5kg的袋子

物品：

钱：6 10 12

Kg：1 2 4

我们把5kg的袋子，拆分成1kg，1kg这样子计算，里面的表格就表示当前重量下能装的最多的钱。表格的数列就表示是要装的物品

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 1kg | 2kg | 3kg | 4kg | 5kg |
| 加入物品1 | 6 | 6 | 6 | 6 | 6 |
| 加入物品2 | 6 | 10 | 10+6=16 | 10+6=16 | 16 |
| 加入物品3 | 6 | 10 | 16 | 16 | 18 |
|  |  |  |  |  |  |

加入物品2时，袋子当前为1kg 的容量时，我们发现物品2装不进去。那我们应该取多少呢？是不是只要取物品进来时1kg最大钱？，当袋子为2kg时，我们发现物品2可以装下去，此时可以得到10块钱，之前物品1进来时2kg最大是6吧，那我们肯定要选择大的这个10，而不是6.此时袋子还剩0kg可以装。

袋子为3kg时，我们还是可以装下这个物品2,得到10块，袋子还剩下1kg。

10+1kg能装的东西。

物品3来了，袋子为4kg时，物品3可以转进来，得到12块钱，袋子还剩0kg。

我发现我不装物品3 还能得到16呢

物品3来了，袋子为5kg时，物品3可以转进来，得到12块钱，袋子还剩1kg。那么装了物品3就能得到12+6=18块钱

我发现我不装物品3 能得到16，比18小，所以决定装。

上面这一个递推过程总结起来就是一个东西------**状态转移方程**：

能装的时候 每次和上面的比较，大我就装，否则就不装。

Max(money[i]+res[i-1][w-weight[i]],res[i-1][w]);

money[i]+res[i-1][w-weight[i]]:装这个物品

w-weight[i] :表示装完还剩下的空间

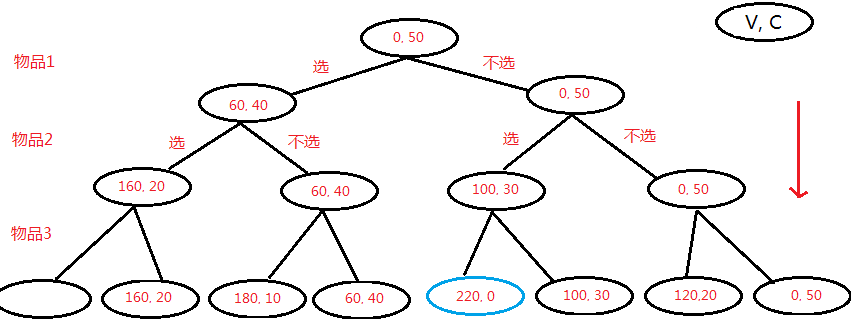
res[i-1][w-weight[i]]:表示装完后剩下的空间还能装的最大值，取上一次的结果。

Res[i-1][w]表示不装这个物品的值

**和遍历的比较及优化：**

遍历每次在物品加进来的时候都会保存选择与不选择两种状态那么这样下去越到后面状态保存的就越多其实就是2^n次，因为每个物品都有选与不选两种情况。而动态规划是每次都会把当前情况下的最优解计算出来，层层递推，下一层的最优解都是基于它上一次结果存下来的，所以最后一个结果就一定是最优解。

其实也是把问题都分解成了一个子问题，然后通过子问题去求解全局最优解。



**动归和贪心的比较：**

贪心是只管眼前不会管后的情况，而动归不一样，它的每次递推都是基于上一次的最优解进行。所以往往动归是一定能求出最优解的，而贪心不一定，这也是贪心算法的缺点，但是大家都看到了动归的时间复杂度是O(n\*m)而贪心是O(nlogn)，所以贪心算法的是高效的，动归如果子问题太多的话 就容易算不出结果，而且能用动归的问题往往用贪心都能解决一部分，甚至很大一部分。因此如果在实际项目中要求不是特别严的话 我建议使用贪心算法求最优解，其实我们很多时候并不用保证100%的准确，能尽量准确就可以了，贪心恰恰是符合这个规则的。