原 算法-动态规划 Dynamic Programming--从菜鸟到老鸟

2017年07月15日 22:58:29 HankingHu 阅读数：53178 标签： 动态规划 算法
更多

前言

最近在牛客网上做了几套公司的真题，发现有关动态规划（Dynamic Programming）算法的题目很多。相对于我来说，算法里面遇到的问题里面感觉最难的也就是动态规划（Dynamic Programming）算法了，于是花了好长时间，查找了相关的文献和资料准备彻底的理解动态规划（Dynamic Programming）算法。一是帮助自己总结知识点，二是也能够帮助他人更好的理解这个算法。后面的参考文献只是我看到的文献的一部分。

动态规划算法的核心

理解一个算法就要理解一个算法的核心，动态规划算法的核心是下面的一张图片和一个小故事。


```
1  A * "1+1+1+1+1+1+1+1 =? " *
2
3  A : "上面等式的值是多少"
4  B : *计算* "8!"
```

```
5
6  A *在上面等式的左边写上 "1+" *
7  A : "此时等式的值为多少"
8  B : *quickly* "9!"
9  A : "你怎么这么快就知道答案了"
10 A : "只要在8的基础上加1就行了"
11 A : "所以你不用重新计算因为你记住了第一个等式的值为8!动态规划算法也可以说是 '记住求过的解来节省时间'"
```

由上面的图片和小故事可以知道动态规划算法的核心就是记住已经解决过的子问题的解。

动态规划算法的两种形式

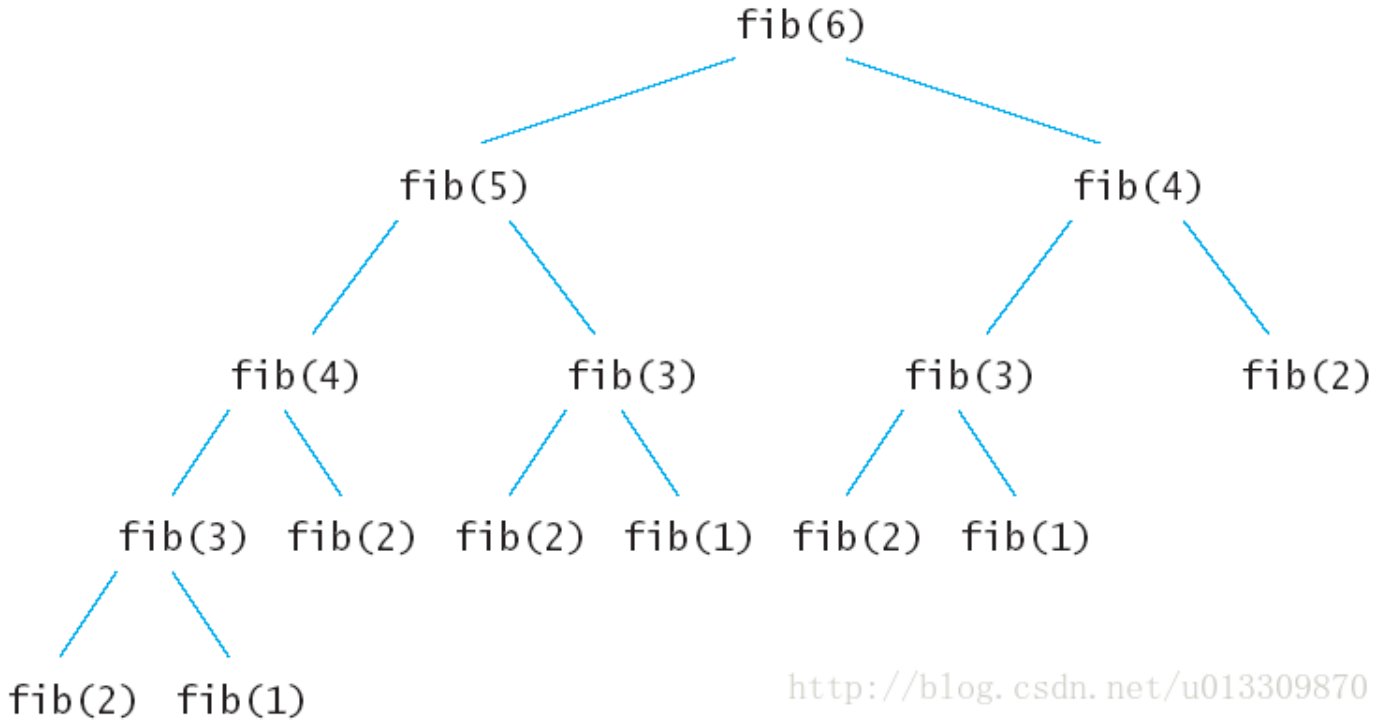
上面已经知道动态规划算法的核心是记住已经求过的解，记住求解的方式有两种：①**自顶向下的备忘录法** ②**自底向上**。
为了说明动态规划的这两种方法，举一个最简单的例子：求斐波拉契数列**Fibonacci** 。先看一下这个问题：

```
1  Fibonacci (n) = 1;   n = 0
2
3  Fibonacci (n) = 1;   n = 1
4
5  Fibonacci (n) = Fibonacci(n-1) + Fibonacci(n-2)
```

以前学c语言的时候写过这个算法使用递归十分的简单。先使用递归版本来实现这个算法：

```
1  public int fib(int n)
2  {
3      if(n<=0)
4          return 0;
5      if(n==1)
6          return 1;
7      return fib( n-1)+fib(n-2);
8  }
9  //输入6
10 //输出：8
```

先来分析一下递归算法的执行流程，假如输入6，那么执行的递归树如下：



上面的递归树中的每一个子节点都会执行一次，很多重复的节点被执行，fib(2)被重复执行了5次。由于调用每一个函数的时候都要保留上下文，所以空间上开销也不小。这么多的子节点被重复执行，如果在执行的时候把执行过的子节点保存起来，后面要用到的时候直接查表调用的话可以节约大量的时间。下面就看看动态规划的两种方法怎样来解决斐波拉契数列**Fibonacci**数列问题。

①自顶向下的备忘录法


```
1  public static int Fibonacci(int n)
2  {
3          if(n<=0)
4              return n;
5          int []Memo=new int[n+1];
6          for(int i=0;i<=n;i++)
7              Memo[i]=-1;
8          return fib(n, Memo);
9      }
10     public static int fib(int n,int []Memo)
11     {
12
13         if(Memo[n]!=-1)
14             return Memo[n];
15     //如果已经求出了fib（n）的值直接返回，否则将求出的值保存在Memo备忘录中。
```

```
16
17          if(n<=2)
18              Memo[n]=1;
19
20          else Memo[n]=fib( n-1,Memo)+fib(n-2,Memo);
21
22          return Memo[n];
        }
```

备忘录法也是比较好理解的，创建了一个n+1大小的数组来保存求出的斐波拉契数列中的每一个值，在递归的时候如果发现前面fib（n）的值计算出来了就不再计算，如果未计算出来，则计算出来后保存在Memo数组中，下次在调用fib（n）的时候就不会重新递归了。比如上面的递归树中在计算fib（6）的时候先计算fib（5），调用fib（5）算出了fib（4）后，fib（6）再调用fib（4）就不会在递归fib（4）的子树了，因为fib（4）的值已经保存在Memo[4]中。

②自底向上的动态规划

备忘录法还是利用了递归，上面算法不管怎样，计算fib（6）的时候最后还是要计算出fib（1），fib（2），fib（3）……,那么何不先计算出fib（1），fib（2），fib（3）……,呢？这也就是动态规划的核心，先计算子问题，再由子问题计算父问题。

```
1  public static int fib(int n)
2  {
3          if(n<=0)
4              return n;
5          int []Memo=new int[n+1];
6          Memo[0]=0;
7          Memo[1]=1;
8          for(int i=2;i<=n;i++)
9          {
10             Memo[i]=Memo[i-1]+Memo[i-2];
11         }
12         return Memo[n];
13 }
```

自底向上方法也是利用数组保存了先计算的值，为后面的调用服务。观察参与循环的只有 i，i-1 , i-2三项，因此该方法的空间可以进一步的压缩如下。

```
1  public static int fib(int n)
2      {
```

```
3           if(n<=1)
4               return n;
5
6           int Memo_i_2=0;
7           int Memo_i_1=1;
8           int Memo_i=1;
9           for(int i=2;i<=n;i++)
10          {
11              Memo_i=Memo_i_2+Memo_i_1;
12              Memo_i_2=Memo_i_1;
13              Memo_i_1=Memo_i;
14          }
15          return Memo_i;
16      }
```

一般来说由于备忘录方式的动态规划方法使用了递归，递归的时候会产生额外的开销，使用自底向上的动态规划方法要比备忘录方法好。
你以为看懂了上面的例子就懂得了动态规划吗？那就too young too simple了。动态规划远远不止如此简单，下面先给出一个例子看看能否独立完成。然后再对动态规划的其他特性进行分析。

动态规划小试牛刀

例题：钢条切割

假定我们知道 Serling 公司出售一段长度为 i 英寸的钢条的价格为 p_i（$i=1, 2, \cdots$，单位为美元）。钢条的长度均为整英寸。图 15-1 给出了一个价格表的样例。

长度 i	1	2	3	4	5	6	7	8	9	10
价格 p_i	1	5	8	9	10	17	17	20	24	30

图 15-1　钢条价格表样例。每段长度为 i 英寸的钢条为公司带来 p_i 美元的收益

钢条切割问题是这样的：给定一段长度为 n 英寸的钢条和一个价格表 p_i（$i=1, 2, \cdots, n$），求切割钢条方案，使得销售收益 r_n 最大。注意，如果长度为 n 英寸的钢条的价格 p_n 足够大，最优解可能就是完全不需要切割。

我们发现，将一段长度为 4 英寸的钢条切割为两段各长 2 英寸的钢条，将产生 $p_2+p_2=5+5=10$ 的收益，为最优解。

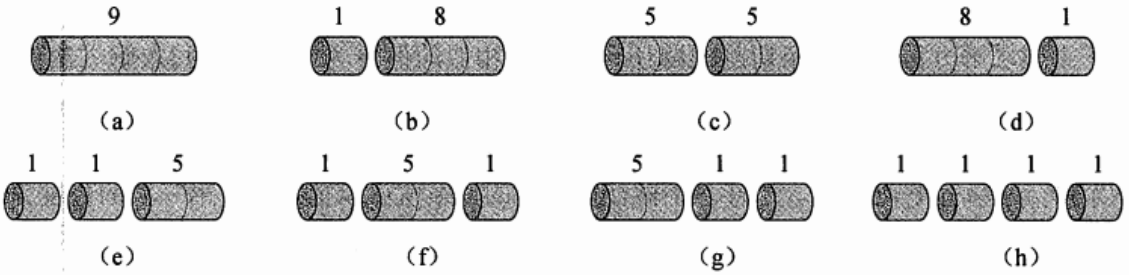


图 15-2 4 英寸钢条的 8 种切割方案。根据图 15-1 中的价格表，在每段钢条之上标记了它的价格。最优策略为方案(c)——将钢条切割为两段长度均为 2 英寸的钢条——总价值为 10

更一般地，对于 $r_n(n\geqslant 1)$，我们可以用更短的钢条的最优切割收益来描述它：

$$r_n=\max(p_n,r_1+r_{n-1},r_2+r_{n-2},\cdots,r_{n-1}+r_1) \tag{15.1}$$

第一个参数 p_n 对应不切割，直接出售长度为 n 英寸的钢条的方案。其他 $n-1$ 个参数对应另外 $n-1$ 种方案：对每个 $i=1,2,\cdots,n-1$，首先将钢条切割为长度为 i 和 $n-i$ 的两段，接着求解这两段的最优切割收益 r_i 和 r_{n-i}(每种方案的最优收益为两段的最优收益之和)。由于无法预知哪种方案会获得最优收益，我们必须考察所有可能的 i，选取其中收益最大者。如果直接出售原钢条会获得最大收益，我们当然可以选择不做任何切割。

关子问题的最优解，并在所有可能的两段切割方案中选取组合收益最大者，构成原问题的最优解。我们称钢条切割问题满足**最优子结构**(optimal substructure)性质：问题的最优解由相关子问题的最优解组合而成，而这些子问题可以独立求解。

除了上述求解方法外，钢条切割问题还存在一种相似的但更为简单的递归求解方法：我们将钢条从左边切割下长度为 i 的一段，只对右边剩下的长度为 $n-i$ 的一段继续进行切割(递归求解)，对左边的一段则不再进行切割。即问题分解的方式为：将长度为 n 的钢条分解为左边开始一段，以及剩余部分继续分解的结果。这样，不做任何切割的方案就可以描述为：第一段的长度为 n，收益为 p_n，剩余部分长度为 0，对应的收益为 $r_0=0$。于是我们可以得到公式(15.1)的简化版本：

$$r_n=\max_{1\leqslant i\leqslant n}(p_i+r_{n-i}) \tag{15.2}$$

上面的例题来自于**算法导论**

关于题目的讲解就直接截图算法导论书上了这里就不展开讲。现在使用一下前面讲到三种方法来来实现一下。

①递归版本

```
1  public static int cut(int []p,int n)
2      {
```

```
        if(n==0)
            return 0;
        int q=Integer.MIN_VALUE;
        for(int i=1;i<=n;i++)
        {
            q=Math.max(q, p[i-1]+cut(p, n-i));
        }
        return q;
    }
```

递归很好理解，如果不懂可以看上面的讲解，递归的思路其实和回溯法是一样的，遍历所有解空间但这里和上面斐波拉契数列的不同之处在于，在每一层上都进行了一次最优解的选择，q=Math.max(q, p[i-1]+cut(p, n-i));这个段语句就是最优解选择，这里上一层的最优解与下一层的最优解相关。

②备忘录版本

```
public static int cutMemo(int []p)
    {
        int []r=new int[p.length+1];
        for(int i=0;i<=p.length;i++)
            r[i]=-1;
        return cut(p, p.length, r);
    }
    public static int cut(int []p,int n,int []r)
    {
        int q=-1;
        if(r[n]>=0)
            return r[n];
        if(n==0)
            q=0;
        else {
            for(int i=1;i<=n;i++)
                q=Math.max(q, cut(p, n-i,r)+p[i-1]);
        }
        r[n]=q;

        return q;
    }
```

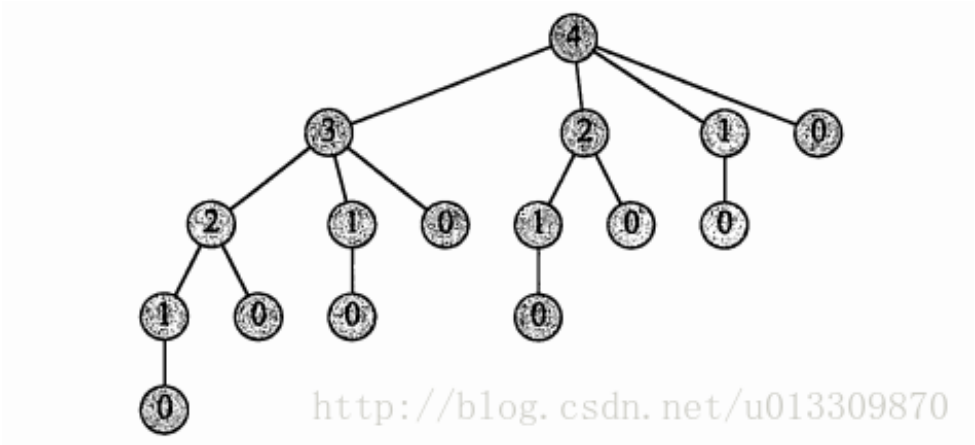
有了上面求斐波拉契数列的基础，理解备忘录方法也就不难了。备忘录方法无非是在递归的时

候记录下已经调用过的子函数的值。这道钢条切割问题的经典之处在于自底向上的动态规划问题的处理，理解了这个也就理解了动态规划的精髓。

③自底向上的动态规划

```
1  public static int buttom_up_cut(int []p)
2      {
3          int []r=new int[p.length+1];
4          for(int i=1;i<=p.length;i++)
5          {
6              int q=-1;
7              //①
8              for(int j=1;j<=i;j++)
9                  q=Math.max(q, p[j-1]+r[i-j]);
10             r[i]=q;
11         }
12         return r[p.length];
13     }
```

自底向上的动态规划问题中最重要的是理解注释①处的循环，这里外面的循环是求r[1],r[2]……，里面的循环是求出r[1],r[2]……的最优解，也就是说r[i]中保存的是钢条长度为i时划分的最优解，这里面涉及到了最优子结构问题，也就是一个问题取最优解的时候，它的子问题也一定要取得最优解。下面是长度为4的钢条划分的结构图。我就偷懒截了个图。



动态规划原理

虽然已经用动态规划方法解决了上面两个问题，但是大家可能还跟我一样并不知道什么时候要用到动态规划。总结一下上面的斐波拉契数列和钢条切割问题，发现两个问题都涉及到了重叠

子问题，和最优子结构。

①最优子结构

用动态规划求解最优化问题的第一步就是刻画最优解的结构，如果一个问题的解结构包含其子问题的最优解，就称此问题具有最优子结构性质。因此，某个问题是否适合应用动态规划算法，它是否具有最优子结构性质是一个很好的线索。使用动态规划算法时，用子问题的最优解来构造原问题的最优解。因此必须考查最优解中用到的所有子问题。

---

②重叠子问题

在斐波拉契数列和钢条切割结构图中，可以看到大量的重叠子问题，比如说在求fib（6）的时候，fib（2）被调用了5次，在求cut（4）的时候cut（0）被调用了4次。如果使用递归算法的时候会反复的求解相同的子问题，不停的调用函数，而不是生成新的子问题。如果递归算法反复求解相同的子问题，就称为具有重叠子问题（overlapping subproblems）性质。在动态规划算法中使用数组来保存子问题的解，这样子问题多次求解的时候可以直接查表不用调用函数递归。

动态规划的经典模型

线性模型

线性模型的是动态规划中最常用的模型，上文讲到的钢条切割问题就是经典的线性模型，这里的线性指的是状态的排布是呈线性的。【例题1】是一个经典的面试题，我们将它作为线性模型的敲门砖。

【例题1】在一个夜黑风高的晚上，有n（n <= 50）个小朋友在桥的这边，现在他们需要过桥，但是由于桥很窄，每次只允许不大于两人通过，他们只有一个手电筒，所以每次过桥的两个人需要把手电筒带回来，i号小朋友过桥的时间为T[i]，两个人过桥的总时间为二者中时间长者。问所有小朋友过桥的总时间最短是多少。

每次过桥的时候最多两个人，如果桥这边还有人，那么还得回来一个人（送手电筒），也就是说N个人过桥的次数为2*N-3（倒推，当桥这边只剩两个人时只需要一次，三个人的情况为来回一次后加上两个人的情况...）。有一个人需要来回跑，将手电筒送回来（也许不是同一个人，realy？！）这个回来的时间是没办法省去的，并且回来的次数也是确定的，为N-2，如果是我，我会选择让跑的最快的人来干这件事情，但是我错了...如果总是跑得最快的人跑回来的话，那么他在每次别人过桥的时候一定得跟过去，于是就变成就是很简单的问题了，花费的总时间：

T = minPTime * (N-2) + (totalSum-minPTime)

来看一组数据 四个人过桥花费的时间分别为 1 2 5 10，按照上面的公式答案是19，但是实际答案应该是17。

具体步骤是这样的：

第一步：1和2过去，花费时间2，然后1回来（花费时间1）；

第二步：3和4过去，花费时间10，然后2回来（花费时间2）；

第三部：1和2过去，花费时间2，总耗时17。

所以之前的贪心想法是不对的。我们先将所有人按花费时间递增进行排序，假设前i个人过河花费的最少时间为opt[i]，那么考虑前i-1个人过河的情况，即河这边还有1个人，河那边有i-1个人，并且这时候手电筒肯定在对岸，所以opt[i] = opt[i-1] + a[1] + a[i] (让花费时间最少的人把手电筒送过来，然后和第i个人一起过河)如果河这边还有两个人，一个是第i号，另外一个无所谓，河那边有i-2个人，并且手电筒肯定在对岸，所以opt[i] = opt[i-2] + a[1] + a[i] + 2*a[2] (让花费时间最少的人把电筒送过来，然后第i个人和另外一个人一起过河，由于花费时间最少的人在这边，所以下一次送手电筒过来的一定是花费次少的，送过来后花费最少的和花费次少的一

起过河，解决问题）
所以 opt[i] = min{opt[i-1] + a[1] + a[i] , opt[i-2] + a[1] + a[i] + 2*a[2] }

区间模型

区间模型的状态表示一般为d[i][j]，表示区间[i, j]上的最优解，然后通过状态转移计算出[i+1, j]或者[i, j+1]上的最优解，逐步扩大区间的范围，最终求得[1, len]的最优解。

【例题2】给定一个长度为n（n <= 1000）的字符串A，求插入最少多少个字符使得它变成一个回文串。
典型的区间模型，回文串拥有很明显的子结构特征，即当字符串X是一个回文串时，在X两边各添加一个字符’a’后，aXa仍然是一个回文串，我们用d[i][j]来表示A[i...j]这个子串变成回文串所需要添加的最少的字符数，那么对于A[i] == A[j]的情况，很明显有 d[i][j] = d[i+1][j-1] （这里需要明确一点，当i+1 > j-1时也是有意义的，它代表的是空串，空串也是一个回文串，所以这种情况下d[i+1][j-1] = 0）；当A[i] != A[j]时，我们将它变成更小的子问题求解，我们有两种决策：

1、在A[j]后面添加一个字符A[i]；

2、在A[i]前面添加一个字符A[j]；

根据两种决策列出状态转移方程为：

d[i][j] = min{ d[i+1][j], d[i][j-1] } + 1;（每次状态转移，区间长度增加1）

空间复杂度O(n^2)，时间复杂度O(n^2)， 下文会提到将空间复杂度降为O(n)的优化算法。

背包模型

背包问题是动态规划中一个最典型的问题之一。由于网上有非常详尽的背包讲解，这里只将常用部分抽出来。

【例题3】有N种物品（每种物品1件）和一个容量为V的背包。放入第 i 种物品耗费的空间是Ci，得到的价值是Wi。求解将哪些物品装入背包可使价值总和最大。f[i][v]表示前i种物品恰好放入一个容量为v的背包可以获得的最大价值。决策为第i个物品在前i-1个物品放置完毕后，是选择放还是不放，状态转移方程为：

f[i][v] = max{ f[i-1][v], f[i-1][v – Ci] +Wi }

时间复杂度O(VN)，空间复杂度O(VN)（空间复杂度可利用滚动数组进行优化达到O(V)）。

动态规划题集整理

1、最长单调子序列
Constructing Roads In JG Kingdom★★☆☆☆
Stock Exchange ★★☆☆☆

2、最大M子段和
Max Sum ★☆☆☆☆
最长公共子串 ★★☆☆☆

3、线性模型
Skiing ★☆☆☆☆

总结

弄懂动态规划问题的基本原理和动态规划问题的几个常见的模型，对于解决大部分的问题已经足够了。希望能对大家有所帮助，转载请标明出处http://write.blog.csdn.net/mdeditor#!postId=75193592，创作实在不容易，这篇博客花了我将近一个星期的时间。

参考文献

1.算法导论