

讲堂 > 数据结构与算法之美 > 文章详情

40 | 初识动态规划：如何巧妙解决“双十一”购物时的凑单问题？

2018-12-26 王争



40 | 初识动态规划：如何巧妙解决“双十一”购物时的凑单问题？

朗读人：修阳 16'25" | 15.04M

淘宝的“双十一”购物节有各种促销活动，比如“满 200 元减 50 元”。假设你女朋友的购物车中有 n 个 ($n > 100$) 想买的商品，她希望从里面选几个，在凑够满减条件的前提下，让选出来的商品价格总和最大程度地接近满减条件（200 元），这样就可以极大限度地“薅羊毛”。作为程序员的你，能不能编个代码来帮她搞定呢？

要想高效地解决这个问题，就要用到我们今天讲的动态规划（Dynamic Programming）。

动态规划学习路线

动态规划比较适合用来求解最优问题，比如求最大值、最小值等等。它可以非常显著地降低时间复杂度，提高代码的执行效率。不过，它也是出了名的难学。它的主要学习难点跟递归类似，那就是，求解问题的过程不太符合人类常规的思维方式。对于新手来说，要想入门确实不容易。不过，等你掌握了之后，你会发现，实际上并没有想象中那么难。

为了让你更容易理解动态规划，我分了三节给你讲解。这三节分别是，初识动态规划、动态规划理论、动态规划实战。

第一节，我会通过两个非常经典的动态规划问题模型，向你展示我们为什么需要动态规划，以及动态规划解题方法是如何演化出来的。实际上，你只要掌握了这两个例子的解决思路，对于其他很多动态规划问题，你都可以套用类似的思路来解决。

第二节，我会总结动态规划适合解决的问题的特征，以及动态规划解题思路。除此之外，我还会将贪心、分治、回溯、动态规划这四种算法思想放在一起，对比分析它们各自的特点以及适用的场景。


第三节，我会教你应用第二节讲的动态规划理论知识，实战解决三个非常经典的动态规划问题，加深你对理论的理解。弄懂了这三节中的例子，对于动态规划这个知识点，你就算是入门了。

0-1 背包问题

我在讲贪心算法、回溯算法的时候，多次讲到背包问题。今天，我们依旧拿这个问题来举例。

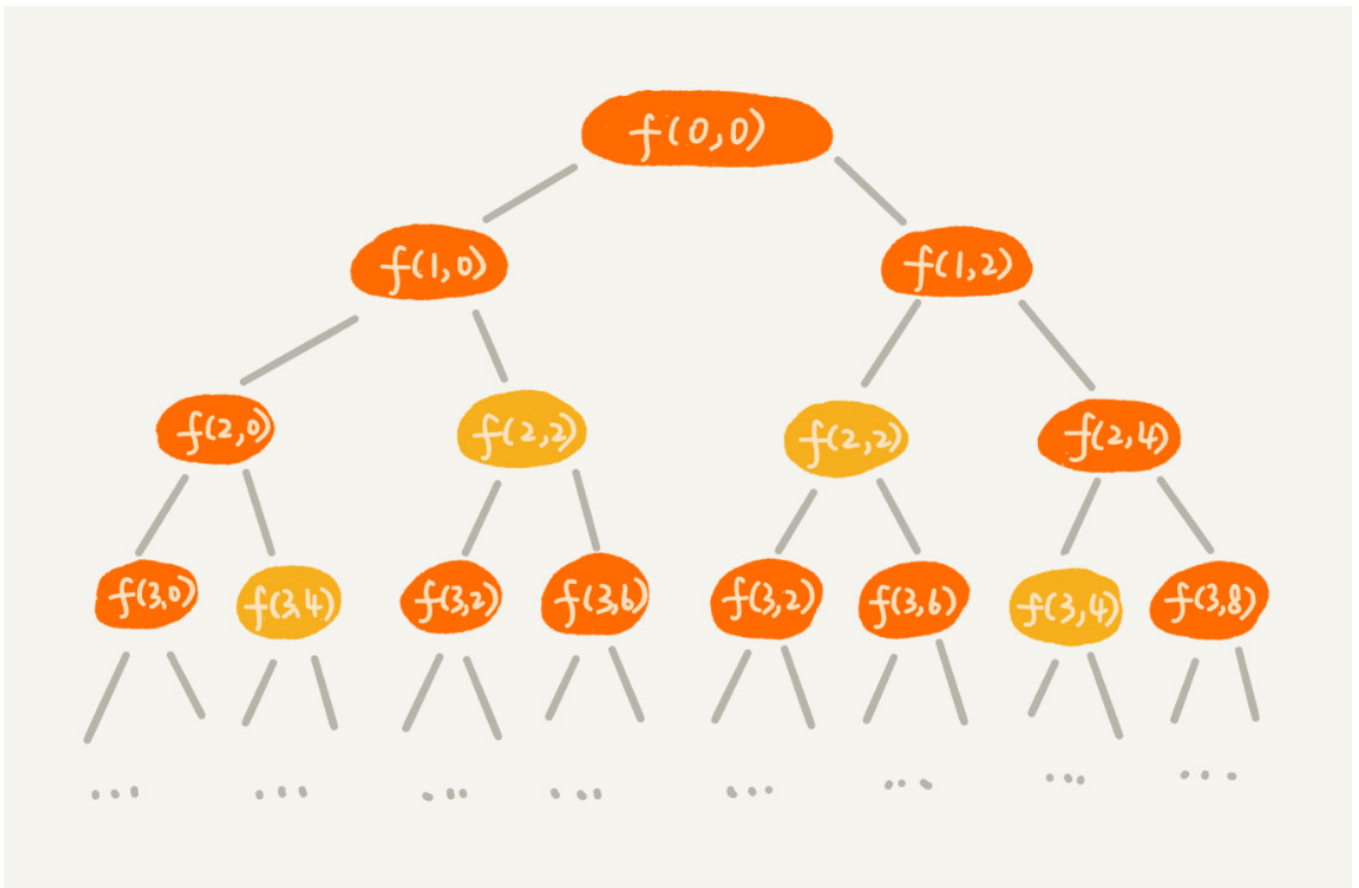
对于一组不同重量、不可分割的物品，我们需要选择一些装入背包，在满足背包最大重量限制的前提下，背包中物品总重量的最大值是多少呢？

关于这个问题，我们上一节讲了回溯的解决方法，也就是穷举搜索所有可能的装法，然后找出满足条件的最大值。不过，回溯算法的复杂度比较高，是指数级别的。那有没有什么规律，可以有效降低时间复杂度呢？我们一起来看看。

 复制代码

```
1 // 回溯算法实现。注意：我把输入的变量都定义成了成员变量。
2 private int maxW = Integer.MIN_VALUE; // 结果放到 maxW 中
3 private int[] weight = {2, 2, 4, 6, 3}; // 物品重量
4 private int n = 5; // 物品个数
5 private int w = 9; // 背包承受的最大重量
6 public void f(int i, int cw) { // 调用 f(0, 0)
7     if (cw == w || i == n) { // cw==w 表示装满了，i==n 表示物品都考察完了
8         if (cw > maxW) maxW = cw;
9         return;
10    }
11    f(i+1, cw); // 选择不装第 i 个物品
12    if (cw + weight[i] <= w) {
13        f(i+1, cw + weight[i]); // 选择装第 i 个物品
14    }
15 }
```

规律是不是不好找？那我们就举个例子、画个图看看。我们假设背包的最大承载重量是 9。我们有 5 个不同的物品，每个物品的重量分别是 2, 2, 4, 6, 3。如果我们把这个例子的回溯求解过程，用递归树画出来，就是下面这个样子：



递归树中的每个节点表示一种状态，我们用 (i, cw) 来表示。其中， i 表示将要决策第几个物品是否装入背包， cw 表示当前背包中物品的总重量。比如， $(2, 2)$ 表示我们将要决策第 2 个物品是否装入背包，在决策前，背包中物品的总重量是 2。

从递归树中，你应该能会发现，有些子问题的求解是重复的，比如图中 $f(2, 2)$ 和 $f(3, 4)$ 都被重复计算了两次。我们可以借助[递归](#)那一节讲的“备忘录”的解决方式，记录已经计算好的 $f(i, cw)$ ，当再次计算到重复的 $f(i, cw)$ 的时候，可以直接从备忘录中取出来用，就不用再递归计算了，这样就可以避免冗余计算。

复制代码

```

1 private int maxW = Integer.MIN_VALUE; // 结果放到 maxW 中
2 private int[] weight = {2, 2, 4, 6, 3}; // 物品重量
3 private int n = 5; // 物品个数
4 private int w = 9; // 背包承受的最大重量
5 private boolean[][] mem = new boolean[5][10]; // 备忘录，默认值 false
6 public void f(int i, int cw) { // 调用 f(0, 0)
7     if (cw == w || i == n) { // cw==w 表示装满了，i==n 表示物品都考察完了
8         if (cw > maxW) maxW = cw;
9         return;
10    }
11    if (mem[i][cw]) return; // 重复状态
12    mem[i][cw] = true; // 记录 (i, cw) 这个状态
13    f(i+1, cw); // 选择不装第 i 个物品
14    if (cw + weight[i] <= w) {
15        f(i+1, cw + weight[i]); // 选择装第 i 个物品
16    }
17 }

```

这种解决方法非常好。实际上，它已经跟动态规划的执行效率基本上没有差别。但是，多一种方法就多一种解决思路，我们现在来看看动态规划是怎么做的。

我们把整个求解过程分为 n 个阶段，每个阶段会决策一个物品是否放到背包中。每个物品决策（放入或者不放入背包）完之后，背包中的物品的重量会有多种情况，也就是说，会达到多种不同的状态，对应到递归树中，就是有很多不同的节点。

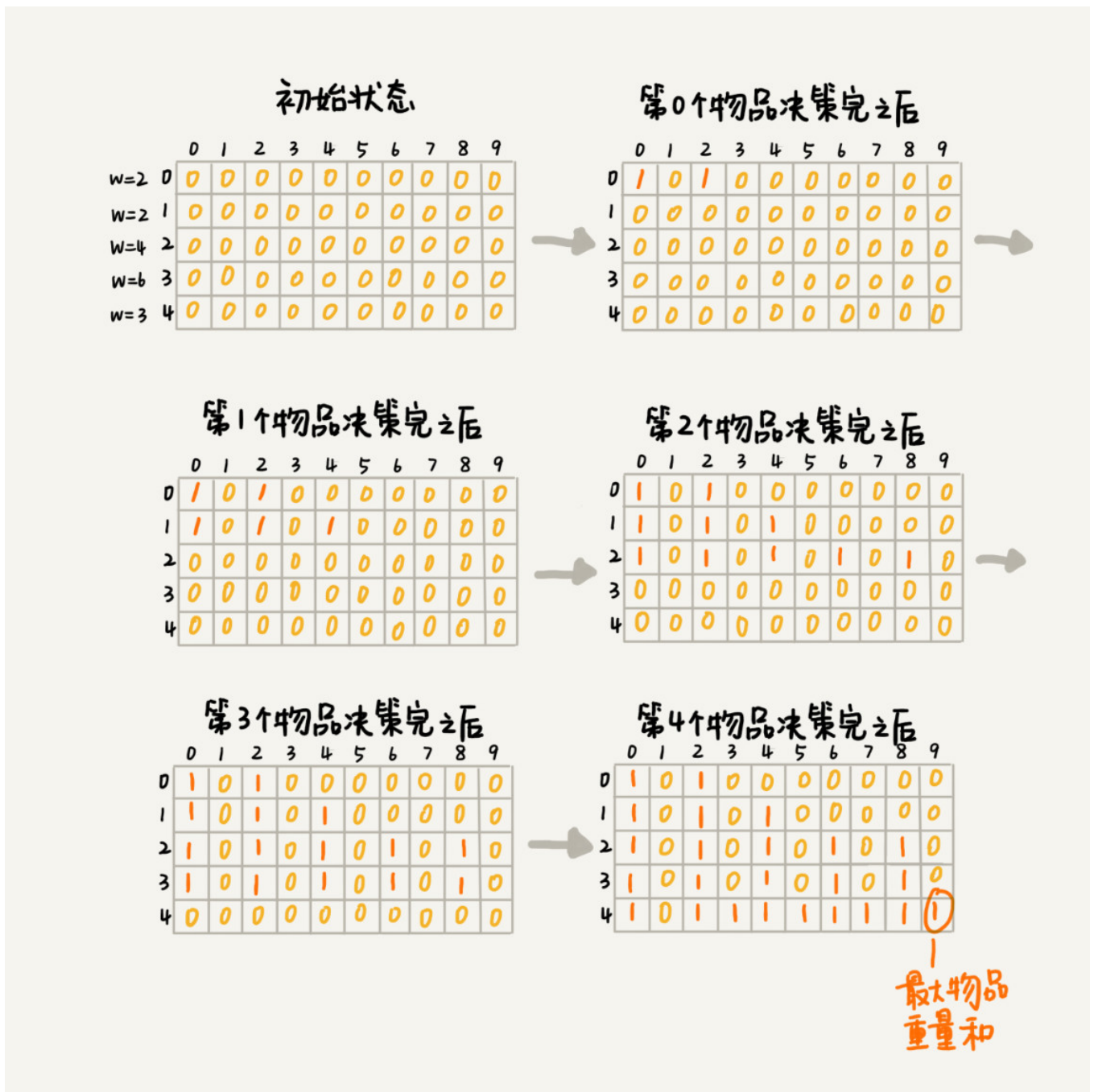
我们把每一层重复的状态（节点）合并，只记录不同的状态，然后基于上一层的状态集合，来推导下一层的状态集合。我们可以通过合并每一层重复的状态，这样就保证每一层不同状态的个数都不会超过 w 个（ w 表示背包的承载重量），也就是例子中的 9。于是，我们就成功避免了每层状态个数的指数级增长。

我们用一个二维数组 `states[n][w+1]`，来记录每层可以达到的不同状态。

第 0 个（下标从 0 开始编号）物品的重量是 2，要么装入背包，要么不装入背包，决策完之后，会对应背包的两种状态，背包中物品的总重量是 0 或者 2。我们用 `states[0][0]=true` 和 `states[0][2]=true` 来表示这两种状态。

第 1 个物品的重量也是 2，基于之前的背包状态，在这个物品决策完之后，不同的状态有 3 个，背包中物品总重量分别是 $0(0+0)$ ， $2(0+2 \text{ or } 2+0)$ ， $4(2+2)$ 。我们用 `states[1][0]=true`，`states[1][2]=true`，`states[1][4]=true` 来表示这三种状态。

以此类推，直到考察完所有的物品后，整个 `states` 状态数组就都计算好了。我把整个计算的过程画了出来，你可以看看。图中 0 表示 false，1 表示 true。我们只需要在最后一层，找一个值为 true 的最接近 w （这里是 9）的值，就是背包中物品总重量的最大值。



文字描述可能还不够清楚。我把上面的过程，翻译成代码，你可以结合着一块看下。

复制代码

```

1 weight: 物品重量, n: 物品个数, w: 背包可承载重量
2 public int knapsack(int[] weight, int n, int w) {
3     boolean[][] c = new boolean[n][w+1]; // 默认值 false
4     states[0][0] = true; // 第一行的数据要特殊处理, 可以利用哨兵优化
5     states[0][weight[0]] = true;
6     for (int i = 1; i < n; ++i) { // 动态规划状态转移
7         for (int j = 0; j <= w; ++j) { // 不把第 i 个物品放入背包
8             if (states[i-1][j] == true) states[i][j] = states[i-1][j];
9         }
10        for (int j = 0; j <= w-weight[i]; ++j) { // 把第 i 个物品放入背包
11            if (states[i-1][j]==true) states[i][j+weight[i]] = true;
12        }
13    }

14    for (int i = w; i >= 0; --i) { // 输出结果

```



```
15     if (states[n-1][i] == true) return i;
16 }
17 return 0;
18 }
```

实际上，这就是一种用动态规划解决问题的思路。我们把问题分解为多个阶段，每个阶段对应一个决策。我们记录每一个阶段可达的状态集合（去掉重复的），然后通过当前阶段的状态集合，来推导下一个阶段的状态集合，动态地往前推进。这也是动态规划这个名字的由来，你可以自己体会一下，是不是还挺形象的？

前面我们讲到，用回溯算法解决这个问题时间复杂度 $O(2^n)$ ，是指数级的。那动态规划解决方案的时间复杂度是多少呢？我来分析一下。

这个代码的时间复杂度非常好分析，耗时最多的部分就是代码中的两层 for 循环，所以时间复杂度是 $O(n*w)$ 。n 表示物品个数，w 表示背包可以承载的总重量。


从理论上讲，指数级的时间复杂度肯定要比 $O(n*w)$ 高很多，但是为了让你有更加深刻的感受，我来举一个例子给你比较一下。

我们假设有 10000 个物品，重量分布在 1 到 15000 之间，背包可以承载的总重量是 30000。如果我们用回溯算法解决，用具体的数值表示出时间复杂度，就是 2^{10000} ，这是一个相当大的一个数字。如果我们用动态规划解决，用具体的数值表示出时间复杂度，就是 $10000*30000$ 。虽然看起来也很大，但是和 2^{10000} 比起来，要小太多了。

尽管动态规划的执行效率比较高，但是就刚刚的代码实现来说，我们需要额外申请一个 n 乘以 w+1 的二维数组，对空间的消耗比较多。所以，有时候，我们会说，动态规划是一种空间换时间的解决思路。你可能要问了，有什么办法可以降低空间消耗吗？

实际上，我们只需要一个大小为 w+1 的一维数组就可以解决这个问题。动态规划状态转移的过程，都可以基于这个一维数组来操作。具体的代码实现我贴在这里，你可以仔细看下。

```
1 public static int knapsack2(int[] items, int n, int w) {
2     boolean[] states = new boolean[w+1]; // 默认值 false
3     states[0] = true; // 第一行的数据要特殊处理，可以利用哨兵优化
4     states[items[0]] = true;
5     for (int i = 1; i < n; ++i) { // 动态规划
6         for (int j = w-items[i]; j >= 0; --j) { // 把第 i 个物品放入背包
7             if (states[j]==true) states[j+items[i]] = true;
8         }
9     }
10    for (int i = w; i >= 0; --i) { // 输出结果
11        if (states[i] == true) return i;
12    }
13    return 0;
}
```

 复制代码

```
14 }
```

这里我特别强调一下代码中的第 6 行，j 需要从大到小来处理。如果我们按照 j 从小到大处理的话，会出现 for 循环重复计算的问题。你可以自己想一想，这里我就不详细说了。

0-1 背包问题升级版

我们继续升级难度。我改造了一下刚刚的背包问题。你看这个问题又该如何用动态规划解决？

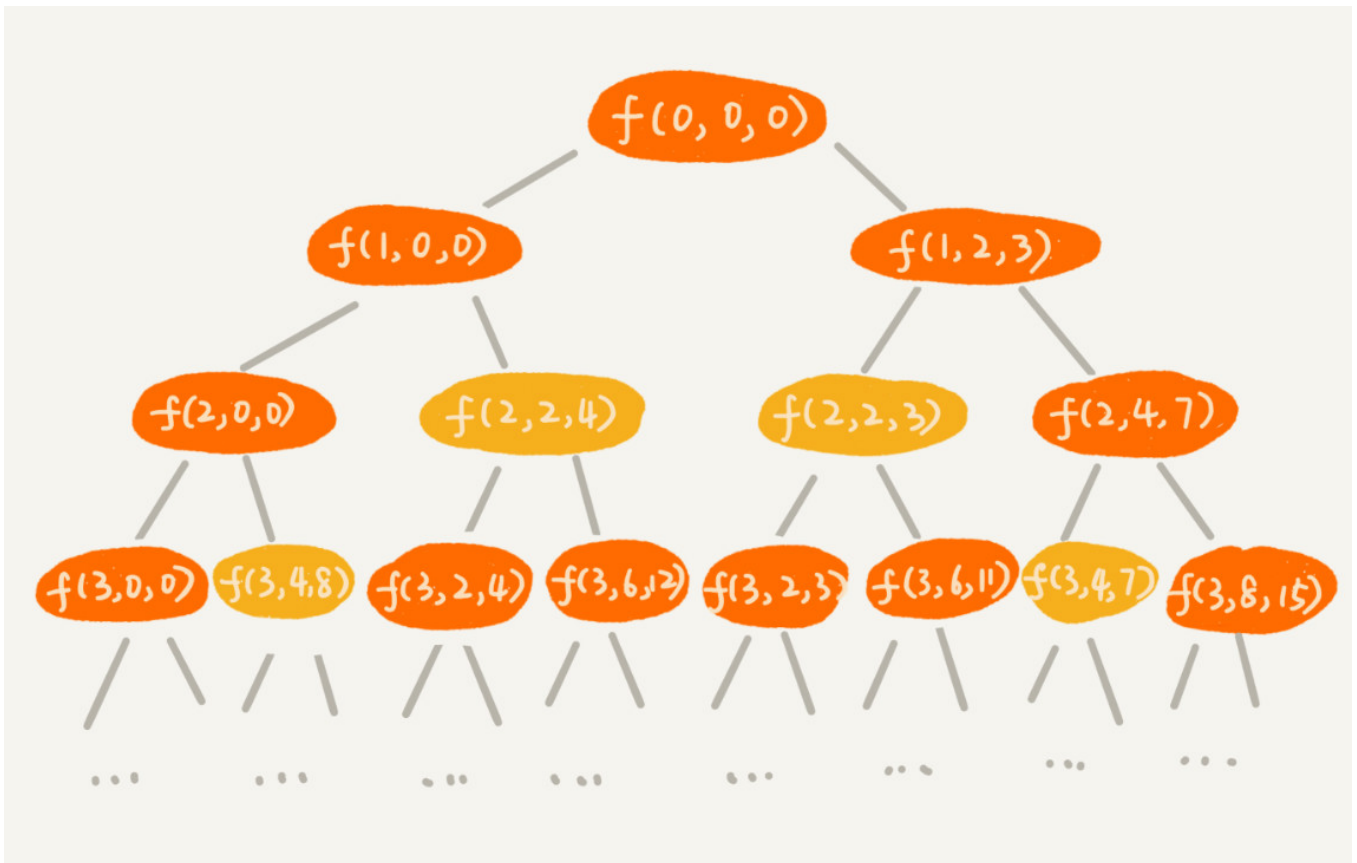
我们刚刚讲的背包问题，只涉及背包重量和物品重量。我们现在引入物品价值这一变量。对于一组不同重量、不同价值、不可分割的物品，我们选择将某些物品装入背包，在满足背包最大重量限制的前提下，背包中可装入物品的总价值最大是多少呢？

这个问题依旧可以用回溯算法来解决。这个问题并不复杂，所以具体的实现思路，我就不用文字描述了，直接给你看代码。

```
1 private int maxV = Integer.MIN_VALUE; // 结果放到 maxV 中
2 private int[] items = {2, 2, 4, 6, 3}; // 物品的重量
3 private int[] value = {3, 4, 8, 9, 6}; // 物品的价值
4 private int n = 5; // 物品个数
5 private int w = 9; // 背包承受的最大重量
6 public void f(int i, int cw, int cv) { // 调用 f(0, 0, 0)
7     if (cw == w || i == n) { // cw==w 表示装满了，i==n 表示物品都考察完了
8         if (cv > maxV) maxV = cv;
9         return;
10    }
11    f(i+1, cw, cv); // 选择不装第 i 个物品
12    if (cw + weight[i] <= w) {
13        f(i+1, cw+weight[i], cv+value[i]); // 选择装第 i 个物品
14    }
15 }
```

[复制代码](#)

针对上面的代码，我们还是照例画出递归树。在递归树中，每个节点表示一个状态。现在我们需要 3 个变量 (i, cw, cv) 来表示一个状态。其中，i 表示即将要决策第 i 个物品是否装入背包，cw 表示当前背包中物品的总重量，cv 表示当前背包中物品的总价值。



我们发现，在递归树中，有几个节点的 i 和 cw 是完全相同的，比如 $f(2, 2, 4)$ 和 $f(2, 2, 3)$ 。在背包中物品总重量一样的情况下， $f(2, 2, 4)$ 这种状态对应的物品总价值更大，我们可以舍弃 $f(2, 2, 3)$ 这种状态，只需要沿着 $f(2, 2, 4)$ 这条决策路线继续往下决策就可以。

也就是说，对于 (i, cw) 相同的不同状态，那我们只需要保留 cv 值最大的那个，继续递归处理，其他状态不予考虑。

思路说完了，但是代码如何实现呢？如果用回溯算法，这个问题就没法再用“备忘录”解决了。所以，我们就需要换一种思路，看看动态规划是不是更容易解决这个问题？

我们还是把整个求解过程分为 n 个阶段，每个阶段会决策一个物品是否放到背包中。每个阶段决策完之后，背包中的物品的总重量以及总价值，会有多种情况，也就是会达到多种不同的状态。

我们用一个二维数组 $states[n][w+1]$ ，来记录每层可以达到的不同状态。不过这里数组存储的值不再是 `boolean` 类型的了，而是当前状态对应的最大总价值。我们把每一层中 (i, cw) 重复的状态（节点）合并，只记录 cv 值最大的那个状态，然后基于这些状态来推导下一层的状态。

我们把这个动态规划的过程翻译成代码，就是下面这个样子：

```
1 public static int knapsack3(int[] weight, int[] value, int n, int w) {
2     int[][] states = new int[n][w+1];
3     for (int i = 0; i < n; ++i) { // 初始化 states
4
5         for (int j = 0; j < w+1; ++j) {
```

[复制代码](#)


```
5     states[i][j] = -1;
6 }
7 }
8 states[0][0] = 0;
9 states[0][weight[0]] = value[0];
10 for (int i = 1; i < n; ++i) { // 动态规划，状态转移
11     for (int j = 0; j <= w; ++j) { // 不选择第 i 个物品
12         if (states[i-1][j] >= 0) states[i][j] = states[i-1][j];
13     }
14     for (int j = 0; j <= w-weight[i]; ++j) { // 选择第 i 个物品
15         if (states[i-1][j] >= 0) {
16             int v = states[i-1][j] + value[i];
17             if (v > states[i][j+weight[i]]) {
18                 states[i][j+weight[i]] = v;
19             }
20         }
21     }
22 }
23 // 找出最大值
24 int maxvalue = -1;
25 for (int j = 0; j <= w; ++j) {
26     if (states[n-1][j] > maxvalue) maxvalue = states[n-1][j];
27 }
28 return maxvalue;
29 }
```

关于这个问题的时间、空间复杂度的分析，跟上一个例子大同小异，所以我就不赘述了。我直接给出答案，时间复杂度是 $O(n*w)$ ，空间复杂度也是 $O(n*w)$ 。跟上一个例子类似，空间复杂度也是可以优化的，你可以自己写一下。

解答开篇

掌握了今天讲的两个问题之后，你是不是觉得，开篇的问题很简单？


对于这个问题，你当然可以利用回溯算法，穷举所有的排列组合，看大于等于 200 并且最接近 200 的组合是哪一个？但是，这样效率太低了点，时间复杂度非常高，是指数级的。当 n 很大的时候，可能“双十一”已经结束了，你的代码还没有运行出结果，这显然会让你在女朋友心中的形象大大减分。

实际上，它跟第一个例子中讲的 0-1 背包问题很像，只不过是把“重量”换成了“价格”而已。购物车中有 n 个商品。我们针对每个商品都决策是否购买。每次决策之后，对应不同的状态集合。我们还是用一个二维数组 $states[n][x]$ ，来记录每次决策之后所有可达的状态。不过，这里的 x 值是多少呢？

0-1 背包问题中，我们找的是小于等于 w 的最大值， x 就是背包的最大承载重量 $w+1$ 。对于这个问题来说，我们要找的是大于等于 200（满减条件）的值中最小的，所以就不能设置为 200

加 1 了。就这个实际的问题而言，如果要购买的物品的总价格超过 200 太多，比如 1000，那这个羊毛“薅”得就没有太大意义了。所以，我们可以限定 x 值为 1001。

不过，这个问题不仅要求大于等于 200 的总价格中的最小的，我们还要找出这个最小总价格对应都要购买哪些商品。实际上，我们可以利用 states 数组，倒推出这个被选择的商品序列。我先把代码写出来，待会再照着代码给你解释。

 复制代码

```
1 // items 商品价格, n 商品个数, w 表示满减条件, 比如 200
2 public static void double11advance(int[] items, int n, int w) {
3     boolean[][] states = new boolean[n][3*w+1]; // 超过 3 倍就没有薅羊毛的价值了
4     states[0][0] = true; // 第一行的数据要特殊处理
5     states[0][items[0]] = true;
6     for (int i = 1; i < n; ++i) { // 动态规划
7         for (int j = 0; j <= 3*w; ++j) { // 不购买第 i 个商品
8             if (states[i-1][j] == true) states[i][j] = states[i-1][j];
9         }
10        for (int j = 0; j <= 3*w-items[i]; ++j) { // 购买第 i 个商品
11            if (states[i-1][j]==true) states[i][j+items[i]] = true;
12        }
13    }
14
15    int j;
16    for (j = w; j < 3*w+1; ++j) {
17        if (states[n-1][j] == true) break; // 输出结果大于等于 w 的最小值
18    }
19    if (j == -1) return; // 没有可行解
20    for (int i = n-1; i >= 1; --i) { // i 表示二维数组中的行, j 表示列
21        if(j-items[i] >= 0 && states[i-1][j-items[i]] == true) {
22            System.out.print(items[i] + " "); // 购买这个商品
23            j = j - items[i];
24        } // else 没有购买这个商品, j 不变。
25    }
26    if (j != 0) System.out.print(items[0]);
27 }
```

代码的前半部分跟 0-1 背包问题没有什么不同，我们着重看后半部分，看它是如何打印出选择购买哪些商品的。

状态 (i, j) 只有可能从 (i-1, j) 或者 (i-1, j-value[i]) 两个状态推导过来。所以，我们就检查这两个状态是否是可达的，也就是 states[i-1][j] 或者 states[i-1][j-value[i]] 是否是 true。

如果 states[i-1][j] 可达，就说明我们没有选择购买第 i 个商品，如果 states[i-1][j-value[i]] 可达，那就说明我们选择了购买第 i 个商品。我们从中选择一个可达的状态（如果两个都可达，就随意选择一个），然后，继续迭代地考察其他商品是否有选择购买。

内容小结

动态规划的第一节到此就讲完了。内容比较多，你可能需要多一点时间来消化。为了帮助你有的放矢地学习，我来强调一下，今天你应该掌握的重点内容。

今天的内容不涉及动态规划的理论，我通过两个例子，给你展示了动态规划是如何解决问题的，并且一点一点详细给你讲解了动态规划解决问题的思路。这两个例子都是非常经典的动态规划问题，只要你真正搞懂这两个问题，基本上动态规划已经入门一半了。所以，你要多花点时间，真正弄懂这两个问题。

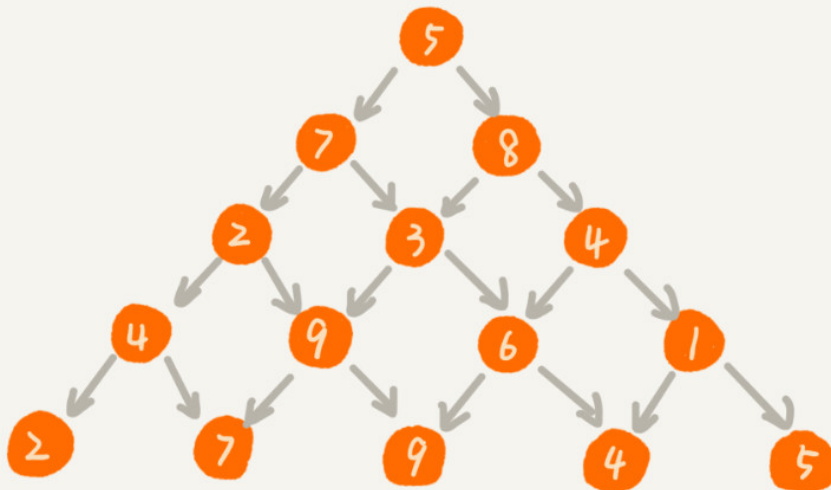
从例子中，你应该能发现，大部分动态规划能解决的问题，都可以通过回溯算法来解决，只不过回溯算法解决起来效率比较低，时间复杂度是指数级的。动态规划算法，在执行效率方面，要高很多。尽管执行效率提高了，但是动态规划的空间复杂度也提高了，所以，很多时候，我们会说，动态规划是一种空间换时间的算法思想。

我前面也说了，今天的内容并不涉及理论的知识。这两个例子的分析过程，我并没有涉及任何高深的理论方面的东西。而且，我个人觉得，贪心、分治、回溯、动态规划，这四个算法思想有关的理论知识，大部分都是“后验性”的，也就是说，在解决问题的过程中，我们往往是先想到如何用某个算法思想解决问题，然后再用算法理论知识，去验证这个算法思想解决问题的正确性。所以，你大可不必过于急于寻求动态规划的理论知识。

课后思考

“杨辉三角”不知道你听说过吗？我们现在对它进行一些改造。每个位置的数字可以随意填写，经过某个数字只能到达下面一层相邻的两个数字。

假设你站在第一层，往下移动，我们把移动到最底层所经过的所有数字之和，定义为路径的长度。请你编程求出从最高层移动到最底层的最短路径长度。



欢迎留言和我分享，也欢迎点击“[请朋友读](#)”，把今天的内容分享给你的好友，和他一起讨论、学习。



数据结构与算法之美

为工程师量身打造的数据结构与算法私教课

王争

前 Google 工程师



新版升级：点击「[请朋友读](#)」，10位好友免费读，邀请订阅更有[现金奖励](#)。

©版权归极客邦科技所有，未经许可不得转载

上一篇 39 | 回溯算法：从电影《蝴蝶效应》中学习回溯算法的核心思想

下一篇 不定期福利第四期 | 刘超：我是怎么学习《数据结构与算法之美》的？

写留言

精选留言



茴香根

12

我理解的动态规划，就是从全遍历的递归树为出发点，广度优先遍历，在遍历完每一层之后对每层结果进行合并（结果相同的）或舍弃（已经超出限制条件的），确保下一层遍历的数量不会超过限定条件数 W ，通过这个操作达到大大减少不必要遍历的目的。
在空间复杂度优化上，通过在计算中只保留最优结果的目的重复利用内存空间。

2018-12-26



zixuan

6

贪心：一条路走到黑，就一次机会，只能哪边看着顺眼走哪边
回溯：一条路走到黑，无数次重来的机会，还怕我走不出来 (Snapshot View)

动态规划：拥有上帝视角，手握无数平行宇宙的历史存档，同时发展出无数个未来 (Versioned Archive View)

2018-12-30



hfy

首先得有个女朋友

2018-12-26

5



Andylee

老师，倒数第二段的代码(背包升级版)的12行的if条件判断是不是写错了

2018-12-26

4

作者回复

是的 我改下

2018-12-26



feifei

这个动态规划学习了三天了，把老师的代码都手练了一遍，感觉对动态规划有点感觉了！然后在写这个课后题，我也练了一遍，我练了这么多，但我觉得动态规则这个最重要的是每层可达的状态这个怎么计算的，这是重点，我开始的时候，用纸和笔，把老师的第一例子，中的状态都画了出来，然后再来看代码，感觉很有帮助！

3

杨晖三角的代码我我也贴出来，希望对其他童鞋有帮助，老师，也麻烦你帮忙看下，看我的实现是否存在问题，谢谢！

由于这个限制，限制长度，没有贴出来倒推出路径，可查看我的git

<https://github.com/kkzfl22/datastruct/blob/master/src/main/java/com/liujun/datastruct/algorithm/dynamicProgramming/triangle/Triangle.java>

```
int[][] status = new int[triangles.length][triangles[triangles.length - 1].length];
```

```
int startPoint = triangles.length - 1;
```

```
int maxpoint = triangles[triangles.length - 1].length;
```

```
// 初始化相关的数据
```

```
for (int i = 0; i <= startPoint; i++) {  
    for (int j = 0; j < maxpoint; j++) {  
        status[i][j] = -1;  
    }  
}
```

```
// 初始化杨晖三解的第一个顶点
```

```
status[0][startPoint] = triangles[0][startPoint];
```



```
// 开始求解第二个三角形顶点
// 按层级遍历
for (int i = 1; i <= startPoint; i++) {
    // 加入当前的位置节点
    int currIndex = 0;
    while (currIndex < maxpoint) {
        if (status[i - 1][currIndex] > 0) {
            // 计算左节点
            int leftValue = status[i - 1][currIndex] + triangles[i][currIndex - 1];

            // 1,检查当前左节点是否已经设置,如果没有,则直接设置
            if (status[i][currIndex - 1] == -1) {
                status[i][currIndex - 1] = leftValue;
            } else {
                if (leftValue < status[i][currIndex - 1]) {
                    status[i][currIndex - 1] = leftValue;
                }
            }
            // 计算右节点
            int rightValue = status[i - 1][currIndex] + triangles[i][currIndex + 1];

            if (status[i][currIndex + 1] == -1) {
                status[i][currIndex + 1] = rightValue;
            }
            currIndex++;
        }
        currIndex++;
    }

    int minValue = Integer.MAX_VALUE;
    for (int i = 0; i < maxpoint; i++) {
        if (minValue > status[startPoint][i] && status[startPoint][i] != -1) {
            minValue = status[startPoint][i];
        }
    }
    System.out.println("最短路径结果为:" + minValue);
}
```

2018-12-28



P@trick

👍 3

老师你这个只能精确到元，女朋友羊毛精说要精确到0.01元，时间空间复杂度增大100倍



2018-12-26

作者回复

说的没错

2018-12-26



郭霖

2

王争老师动态规划讲得确实精彩，就是课后练习没有答案，有时候解不出来会很难受。我是看了下一篇文章的讲解然后明白了这篇文章的课后习题解法，这里分享一下吧，希望对大家有帮助。

```
int[][] matrix = {{5},{7,8},{2,3,4},{4,9,6,1},{2,7,9,4,5}};

public int yanghuiTriangle(int[][] matrix) {
    int[][] state = new int[matrix.length][matrix.length];
    state[0][0] = matrix[0][0];
    for (int i = 1; i < matrix.length; i++) {
        for (int j = 0; j < matrix[i].length; j++) {
            if (j == 0) state[i][j] = state[i - 1][j] + matrix[i][j];
            else if (j == matrix[i].length - 1) state[i][j] = state[i - 1][j - 1] + matrix[i][j];
            else {
                int top1 = state[i - 1][j - 1];
                int top2 = state[i - 1][j];
                state[i][j] = Math.min(top1, top2) + matrix[i][j];
            }
        }
    }
    int minDis = Integer.MAX_VALUE;
    for (int i = 0; i < matrix[matrix.length - 1].length; i++) {
        int distance = state[matrix.length - 1][i];
        if (distance < minDis) minDis = distance;
    }
    return minDis;
}
```

2019-01-02



煦暖

2

老师你好，您在专栏里提到好几次哨兵，啥时候给我们讲解一下呢？

2018-12-28



失火的夏天

2

杨辉三角的动态规划转移方程是： $S[i][j] = \min(S[i-1][j], S[i-1][j-1]) + a[i][j]$ 。

其中a表示到这个点的value值，S表示到a[i][j]这个点的最短路径值。

这里没有做边界条件限制，只是列出一个方程通式。边界条件需要在代码里具体处理。个人感觉动态规划的思想关键在于如何列出动态规划方程，有了方程，代码基本就是水到渠成了。

2018-12-27



Monday

1

1、这里我特别强调一下代码中的第 6 行，j 需要从大到小来处理。

这里自己写代码调试完才恍然大悟，第i轮循环中新设置的值会干扰到后面的设置。

2、特别感谢争哥今天让其他的课程的老师来客串了一节课，让我有了更多的时间学习本节。

2018-12-28

作者回复

不着急你慢慢学就是了 不用非得跟的那么紧

2019-01-02



任悦

1

思考题这个杨辉三角有点巧了，最短路径就是最左边一列

2018-12-28



像玉一样的石头

1

老师，请教个问题，想了好久不知道该如何求解

关于汇率方面的，比如手里有100人民币，设计一个汇率转换的环，比如人民币-》美元-》日元-》韩元-》人民币，兑换一圈后，手里的钱一直在增加，这个问题该如何求解呢

2018-12-27



@

1

第三部分的代码，第11行是不是有问题？根据代码推不出states[4][3]=true???

2018-12-26



blackhole

1

有个疑问：

解答开篇的示例代码中，for (int j = 0; j <= w; ++j) {...} 和 for (int j = 0; j <= w-items[i]; ++j) {...} 的循环条件是不是有问题啊，应分别为 j <= 3 * w 和 j <= 3 * w - items[i] 吧？

2018-12-26

作者回复

是的 我改下 感谢

2018-12-26



家

1

是不是可以从下往上递推，每个节点都选择下一层能到的两个节点中最小的一个和本身相加，加到根节点应该就是最小值。

2018-12-26



Tenderness

👍 0

```
public static int distance(int[][] data){
//1.非空判断
//2.构造矩阵(不必要空间需要节省) 初始化 temp[i][j] 代表顶层走到i j 位置的最短路径
int[][] temp= new int[data.length][data[data.length -1].length];
//3.构造边界
for(int i = 0;i<data.length;i++) {
temp[i][0] = data[i][0];
}
//4.构造最短路径数据集合
for(int i = 1;i<data.length;i++) {
for(int j = 1;j<data[i].length;j++) {
temp[i][j] = Math.min(data[i-1][j], data[i][j-1]) + data[i][j];
}
}
//5.求解最短路径，最短路径在最后一行中找
//多少行
int n = data.length;
//多少列
int m = data[data.length -1].length;
int distance = Integer.MAX_VALUE;
for(int i = m-1;i>=0;--i) {
if(temp[n-1][i] < distance) {
distance = temp[n-1][i];
}
}
return distance;
}
```

2019-01-03



郝大全

👍 0

第一个动态规划代码第三行："boolean[][] c =boolean[n][w+1]; // 默认值 false"
变量应该是states,不是c

2019-01-03



这么写的闫

👍 0

"0-1 背包问题升级版" 上面的代码段11行，输出结果为什么是 "return i" ？

没看明白这个操作是什么意思，求老师指点

2019-01-03





Tsingxu

👍 0

关于背包升级为有价值区别的题中，如果用一维数组存储的话，我是这样写的（PHP）：

```
function knapsack4($weight, $value, $n, $w)
{
    for ($i = 0; $i <= $w; $i++) {
        $states[$i] = -1;
    }
    $states[0] = 0;
    $states[$weight[0]] = $value[0];

    for ($i = 1; $i < $n; $i++) {
        for ($j = $w - $weight[$i]; $j >= 0; $j--) {
            if ($states[$j] && $j + $weight[$i] <= $w) {
                if ($weight[$i] == $j && $value[$i] > $states[$j]) $states[$j] = $value[$i];
                else $states[$j + $weight[$i]] = $states[$j] + $value[$i];
            }
        }
    }

    $maxValue = 0;
    for ($i = $w; $i >= 0; $i--) {
        if ($states[$i] > $maxValue) $maxValue = $states[$i];
    }

    return $maxValue;
}
```

虽然答案是一样的，但不知求解过程是否正确

2019-01-02



易波

👍 0

老师你好，关于动态规划第一段代码（函数：public int knapsack(int[] weight, int n, int w）

中，第5行，需要加上判断条件if (weight[0] <= w),避免数组越界，瑕不掩玉，老师讲解的非常好，思路清晰，受益匪浅！

2019-01-02



趙衍

👍 0

最近在学概率图模型，忽然觉得动态规划和图模型挺像的。当前状态依赖于上一个状态

2019-01-01



Kudo

👍 0

0-1背包python实现：

```
def backpack(items, w):  
    '''  
    # items: python list of item weights  
    # w: upper limit weight the backpack can load  
    '''  
  
    states = [False] * (w + 1) # initialize list with len w+1  
    states[0] = True; states[items[0]] = True # first row  
    for i in items[1:]: # traverse from index 1  
        for j in range(w-i,-1,-1): # traverse from back to front  
            if states[j] == True:  
                states[j+i] = True  
  
    for i in range(w,-1,-1): # output max weight  
        if states[i] == True:  
            print(i)  
            break  
  
    # how to use  
    items = [2, 2, 4, 6, 3]  
    backpack(items, 9)
```

2018-12-29



桂浩晋

👍 0

请教老师一个问题：您觉得为什么会有这些奇怪数据结构呢？

2018-12-29

| 作者回复

😊 很多问题都可以抽象成这些模型 所以才被人总结出来了

2019-01-02



小美

👍 0

看都可以看懂 但是自己根本写不出来 尤其是动态规划的那个数组

2018-12-28



Monster

👍 0

老师 states定义为boolean类型，if (states[i-1][j] == true) 可以简写为 if (states[i-1][j])

2018-12-27



小美

👍 0

哨兵 这块方便指导下吗

2018-12-27



© TM

0

哪位老铁用python实现一下？

2018-12-27



Wu

0

0-1背包升级版的代码实现中，第12行的if (states[i][j] >= 0) 的条件不需要吧。加上这个条件，下一个的值没有办法从上一个转化过来
请老师检查一下

2018-12-27



往事随风，顺其自然

0

5-7-2-4-2是最小的路径

2018-12-27



往事随风，顺其自然

0

这里面可以使用贪心算法解决，每次取最小的，查找对应的作用节点对应的最小值，然后依次递推下去

2018-12-27



往事随风，顺其自然

0

回溯算法，思想很好理解，但是代码哪里体现了回溯的？

2018-12-27



往事随风，顺其自然

0

首先得有钱，然后有女朋友，有女朋友没钱也没得买

2018-12-27



小蘑菇丢丢

0

比如，(2, 2) 表示我们将要决策第2个物品是否装入背包，在决策前，背包中物品的总重量是2。我理解的这个不是决策后的总重量是2？

2018-12-26



纯洁的憎恶

0

上学的时候就没搞懂动态规划，现在指望一天就学会也是有点不现实。今天的内容除了一维数组的0-1背包动态规划代码没看太明白，其他理解起来倒是不难，可是如果让我独立把代码写出来还是很困难，包括回溯算法也是，也许是因为递归与动态规划都太反直觉了吧？

动态规划把整个计算过程（如递归树）分成若干阶段，每个阶段会有若干种状态，其中存在重复的状态，于是可以通过合并每个阶段重复状态的方式，大大降低计算复杂度。并且下一个阶段的状态可以通过上一阶段的结果计算出来。当计算到最后一个阶段后，就可以得到想要的结果了。如果通过递归树来分析动态规划的算法复杂度，那么动态规划相当于把递归树每一层需要计算的节点数量限定在一个相对较小的范围，比如n，这样每层最多计算n次，那么总的计算复杂度就是每层节点数×树的深度h，即O(n*h)。这样理解对么？

2018-12-26



frogoscar

👍 0

动态规划真是太帅了

2018-12-26



往事随风, 顺其自然

👍 0

```
for (int j = 0; j <= w; ++j) { // 不把第 i 个物品放入背包
```

```
if (states[i-1][j] == true) states[i][j] = states[i-1][j];
```

```
}
```

```
for (int j = 0; j <= w-weight[i]; ++j) { // 把第 i 个物品放入背包
```

```
if (states[i-1][j]==true) states[i][j+weight[i]] = true;
```

```
}
```

这两句不大明白为什么表示, 不把dii个物品背包, 比如

if (states[i-1][j] == true) states[i][j] = states[i-1][j]; 这个赋值的时候不是会把所有的情况都列出来, 包括i放入和不放入都添加进去。

2018-12-26

| 作者回复

状态转移。j不变 就说明重量没增加 就说明东西没放进去

2018-12-26



『LHCY』

👍 0

思考题

用数组存储每一层

[1]

[2,3]

[2,3,4]

二维数组 [i][j] i代表行, j代表第几个元素, 值为到达第i行, 第j个元素时的最小路径。

[0][0]是第一个数组的元素1

从第二行开始, 每次取[i-1][j]和[i-1][j-1]加上[i][j], 把小的存起来, 注意数组越界处理, 最后一行就是到达每个元素的最短路径, 超遍历一遍得到总得最小路径

2018-12-26



往事随风, 顺其自然

👍 0

```
for (int i = 1; i < n; ++i) {
```

这里面的不是i<=n?

2018-12-26

| 作者回复

=n怎么理解呢

2018-12-26



往事随风, 顺其自然

👍 0

为什么第一行要特殊处理?

2018-12-26

作者回复

不然代码没法写啊

2018-12-26



往事随风，顺其自然

0

`f(i+1, cw);` // 选择不装第 i 个物品

这个为什么表示不选择第 i 个物品，表示是否装入 $i+1$ 个，前面 i 个已经装入进去了不可以？

2018-12-26

作者回复

`cw` 值没变 你可以对比下面那行 `f(i+1, cw+...)`

2018-12-26



Monster

0

老师，我的基础有点不太好，我在调试0-1背包回溯算法的解法时，这个地方不太明白，第11行入栈5次后， i =物品数量， $maxV$ =放入背包数量，第9行return后，为什么 i 的值 i =物品数量-1

2018-12-26

作者回复

可以多练练写递归代码 回过头去重新看下递归那一节

2018-12-26



白了少年头

0

老师帮忙看下我写的程序思路对不对好吗？

```
def yanghui_triangle(items, n):
```

```
# 从第二行开始动态规划
```

```
for i in range(1, n):
```

```
for j in range(len(items[i])):
```

```
# 当前行第一个元素
```

```
if j == 0:
```

```
items[i][j] += items[i-1][j]
```

```
# 当前行最后一个元素
```

```
elif j == len(items[i]) - 1:
```

```
items[i][j] += items[i-1][j-1]
```

```
# 取上一行较小的元素相加
```

```
else:
```

```
items[i][j] += items[i-1][j-1] if items[i-1][j-1] < items[i-1][j] else items[i-1][j]
```

```
# 输出最小值
```

```
min_value = items[n-1][0]
```

```
for i in range(1, len(items[n-1])):
```

```
if items[n-1][i] < min_value:
```

```
min_value = items[n-1][i]
```

```
print(min_value)

def test():
    items = [
        [5],
        [7, 8],
        [2, 3, 4],
        [4, 9, 6, 1],
        [2, 7, 9, 4, 5]
    ]
    yanghui_triangle(items, 5)

if __name__ == '__main__':
    test()

2018-12-26
```



crazyone

0

动态规划求价值最大的，在不选择第i个物品下的if条件判断应该是if (states[i-1][j] >= 0)吧

2018-12-26

作者回复

对

2018-12-26



Pan^yu

0

课后习题：

- 1.以层数为限制条件
- 2.初始化两个Set，存放最短路径值，一个存上一层的值，一个存当前层的值，当前层遍历完时，当前层Set覆盖上层
- 3.初始化两个list，存放当前层的每个节点值，一个存上层，一个存当前层，当前层遍历完时，当前层list覆盖上层
- 4.循环遍历，遍历条件为上层list，每个节点的下层只能多两个，所以当前层节点个数为上层两倍，而且节点值都可以取随机数
- 5.最后从最底层的Set中取出最小的值

2018-12-26



在路边鼓掌的人

0

个人觉得第一个例子中求背包中物品总重量的最大值可以不用动态规划，可以用求sum close st的问题,可以先对数据排序，然后设定一个滑动窗口求最接近的值，总体的时间复杂度是nlogn，要比动态规划的时间复杂度和空间复杂度低一些。

2018-12-26



皇家救星

0

有个问题，像这种递归函数是否检查存在重叠，用画图方式方法确实很直观。但是画这种图，有没有什么技巧呢？有工具可以辅助生成吗

2018-12-26

作者回复

貌似没有 有的类似决策树哈

2018-12-26



棋圣

很棒

2018-12-26

👍 0