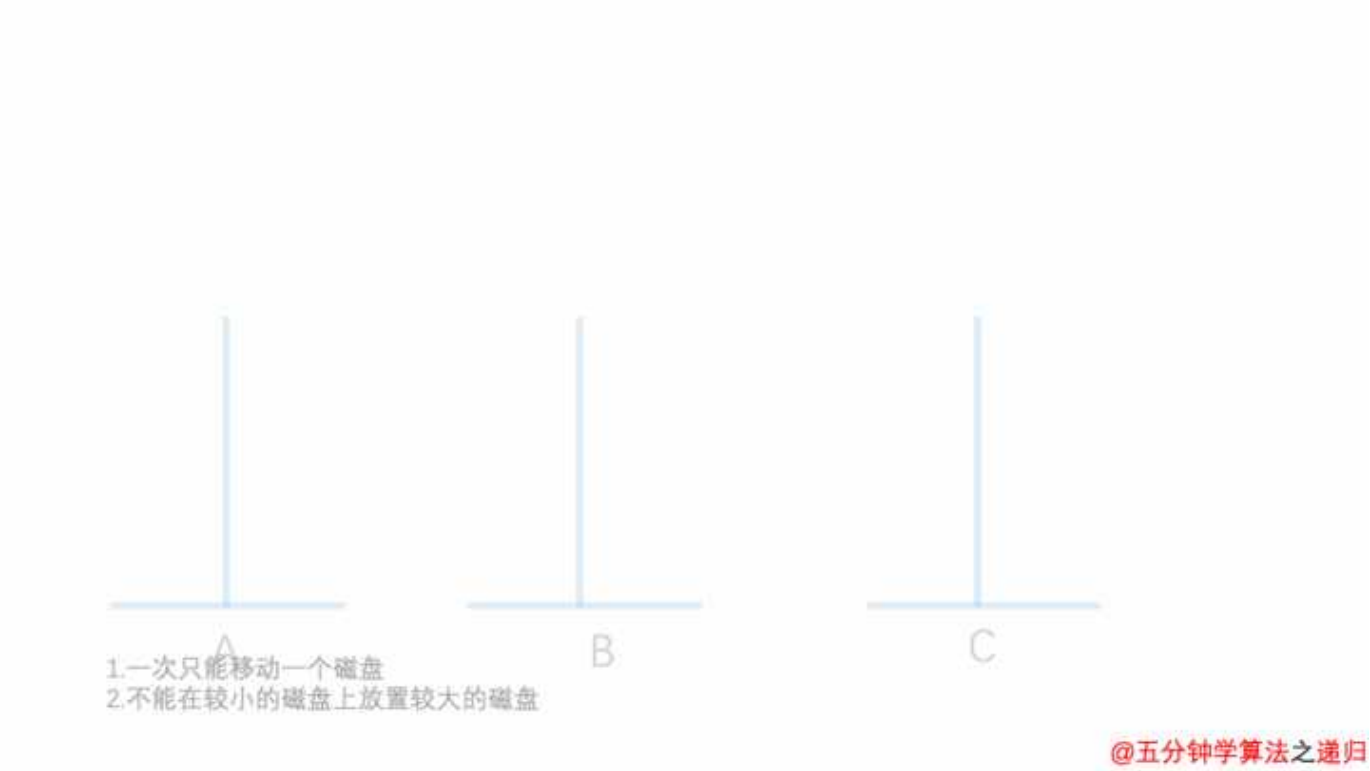
```
1Sum(arr[n-1...n-1] ) = arr[n-1] + Sum([])
```

二.汉诺塔问题

汉诺塔（Hanoi Tower）问题也是一个经典的递归问题，该问题描述如下：

> 汉诺塔问题：古代有一个梵塔，塔内有三个座A、B、C，A座上有64个盘子，盘子大小不等，大的在下，小的在上。有一个和尚想把这个盘子从A座移到B座，但每次只能允许移动一个盘子，并且在移动过程中，3个座上的盘子始终保持大盘在下，小盘在上。





GIF

① 如果只有 1 个盘子，则不需要利用 B 塔，直接将盘子从 A 移动到 C 。
② 如果有 2 个盘子，可以先将盘子 2 上的盘子 1 移动到 B ；将盘子 2 移动到 C ；将盘子 1 移动到 C 。这说明了：可以借助 B 将 2 个盘子从 A 移动到 C ，当然，也可以借助 C 将 2 个盘子从 A 移动到 B 。
③ 如果有 3 个盘子，那么根据 2 个盘子的结论，可以借助 C 将盘子 3 上的两个盘子从 A 移动到 B ；将盘子 3 从 A 移动到 C ，A 变成空座；借助 A 座，将 B 上的两个盘子移动到 C 。
④ 以此类推，上述的思路可以一直扩展到 n 个盘子的情况，将将较小的 n-1个盘子看做一个整体，也就是我们要求的子问题，以借助 B 塔为例，可以借助空塔 B 将盘子A上面的 n-1 个盘子从 A 移动到 B ；将A 最大的盘子移动到 C ， A 变成空塔；借助空塔 A ，将 B 塔上的 n-2 个盘子移动到 A，将 C 最大的盘子移动到 C， B 变成空塔。。。

三.爬台阶问题

问题描述：

> 一个人爬楼梯，每次只能爬1个或2个台阶，假设有n个台阶，那么这个人有多少种不同的爬楼梯方法？

先从简单的开始，以 4 个台阶为例，可以通过每次爬 1 个台阶爬完楼梯：

爬台阶之一

@五分钟学算法之爬台阶

GIF

可以通过先爬 2 个台阶，剩下的每次爬 1 个台阶爬完楼梯

爬台阶之二

@五分钟学算法之爬台阶

GIF

在这里，可以思考一下：可以根据第一步的走法把所有走法分为两类：

① 第一类是第一步走了 1 个台阶
② 第二类是第一步走了 2 个台阶

所以 n 个台阶的走法就等于先走 1 阶后，n-1 个台阶的走法 ，然后加上先走 2 阶后，n-2 个台阶的走法。

用公式表示就是：

```
f(n) = f(n-1)+f(n-2)
```

有了递推公式，递归代码基本上就完成了一半。那么接下来考虑递归终止条件。

当有一个台阶时，我们不需要再继续递归，就只有一种走法。

所以 `f(1)=1` 。

通过用 `n = 2` ， `n = 3` 这样比较小的数试验一下后发现这个递归终止条件还不足够。

`n = 2` 时， `f(2) = f(1) + f(0)` 。如果递归终止条件只有一个 `f(1) = 1` ，那 `f(2)` 就无法求解，递归无法结束。
所以除了 `f(1) = 1` 这一个递归终止条件外，还要有 `f(0) = 1` ，表示走 0 个台阶有一种走法，从思维上以及动图上来看，这显得的有点不符合逻辑。所以为了便于理解，把 `f(2) = 2` 作为一种终止条件，表示走 2 个台阶，有两种走法，一步走完或者分两步来走。

总结如下：

① 假设只有一个台阶，那么只有一种走法，那就是爬 1 个台阶
② 假设有两个个台阶，那么有两种走法，一步走完或者分两步来走

通过递归条件：

```
1f(1) = 1;
2f(2) = 2;
3f(n) = f(n-1)+f(n-2)
```

很容易推导出递归代码：

```
1int f(int n) {
2  if (n == 1) return 1;
3  if (n == 2) return 2;
4  return f(n-1) + f(n-2);
5}
```

通过上述三个示例，总结一下如何写递归代码：

1.找到如何将大问题分解为小问题的规律
2.通过规律写出递推公式
3.通过递归公式的临界点推敲出终止条件
4.将递推公式和终止条件翻译成代码

什么是动态规划

介绍动态规划之前先介绍一下分治策略（Divide and Conquer）。

分治策略

将原问题分解为若干个规模较小但类似于原问题的子问题（**Divide**），「递归」的求解这些子问题（**Conquer**），然后再合并这些子问题的解来建立原问题的解。

因为在求解大问题时，需要递归的求小问题，因此一般用「递归」的方法实现，即自顶向下。

动态规划（Dynamic Programming）

动态规划其实和分治策略是类似的，也是将一个原问题分解为若干个规模较小的子问题，递归的求解这些子问题，然后合并子问题的解得到原问题的解。
区别在于这些子问题会有重叠，一个子问题在求解后，可能会再次求解，于是我们想到将这些子问题的**解存储起来**，当下次再次求解这个子问题时，直接拿过来就是。
其实就是说，动态规划所解决的问题是分治策略所解决问题的一个子集，只是这个子集更适合用动态规划来解决从而得到更小的运行时间。
即用动态规划能解决的问题分治策略肯定能解决，只是运行时间长了。因此，分治策略一般用来解决子问题相互对立的问题，称为标准分治，而动态规划用来解决子问题重叠的问题。

与「分治策略」「动态规划」概念接近的还有「贪心算法」「回溯算法」，由于篇幅限制，程序员小吴就不在这进行展开，在后续的文章中将分别详细的介绍「贪心算法」、「回溯算法」、「分治算法」，敬请关注：）

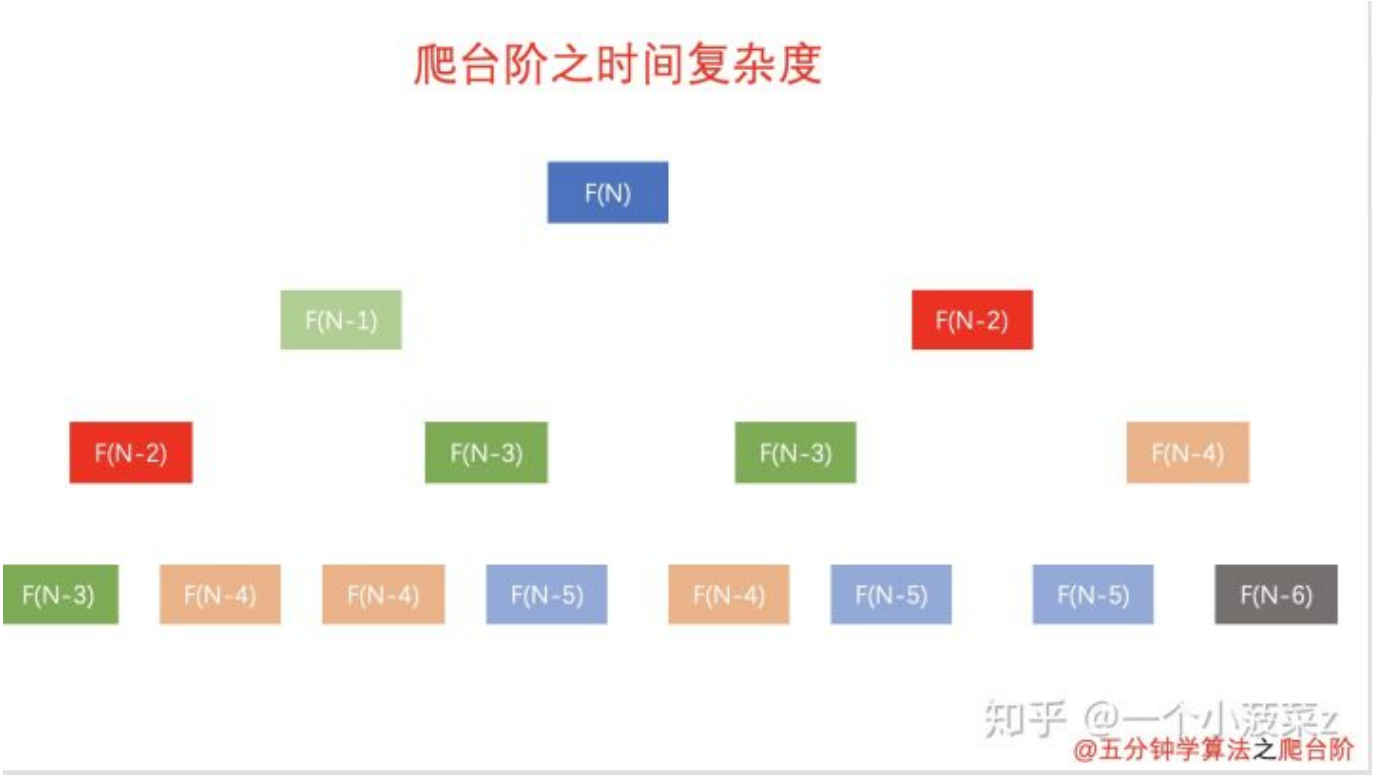
将「动态规划」的概念关键点抽离出来描述就是这样的：

1.动态规划法试图只解决每个子问题一次
2.一旦某个给定子问题的解已经算出，则将其记忆化存储，以便下次需要同一个子问题解之时直接查表。

从递归到动态规划

还是以 **爬台阶** 为例，如果以递归的方式解决的话，那么这种方法的时间复杂度为O(2^n)，具体的计算可以查看笔者之前的文章 《冰与火之歌：时间复杂度与空间复杂度》。

> 相同颜色代表着 爬台阶问题 在递归计算过程中重复计算的部分。



通过图片可以发现一个现象，我们是 自顶向下 的进行递归运算，比如： `f(n)` 是 `f(n-1)` 与 `f(n-2)` 相加， `f(n-1)` 是 `f(n-2)` 与 `f(n-3)` 相加。

思考一下：如果反过来，采取自底向上，用迭代的方式进行推导会怎么样了？

下面通过表格来解释 `f(n)` 自底向上的求解过程。

台阶数123456789走法数12

表格的第一行代表了楼梯台阶的数目，第二行代表了若干台阶对应的走法数。
其中 `f(1) = 1` 和 `f(2) = 2` 是前面明确的结果。

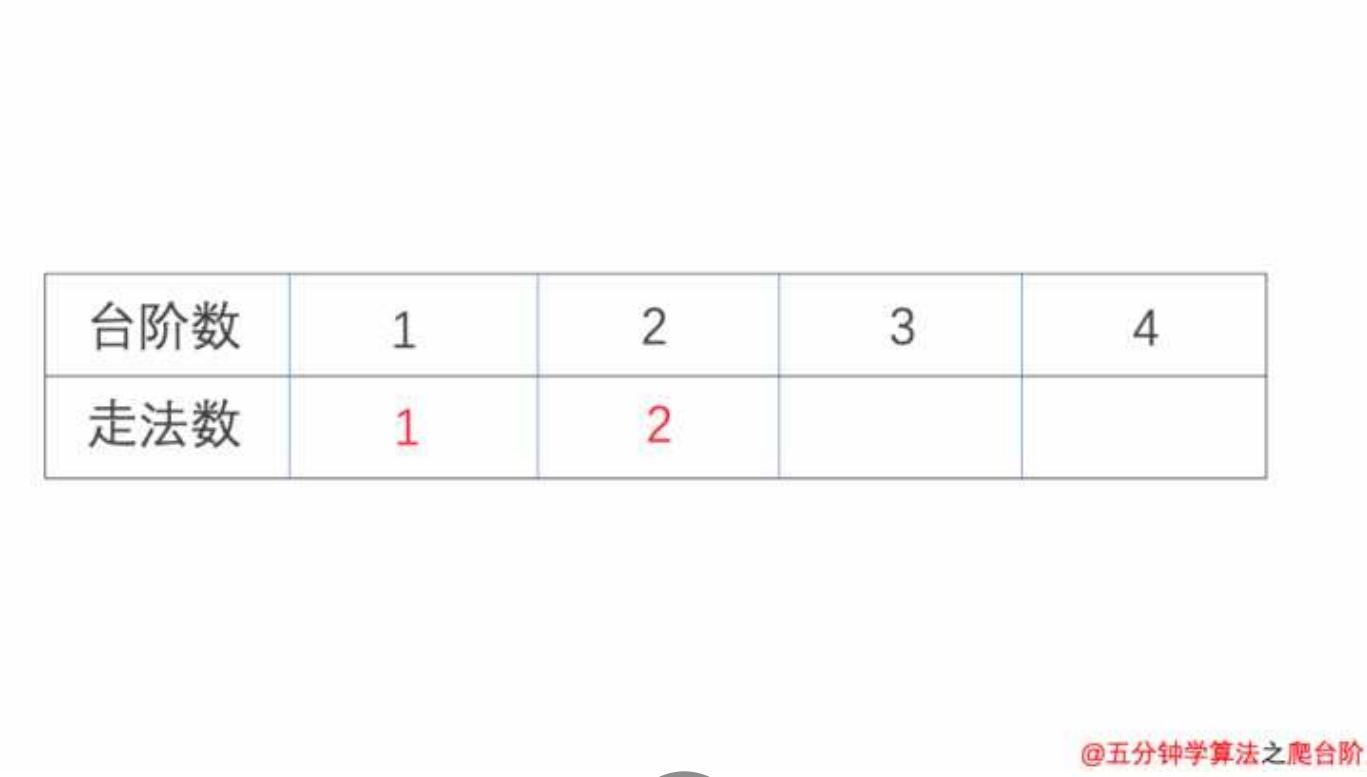
第一次迭代，如果台阶数为 3 ，那么走法数为 3 ，通过 `f(3) = f(2) + f(1)` 得来。

台阶数123456789走法数123

第二次迭代，如果台阶数为 4 ，那么走法数为 5 ，通过 `f(4) = f(3) + f(2)` 得来。

台阶数123456789走法数1235

台阶数	1	2	3	4
走法数	1	2		



由此可见，每一次迭代过程中，只需要保留之前的两个状态，就可以推到出新的状态。

> show me the code

```
int f(int n) {
    if (n == 1) return 1;
    if (n == 2) return 2;
    // a 保存倒数第二个子状态数据，b 保存倒数第一个子状态数据， temp 保存当前状态的数据
    int a = 1, b = 2;
    int temp = a + b;
    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，引入了temp变量。temp代表了当前迭代的结果值。

看一看出，事实上并没有增加太多的代码，只是简单的进行了优化，时间复杂度便就降为O(n)，而空间复杂度也变为O(1)，这，就是「动态规划」的强大！

详解动态规划

「动态规划」中包含三个重要的概念：

【最优子结构】
【边界】
【状态转移公式】

在「 爬台阶问题 」中

> `f(10) = f(9) + f(8)` 是【最优子结构】
> `f(1)` 与 `f(2)` 是【边界】
> `f(n) = f(n-1) + f(n-2)` 【状态转移公式】

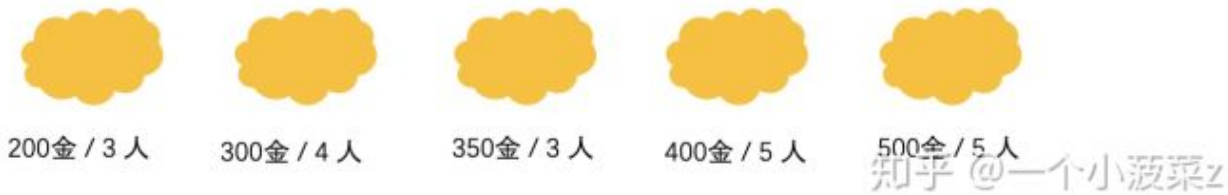
「 爬台阶问题 」 只是动态规划中相对简单的问题，因为它只有一个变化维度，如果涉及多个维度的话，那么问题就变得复杂多了。

难点就在于找出 「动态规划」中的这三个概念。

比如「 国王和金矿问题 」。

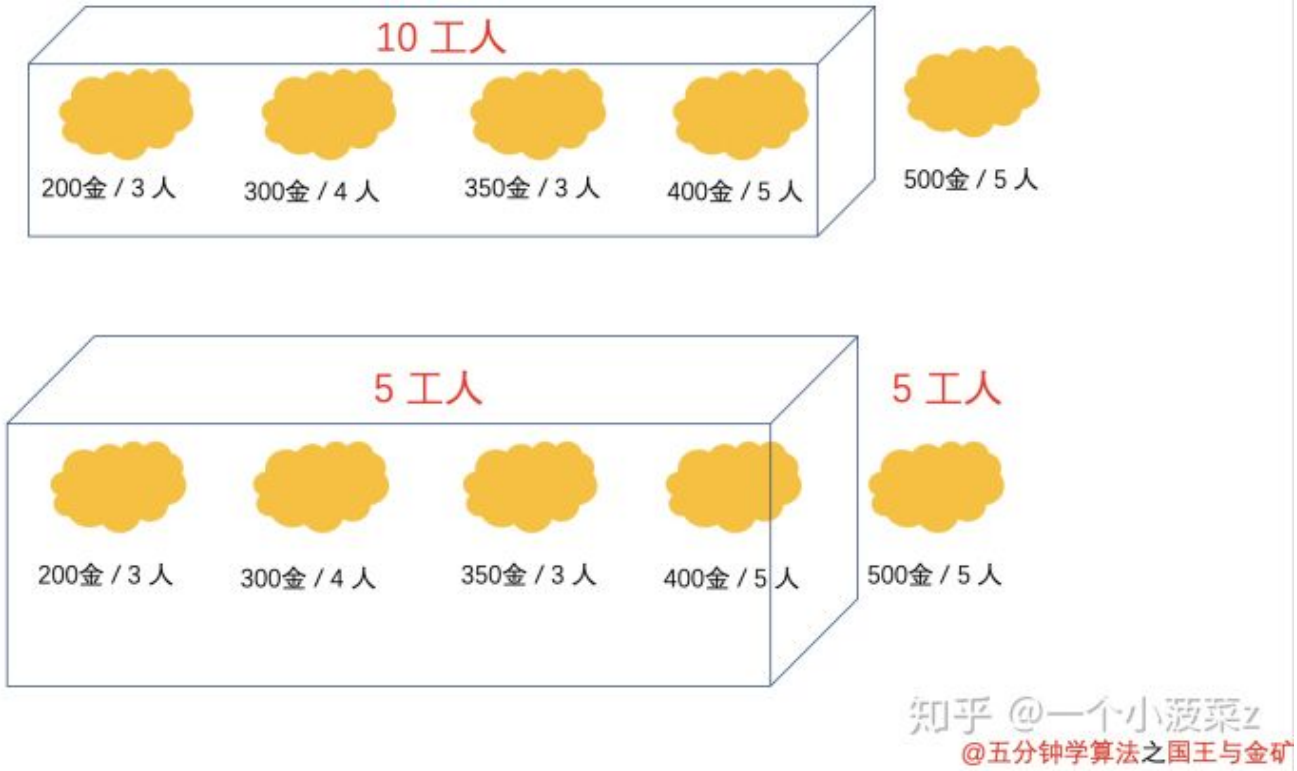
国王和金矿问题

> 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？



找出 「动态规划」中的这三个概念

国王和金矿问题中的【最优子结构】



国王和金矿问题中的【最优子结构】有两个：

① 4 金矿 10 工人的最优选择
② 4 金矿 （ 10 - 5 ） 工人的最优选择

4 金矿的最优选择与 5 金矿的最优选择之间的关系是

```
MAX[（4 金矿 10 工人的挖金数量），（4 金矿 5 工人的挖金数量 + 第 5 座金矿的挖金数量）]
```

国王和金矿问题中的【边界】

国王和金矿问题中的【边界】 有两个：

① 当只有 1 座金矿时，只能挖这座唯一的金矿，得到的黄金数量为该金矿的数量
② 当给定的工人数量不够挖 1 座金矿时，获取的黄金数量为 0

国王和金矿问题中的【状态转移公式】

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[]，得到【状态转移公式】：

边界值：F(n,w) = 0 (n <= 1, w < p[0])

```
F(n,w) = g[0] (n==1, w >= p[0])
```

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

```
F(n,w) = max(F(n-1,w), F(n-1,w-p[n-1]) + g[n-1]) (n &gt; 1, w &gt;= p[n-1])
```

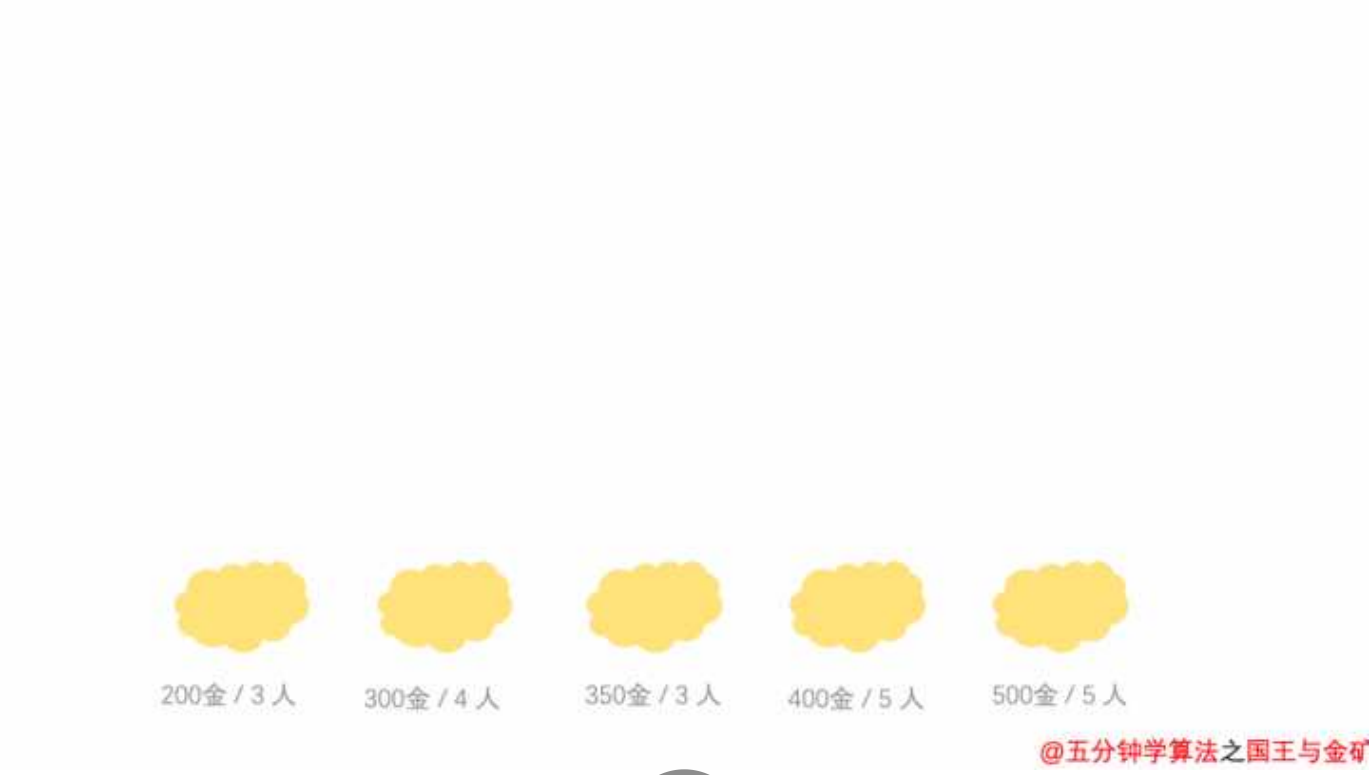
国王和金矿问题中的【实现】

先通过几幅动画来理解 「工人」 与 「金矿」 搭配的方式

1.只挖第一座金矿

在只挖第一座金矿前面两个工人挖矿收益为 零，当有三个工人时，才开始产生收益为 200，而后即使增加再多的工人收益不变，因为只有一座金矿可挖。

2.挖第一座与第二座金矿



GIF

在第一座与第二座金矿这种情况中，前面两个工人挖矿收益为 零，因为 W < 3,所以F(N,W) = F(N-1,W) = 0。

当有 三 个工人时，将其安排挖第 一 个金矿，开始产生收益为 200。

当有 四 个工人时，挖矿位置变化，将其安排挖第 二 个金矿，开始产生收益为 300。

当有 五、六 个工人时，由于多于 四 个工人的人数不足以去开挖第 一 座矿，因此收益还是为 300。

当有 七 个工人时，可以同时开采第 一 个和第 二 个金矿，开始产生收益为 500。

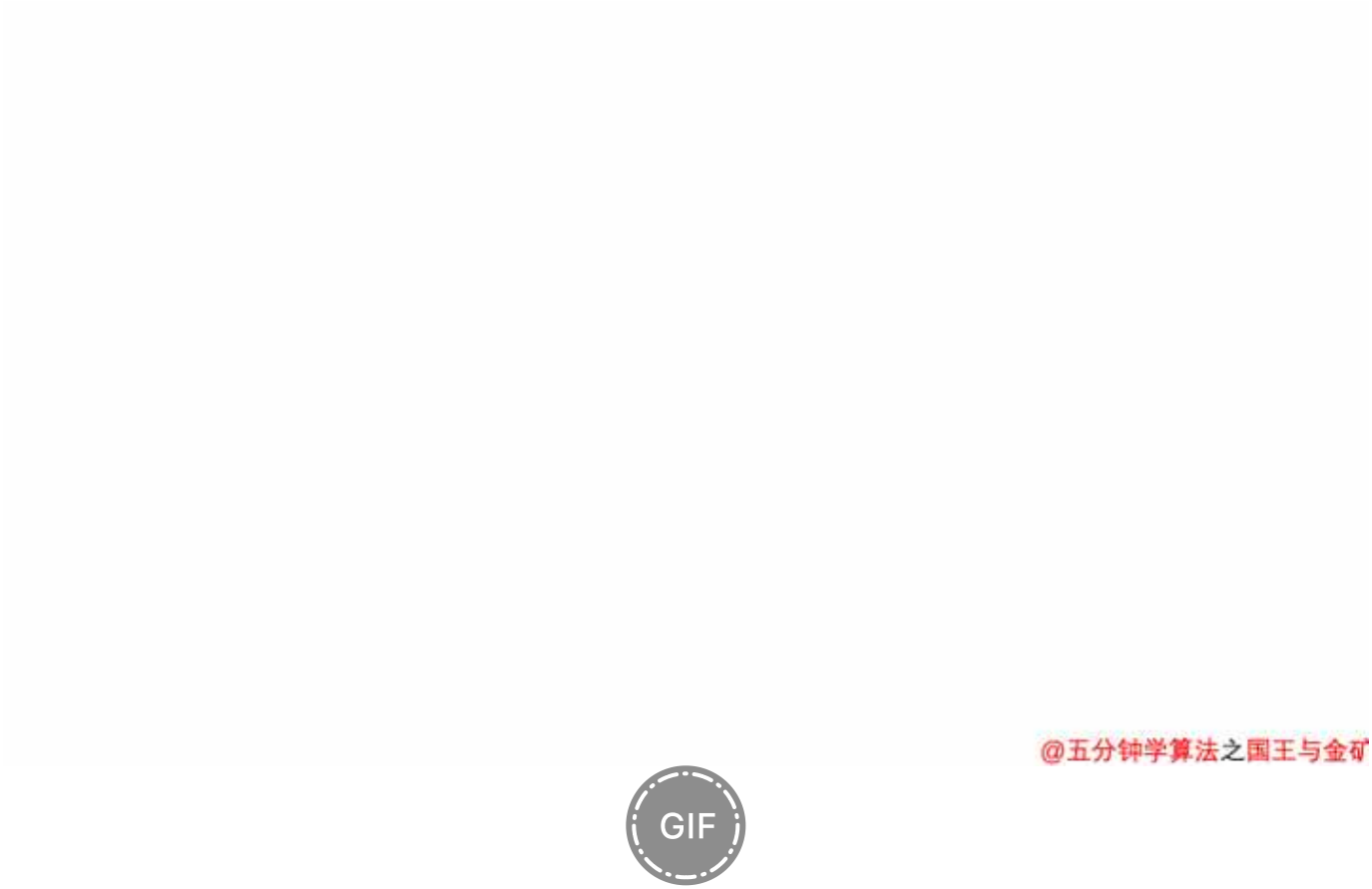
3.挖前三座金矿

这是「国王和金矿」 问题中最重要的一个动画之一，可以多看几遍

GIF

4.挖前四座金矿

这是「国王和金矿」 问题中最重要的一个动画之一，可以多看几遍



国王和金矿问题中的【规律】

仔细观察上面的几组动画可以发现：

对比「挖第一座与第二座金矿」和「挖前三座金矿」，在「挖前三座金矿」中，3 金矿 7 工人的挖矿收益，来自于 2 金矿 7 工人和 2 金矿 4 工人的结果，Max(500,300 + 350) = 650；
对比「挖前三座金矿」和「挖前四座金矿」，在「挖前四座金矿」中，4 金矿 10 工人的挖矿收益，来自于 3 金矿 10 工人和 3 金矿 5 工人的结果，Max(850,400 + 300) = 850；

国王和金矿问题中的【动态规划代码】

```
1代码来源：https://www.cnblogs.com/SDJL/archive/2008/08/22/1274312.html
2
3//maxGold[i][j] 保存了i个人挖前j个金矿能够得到的最大金子数，等于 -1 时表示未知
4int maxGold[max_people][max_n];
5
6int GetMaxGold(int people, int mineNum){
7    int retMaxGold;                              //声明返回的最大金矿数量
8    //如果这个问题曾经计算过
9    if(maxGold[people][mineNum] != -1){
```

```
10        retMaxGold = maxGold[people][mineNum]; //获得保存起来的值
11    }else if(mineNum == 0) {                   //如果仅有一个金矿时 [ 对应动态规划中的"边界
12        if(people >= peopleNeed[mineNum])      //当给出的人数足够开采这座金矿
13            retMaxGold = gold[mineNum];        //得到的最大值就是这座金矿的金子数
14        else                                   //否则这唯一的一座金矿也不能开采
15            retMaxGold = 0;                    //得到的最大值为 0 个金子
16    }else if(people >= peopleNeed[mineNum])    // 如果人够开采这座金矿[对应动态规划中的"最
17    {
18        //考虑开采与不开采两种情况，取最大值
19        retMaxGold = max(
20                        GetMaxGold(people - peopleNeed[mineNum],mineNum - 1) + gold[mine
21                        GetMaxGold(people,mineNum - 1)
22                        );
23    }else//否则给出的人不够开采这座金矿 [ 对应动态规划中的"最优子结构"]
24    {
25        retMaxGold = GetMaxGold(people,mineNum - 1);     //仅考虑不开采的情况
26        maxGold[people][mineNum] = retMaxGold;
27    }
28    return retMaxGold;
29}
```

```
//@五分钟学算法
//maxGold[i][j] 保存了i个人挖前j个金矿能够得到的最大金子数，等于 -1 时表示未知
int maxGold[max_people][max_n];

int GetMaxGold(int people, int mineNum){
    int retMaxGold;                              //声明返回的最大金矿数量
    //如果这个问题曾经计算过
    if(maxGold[people][mineNum] != -1){
        retMaxGold = maxGold[people][mineNum]; //获得保存起来的值
    }else if(mineNum == 0) {                     //如果仅有一个金矿时 [ 对应动态规划中的"边界"]
        if(people >= peopleNeed[mineNum])        //当给出的人数足够开采这座金矿
            retMaxGold = gold[mineNum];          //得到的最大值就是这座金矿的金子数
        else                                     //否则这唯一的一座金矿也不能开采
            retMaxGold = 0;                      //得到的最大值为 0 个金子
    }else if(people >= peopleNeed[mineNum])      // 如果人够开采这座金矿[对应动态规划中的"最优子结构"]
    {
        //考虑开采与不开采两种情况，取最大值
        retMaxGold = max(
                        GetMaxGold(people - peopleNeed[mineNum],mineNum - 1) +
gold[mineNum],
                        GetMaxGold(people,mineNum - 1)
                        );
    }else//否则给出的人不够开采这座金矿 [ 对应动态规划中的"最优子结构"]
    {
        retMaxGold = GetMaxGold(people,mineNum - 1);      //仅考虑不开采的情况
        maxGold[people][mineNum] = retMaxGold;
    }
    return retMaxGold;
}
```


希望通过这篇文章，大家能对「递归」与「动态规划」有一定的理解。后续将以「动态规划」为基础研究多重背包算法、迪杰特斯拉算法等更高深的算法问题，同时「递归」的更多概念也会在「分治算法」章节再次延伸，敬请对程序员小吴保持关注：）

编辑于 2019-01-10

Huper

努力不一定成功，但不努力一定很轻松。

13 人赞同了该回答

教材上在讲解动态规划的意义时，基本上都是给出高度概括的两点：最优子结构和子问题重叠。这两点说的挺好的，但是最好还是换个方法理解。

首先要有这样一个概念：计算机中的大多数问题（算法），实际上都可以理解成搜索问题。在某个特定的解空间中搜索出满足条件的解。（所谓的机器学习和深度学习算法也大都如此，解空间是庞大的参数

组合域，我们根据优化目标去搜索最优或者近似最优的参数解）

一旦把问题转换成搜索问题，我们需要解决的难题就是如何优化搜索问题，而动态规划在我看来本质就是一个优化搜索的技巧，和剪枝一样，只不过剪枝的核心是：不去搜索一看就不包含最优解的解空间，而动态规划说：不去搜索已经搜索过的解空间，已经搜索过的解空间的解我保存下来，可以以O(1)的速度获取，而且没有搜索过的解空间的解是依赖于已经搜索过的解空间的。

所以处理动态规划问题有两个要点，一个是保存已搜索过解空间的解，一般用数组或者hashmap存放保证O(1)获取速度。另一个是构造问题规模，因为大规模问题的解是依赖小规模问题的解的，所以我们要先构造小规模问题求解，然后构造大规模问题求解。

感觉自己说得好乱。。。。

发布于 2019-09-12

小耿
生物狗转职程序猿

23 人赞同了该回答

所谓动态规划就是站起来一边走动一边规划，有效预防颈椎病……

不开玩笑，给一个“加了辣椒的菜都叫做川菜”式的回答吧（正确解释还是应该看上面排1、2、4的答案）：

动态规划，就是把计算过程中产生的中间结果静态地保存起来，后面再用到这些中间结果的时候就不用再算一遍，而是直接读取。

那明明是静态规划为什么要叫动态？
其实据发明这个名字的Bellman教授讲：当时他所在的公司给美国军方打工，军方管他们的头头是个大老粗，属于那种“对‘研究’、‘数学’之类的词汇有病理性的恐惧与憎恶”的那种人，于是机智的Bellman决定不告诉他自己在公司里是研究数学的。他给自己的工作安了个名称叫“动态编程dynamic programming”（翻译成中文时翻成了规划）。为什么叫“动态（dynamic）”呢？Bellman说，除了很恰当地描绘了自己的工作以外，“动态”还是一个神奇的词：它不可能被应用于贬义的语境！你试试用它造个含贬义的句子，造不出来吧？这是一个连政客都挑不出刺的词，所以Bellman用它给自己的研究打掩护。

编辑于 2015-03-20

doodlewind
花名雪碧 | github.com/doodlewind

101 人赞同了该回答

猿爸爸把 1+1+1+1+1+1+1+1 = 写在纸上，问小猿（咦）：
「它们加起来是多少哇？」

（数了一会...）「8 ！」

猿爸爸在左边又加了个 1+，再问一次小猿：
「现在呢？」

（迅速地）「9 ！」

「为什么小猿这么快就知道了呢？」

「因为你刚刚加了 1 啊~」

「所以只要记得之前的结果，就不用再计一次数啦。」

嗯，动态规划就是一种「先记住些事，方便后面节约时间」的神奇方法。

但是如果你觉得这个答案对你的胃口，那么恭喜你，你对 DP 的理解成功地达到了四岁小孩的水平……
FYI:
How should I explain dynamic programming to a 4-year-old?
不谢

发布于 2014-12-20

▲赞同 101 ▼ 8 条评论 分享 收藏 喜欢 ···

李陶冶
Noip普及之路漫漫其修远兮，51nod将上下而求索

54 人赞同了该回答

我在思考这个问题的意义，这里面有个残酷的事实。
很多人在学习的时候都希望先了解整个知识结构，再去研究具体的问题。但把这种方法套用在DP上，会发现行不通。因为DP的概念比DP的问题更难理解。能在这里解释DP概念的，往往都解决了几十个或上百个DP的问题。
如果lz的提问是为了学习DP的话，我建议你先去找些DP的问题来解决，比如：动态规划的问题，从简单到难，做完第1页应该就能自己总结出个梗概了。

发布于 2014-12-18

▲赞同 54 ▼ 15 条评论 分享 收藏 喜欢 ···

naiveman
我祝你好运

21 人赞同了该回答

MIT 6.006 19-22讲及习题课
youtube.com/watch?
MIT 6.046j 15讲
网易公开课

讲的挺好的。还给了好多习题的例子。这里的好多答案都和这个课上讲的不谋而合。
还觉得不够可以再翻翻 《挑战程序设计竞赛 第二版 》的例子

我觉得最重要的前提就是这个问题确实能用dp来解决，也就是原问题的解可以通过子问题来得到。应该是这个意思吧。。。

感兴趣的话还可以看看这个讲义的49-57页，讲怎么并行的处理dp问题。
supertech.csail.mit.edu

发布于 2017-02-07

▲赞同 21 ▼ 添加评论 分享 收藏 喜欢 ···

CHEN CHEN
EECS, OR, and ECON.

11 人赞同了该回答

找到了一个好的拓扑排序，聪明的穷举法？

发布于 2015-04-04

▲赞同 11 ▼ 3 条评论 分享 收藏 喜欢 ···

Fan You
Operations Research

4 人赞同了该回答

> ... the essence of dynamic programming: Multistage problems may be solved by analyzing a sequence of simpler inductively defined single-stage problems.

Puterman, Martin L.*Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons, 2014.

发布于 2019-06-19

▲赞同 4 ▼ 1 条评论 分享 收藏 喜欢 ···

冒泡
IT为主，兴趣广泛，爱好强答

2 人赞同了该回答

如何理解动态规划？ - 冒泡的回答 - 知乎

编辑于 2017-02-08

▲赞同 2 ▼ 添加评论 分享 收藏 喜欢 ···

Xiaohu Zhu
A watchful guardian for AGI.

11 人赞同了该回答

最近在看强化学习，刚好是强相关的。我就补一个追根溯源的粗略的答案吧。
动态规划最早还是 Bellman 用在最优控制理论上研究问题中的设计的优化方法。后来被抽象再简化又放入了算法框架中，便成了我们现在在算法课程中常常看到的 DP。在实际使用中，比如说规划 planning 中，动态规划实际上依赖于 tabular，就是需要把所有可能的状态和行动组合建个表存起来，然后对于状态空间非常大的实际问题常常出现维度灾难。后来在强化学习中被大家不断地发展，现在是一个丰富的研究领域，另外还有个名字叫做 Approximate DP。前段时间大热的 AlphaGo 和 DQN 也可以看做是 DP 的应用啦。

编辑于 2016-09-07

▲赞同 11 ▼ 1 条评论 分享 收藏 喜欢 ···

鱼鲲
中国科学院 保福寺分院 小僧

57 人赞同了该回答

我想，大家对动态规划的**困惑在于拿到一道题目，告诉你是动态规划，能很好的写出来；但自己独立分析问题特征，却很难判断出这道题目需要用动态规划来求解。**

所以，冒昧的揣测一下，TZ疑惑的**不是动态规划是什么，而是为何需要使用动态规划DP来求解问题。**

1) 先来回到第一个问题，**“动态规划是什么”？**
上面的答案基本上已经说的很好了：**动态规划是递归，是缓存，是用空间来换取时间**；但是，如果仅仅知道这些，你还是发现无法设计动态规划的算法。因为你慢慢会发现，**有些问题用动态规划和递归都能求解，但是动态规划的速度会更慢。于是有人说了，动态规划题目的特征在于最优子结构和重叠子问题——这就涉及到下一个问题**

2)为什么需要使用动态规划？
在初等算法中，算法设计的思路一般如下，首先尝试穷举法；然而如何**穷举**？
此时往往要用到**分治法**——而**归递**，在绝大多数时候仅仅是分治法的一种表现形式而已；
在递归和分治法的基础上，往往会用**动态规划**来优化——动态规划，实际上是一种升级版的分治法。
当然，**不是所有的穷举都能使用分治法；不是所有的分治法都能优化成动态规划。此时，就是上文提到的:只有一个问题是可分的，才可以使用分治法；只有分治出来的子问题有重叠，才可以使用DP；只有子问题具有最优子结构，DP才具有意义。**

当然，其中最难的部分可能在于对于一个没有经验的程序设计者来说，判断重叠子问题和最优子结构是不容易的，这部分就不是用文字能够使人明白的了。需要结合实例，分析才行：
动态规划分析总结——如何设计和实现动态规划算法

编辑于 2014-07-19

▲赞同 57 ▼ 6 条评论 分享 收藏 喜欢 ···

lonelycorn
制造类专业很难在大城市立足……

9 人赞同了该回答

知乎上程序员比较多，我来提供一个optimization角度的观点吧
对于某dynamic process，如果定义了additive cost function，且allowable actions仅由当前的状态决定，那么便满足了bellman equation的条件，可以使用动态规划方法求解。
需要注意的是，这里的programming不是写程序的意思。dp并不只是用在离散数学中。比如对于optimal control，解bellman equation可以得到最优控制器。

发布于 2014-12-19

▲赞同 9 ▼ 添加评论 分享 收藏 喜欢 ···

九章算法
美帝代码搬运工，资深面试官，微信公众号-九章算法

51 人赞同了该回答

首先，我要回答你的第一个问题：**什么是动态规划？**

动态规划是一种**通过“大而化小”的思路解决问题**的算法。区别于一些固定形式的算法，如二分法，宽度优先搜索法，动态规划没有实际的步骤来规定第一步做什么第二步做什么。所以更加确切的说，动态规划是一种解决问题的思想。这种思想的本质是，一个规模比较大的问题（假如用2-3个参数可以表示），是通过规模比较小的若干问题的结果来得到的（通过取最大，取最小，或者加起来之类的运算）
所以我们经常看到的动态规划的核心——状态转移方程都长成这样：

* f[i][j] = f[i - 1][j] + f[i][j - 1]

* f[i] = max{f[j] if j < i and ...} + 1

* f[i][j] = f[0][j - 1] && judge(1,i) || f[1][j - 1] && judge(2,i) || ...

现在我要回答你的第二个问题了：**动态规划的意义是什么？**

动态规划一般来说是**“高效”**的代名词，因为其解决的问题一般退而求其次的算法只有搜索了。以“数字三角形”一题为例子（lintcode.com/problem/tr ），在“三角矩阵”中找一条从上到下的路径，使得权值之和最小。如果使用暴力搜索的算法，那么需求穷举出2^(n-1)条路径（n为三角形高度），而使用动态规划的话，则时间复杂度降低到了n^2，完成了质的飞跃。那么究竟为什么这么快呢？原因在于动态规划算法去掉了“无用和重复的运算”。在搜索算法中，假如从A->B有2条路径，一条代价为10，另外一条代价为100，B->终点有1024条路径。当我们选择了代价为10的那条路径走到B时，可以继续往下走完1024条路径到终点，但是在此之后，我们再从代价为100的路径从A走到B时，我们可以发现此时无论如何走，都不可能有刚才从10的路径走过来更好，所以这些计算是“无用”的计算，也可以说是“重复”的计算。这就是动态规划之所以“快”的重要原因。

> 一般求最大值/最小值、求可不可行、求方案总数90%的概率是使用动态规划来求解。要重点说明的是，如果一个问题让你求出“所有的”方案和结果，则肯定不是使用动态规划。

解决一个动态规划问题首先根据“问5”判断是否是动态规划的问题，如果是，则尝试将其按照“问4”进行分类，找到对应的类别和相似的问题。接着从下面的4个要素去逐步剖析解决这道题：

> 1. 状态是什么
> 2. 状态转移方程是什么
> 3. 状态的初始值是什么
> 4. 问题要求的最后答案是什么

每个步骤分析完成之后，就基本上解决了整道动态规划的问题。

现在动态规划在面试中考得很多，并且越来越多了。随着CS从业与求职者的增加，并伴随大家都是“有备而来”的情况下，一般简单的反转链表之类的题目已经无法再在面试中坚挺了。因此在求职者人数与招聘名额的比例较大的情况下，公司会倾向于出更难的面试问题。而动态规划就是一种比较具有难度，又比较“好出”的面试问题。相比其他的算法与数据结构知识来说，贪心法分治法太难出题了，搜索算法往往需要耗费求职者过长的程序编写时间一般也不倾向于出，二叉树链表等问题题目并没有那么多，而且求职者也都会着重准备这一块。因此动态规划这一类的问题，便越来越多的出现在了面试中。

动态规划的常见类型分为如下几种：

> * 矩阵型
> * 序列型
> * 双序列型
> * 划分型
> * 区间型
> * 背包型
> * 状态压缩型
> * 树型

其中，在**技术面试中经常出现的是矩阵型，序列型和双序列型**。划分型，区间型和背包型偶尔出现。状态压缩和树型基本不会出现（一般在算法竞赛中才会出现）。

每种类型都有着自己的题目特点和状态的表示方法。以矩阵型动态规划为例，一般题目会给你一个矩阵，告诉你有一个小人在上面走动，每次只能向右和向下走，然后问你比如有多少种方案从左上走到右下 (lintcode.com/problem/un)。这种类型状态表示的特点一般是使用坐标作为状态，如f[i][j]表示走到(i,j)这个位置的时候，一共有多少种方案。状态的转移则是考虑是从哪儿走到(i,j)这个坐标的。而序列型的动态规划，一般是告诉你一个序列；双序列的动态规划一般是告诉你两个字符串或者两个序列。

将所做过的动态规划问题按照这些类别进行归类，分析状态的表示方法和状态转移方程的构造方法在每种类型中的近似之处，会让你更快的学会动态规划。

有什么书籍和参考资料可以推荐么？

著名的《背包九讲》：

九章算法《动态规划专题班》，有免费试听，含括所有面试的高频动态规划考点。

有哪些动态规划题目必须要练习的？

在LintCode上包含了30余道动态规划的练习题，都是从实际的面试问题中汇总的精选练习：

lintcode.com/tag/dynami

编辑于 2018-09-08

王星泽
Mens et Manus

78 人赞同了该回答

目前的答案其实已经很好了，只不过还是没有深入到最深刻的本质。动态规划的最根本的本质非常简单，而且不局限于计算机算法领域。（目前最高票的答案讲得很好，但是局限于离散问题和finite

horizon；其他的答案也都从不同角度讲了动态规划作为一个计算机算法在实际应用上的一些考虑。）

动态规划是一种思维方法，整个学科的基本思想就是一条，动态规划之父Bellman的Principle of Optimality (翻译成“最优化原理”？)：

设想你想要采用最优的策略解决某件事，而且这件事可以分成好多步；假设你已经知道了做这件事的整体上最优的策略；再假设你根据这个整体最优策略走了几步，接下来剩下的几步的你重新算了一个最优子策略，如果和整体最优策略在接下来这几步的子策略相符合，那么这件事符合最优化原理；然后就可以使用动态规划的算法解决，解决思路就是一步步找出这些最优的子策略，最后得到整体最优策略。

举个例子：我想走最短的距离开车从(A)三藩去(B)纽约。用地图软件一查得到了下图的路径，这就是上面说的整体最优策略。我现在根据这个整体最优的路径从三藩开了两天到了芝加哥，然后我重新用地图软件查从芝加哥到纽约的最短路径，发现和之前查到的完全重合！

（虽然第一次查的时候用到了三藩，第二次完全没用三藩。这个发现，是不是很神奇。。。如果你觉得trivial，说明你生活经验丰富或者脑子转得快，但是我请你再想想，可不是所有问题都符合这个特点。。。比如，学过马尔科夫的童鞋，你懂的。）

这就是一个符合最优化原理、从而可以被动态规划解决的问题。

怎么解决呢？假设我们没有这个强大的地图软件，而人工地使用动态规划算法，那么步骤可以是这样的：假设所有的路径必须经过某些驿站，首先我找到从三藩出发到每一个第一站驿站的路径和距离，接下来计算从三藩出发到每一个第二站驿站的最短路径（分别经过某一个相同或者不同的第一站驿站），然后计算第三站（这时候只用知道到第二站的最短路径）、第四站（这时候只用知道到第三站的最短路径）、最后终于算到纽约，我就知道了从三藩出发到纽约的最短路径，于是就可以按照这个最短路径多快好省地出发了。。。

（上图窃取自斯坦福大学Wen Zheng老师《动态规划》课件。）

如果关于上面这些话，你能说服自己，那么恭喜你，你已经理解了动态规划的灵魂。。。

动态规划的理论意义也只有一个，就是最优化，哪怕你要解决的问题是随机和/或混乱的。（实际意义在不同学科或应用，比如动态资产定价、计算机算法等，可以很不一样；也有许多丰富的内涵，目前的答案讲得很好，就不班门弄斧了。）理论上虽然动态规划可以解决符合最优化原理的问题，实际上由于curse of dimensionality等原因有很大的局限性。

最后说明一下：严格理解动态规划需要学习测度论，以上试图把动态规划这个优美的思想用最容易理解的方式表达（连大自然都在使用动态规划的算法，因为大自然总是想最优化能量的分配，感兴趣的可以去看看HJB方程）。

编辑于 2018-03-29