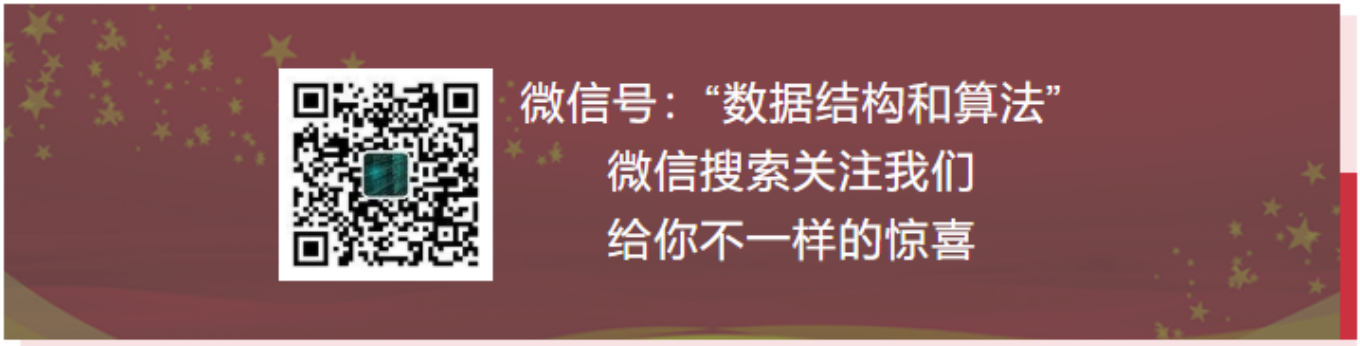
371，背包问题系列之-基础背包问题

原创 山大王wld 数据结构和算法 2020-05-27 23:30

收录于合集
#算法图文分析

164个



描述

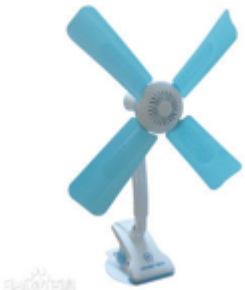
背包问题是动态规划中最经典的一道算法题。背包问题的种类比较多，我们先来看一个最简单的背包问题-基础背包。他是这样描述的。
有N件物品和一个容量为V的包，第i件物品的重量是w[i]，价值是v[i]，求将哪些物品装入背包可使这些物品的重量总和不能超过背包容量，且价值总和最大。我们先来举个例子分析一下

举例分析

假设我们背包可容纳的重量是4kg，有3样东西可供我们选择，一个是高压锅有4kg，价值300元，一个是风扇有3kg，价值200元，最后一个是一双运动鞋有1kg，价值150元。问要装哪些东西在重量不能超过背包容量的情况下价值最大。如果只装高压锅价值才300元，如果装风扇和运动鞋价值将达到350元，所以装风扇和运动鞋才是最优解，我们来画个图分析一下

01 结合图形分析

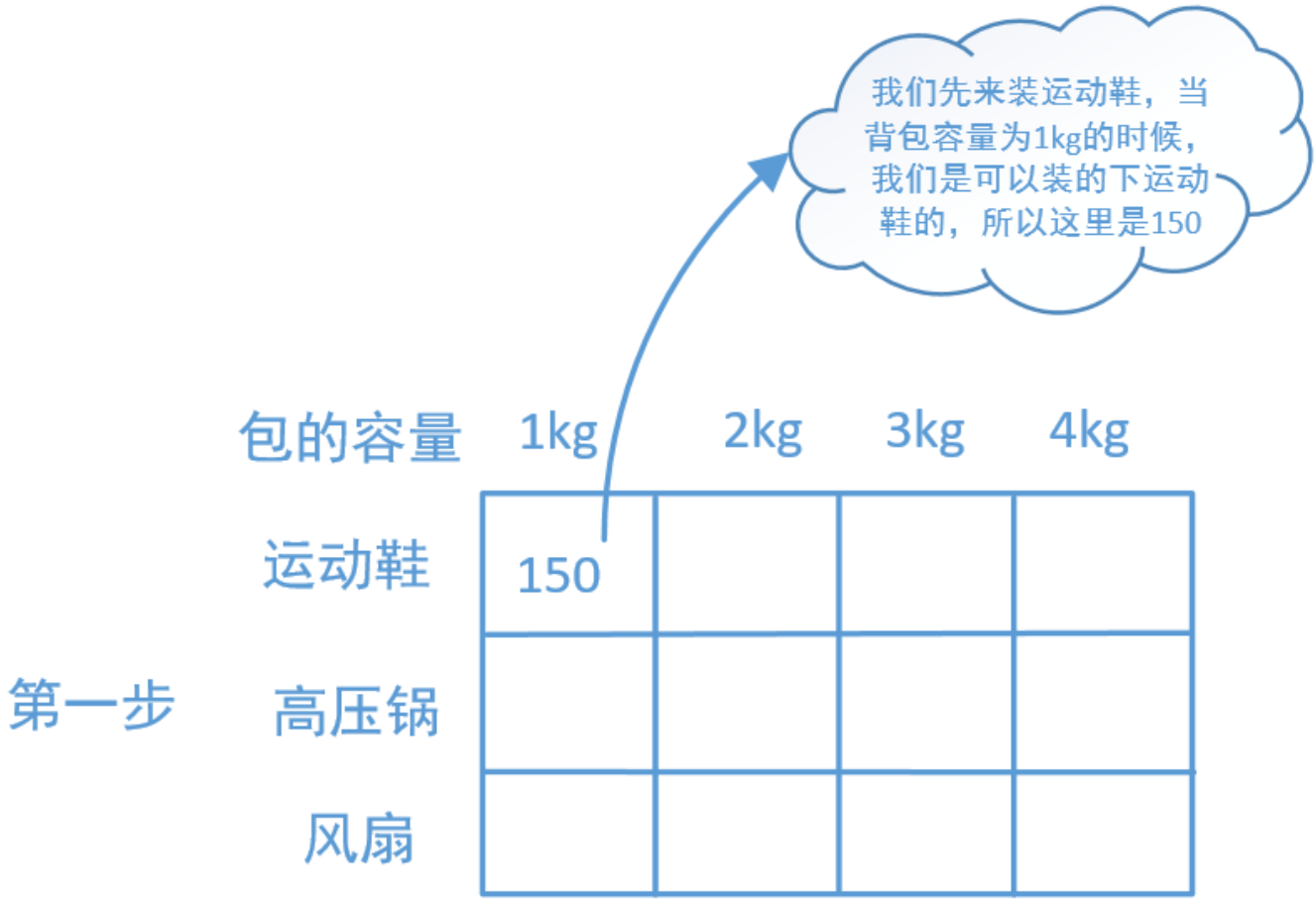
高压锅
重量：4kg
价值：300元

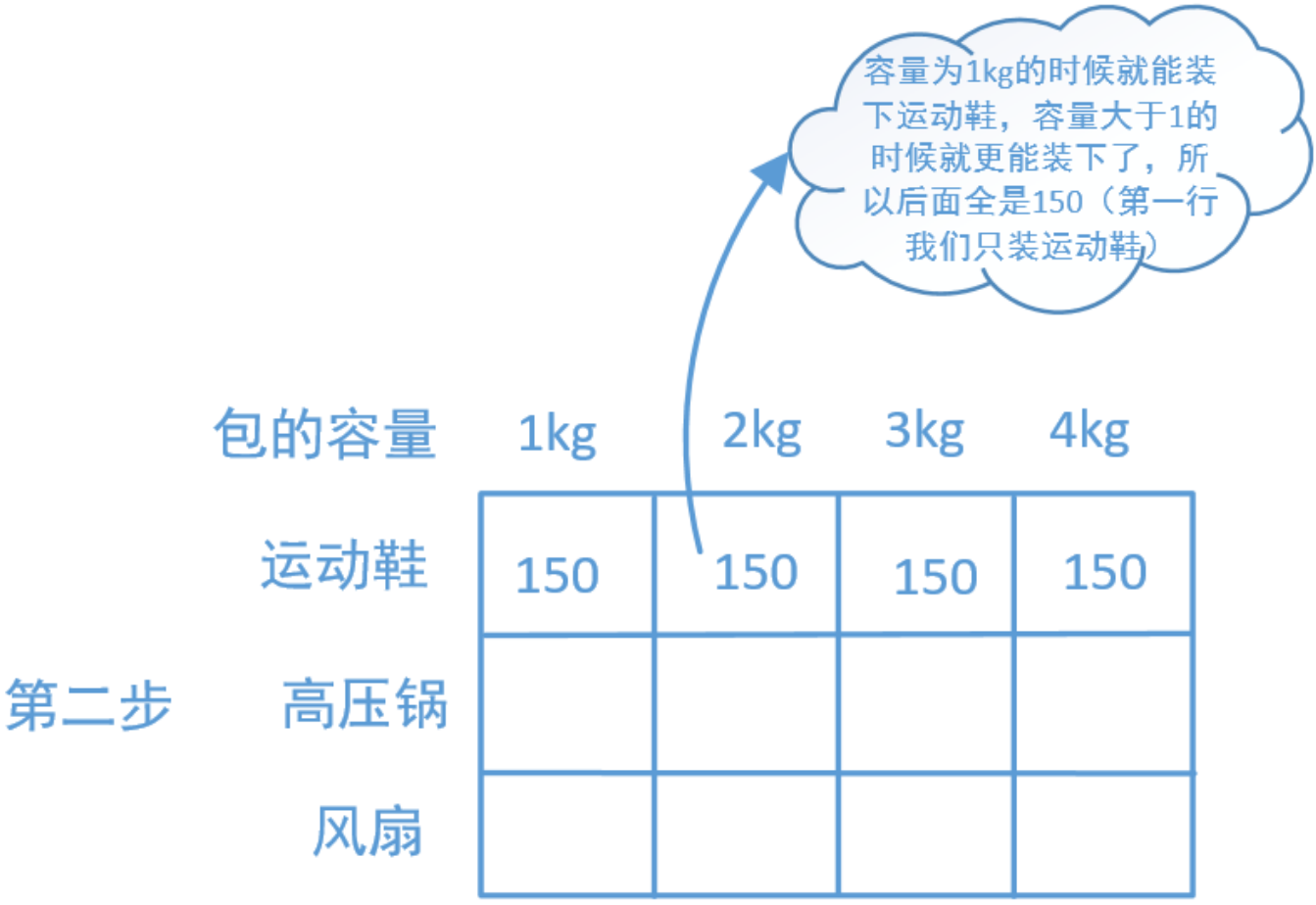


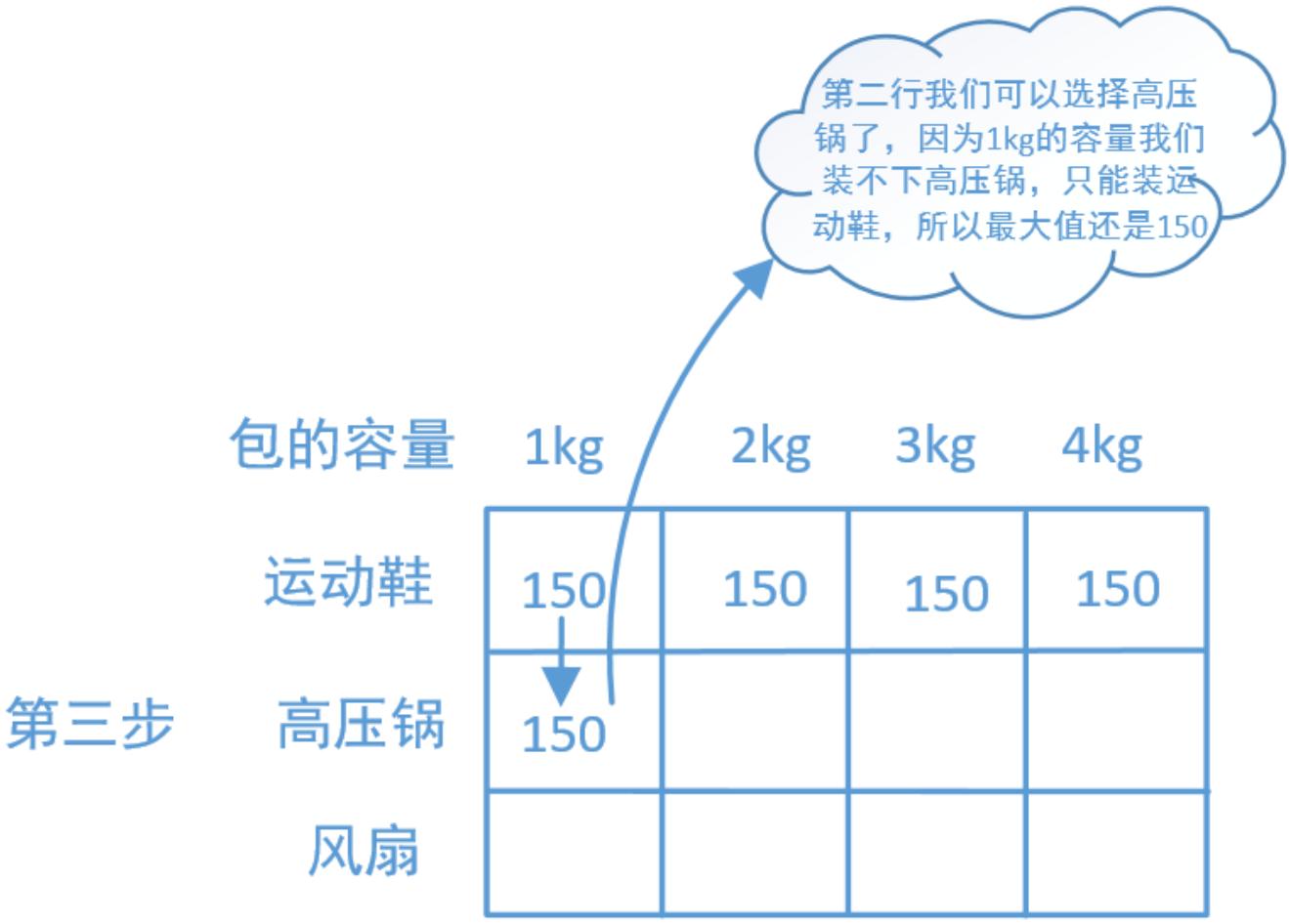
风扇
重量：3kg
价值：200元

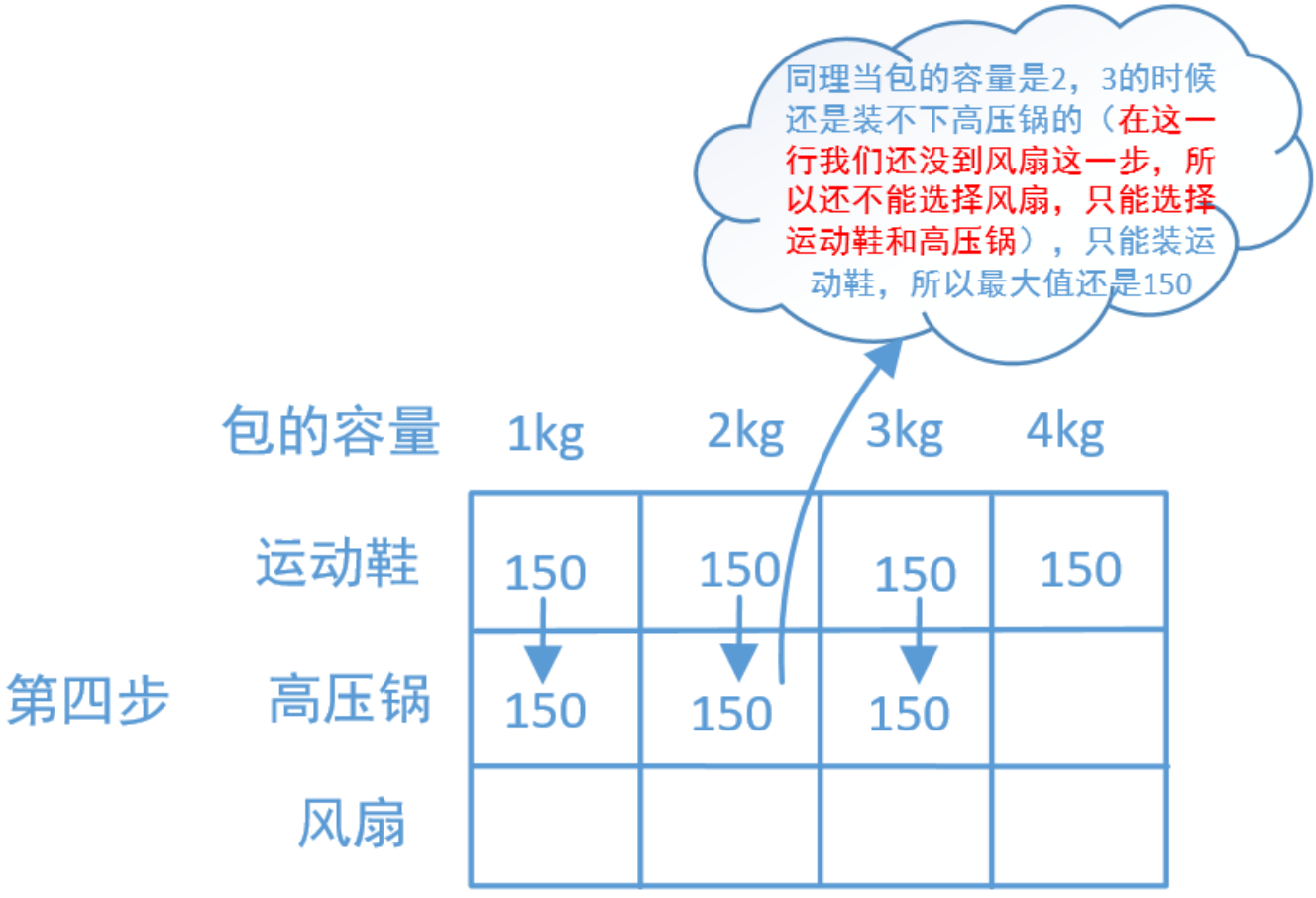
运动鞋
重量：1kg
价值：150元

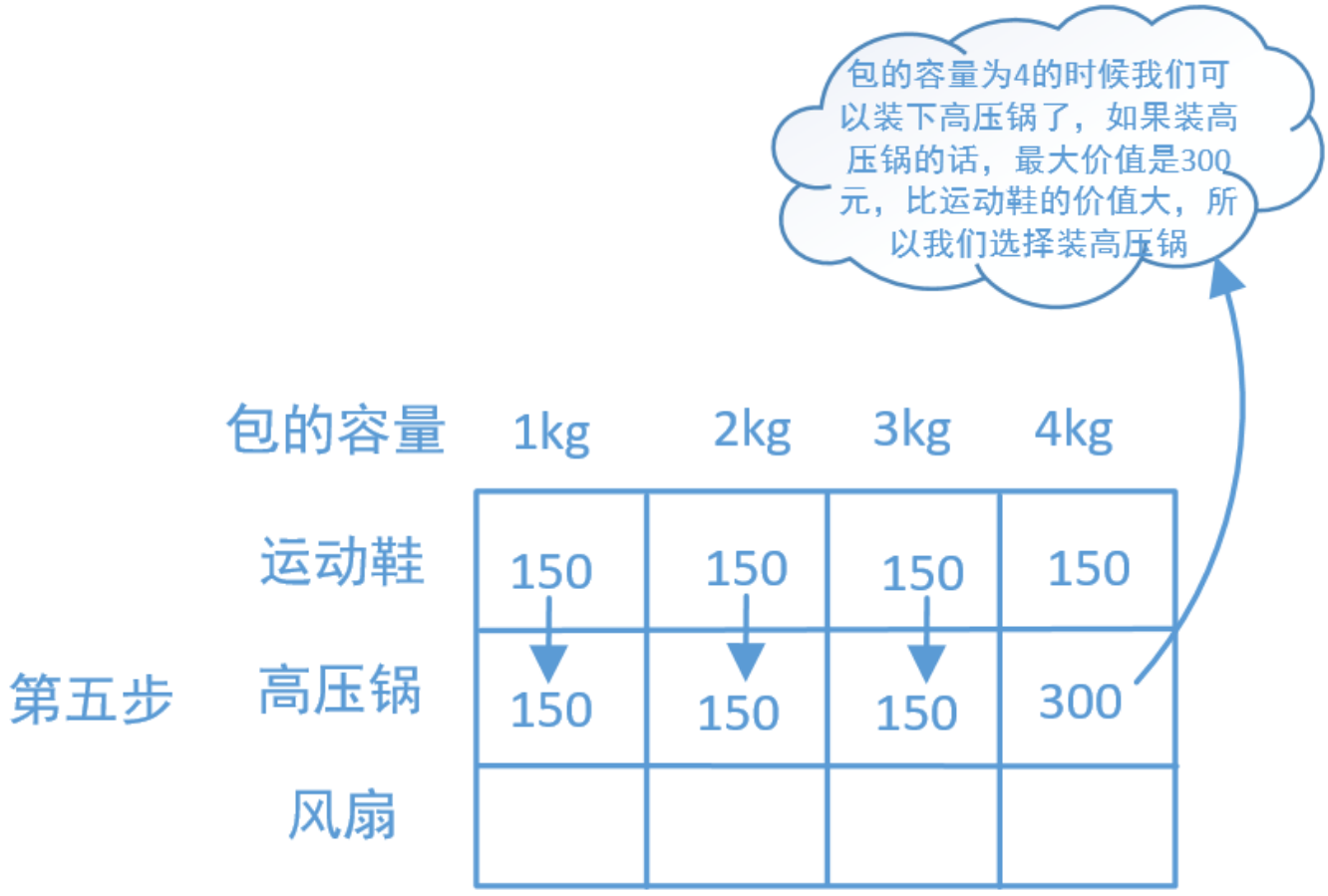
包的容量	1kg	2kg	3kg	4kg
原始状态	运动鞋				
	高压锅				
	风扇				


我们先来装运动鞋，当背包容量为1kg的时候，我们是可以装的下运动鞋的，所以这里是150
包的容量
1kg
2kg
3kg
4kg
运动鞋
150
第一步
高压锅
风扇

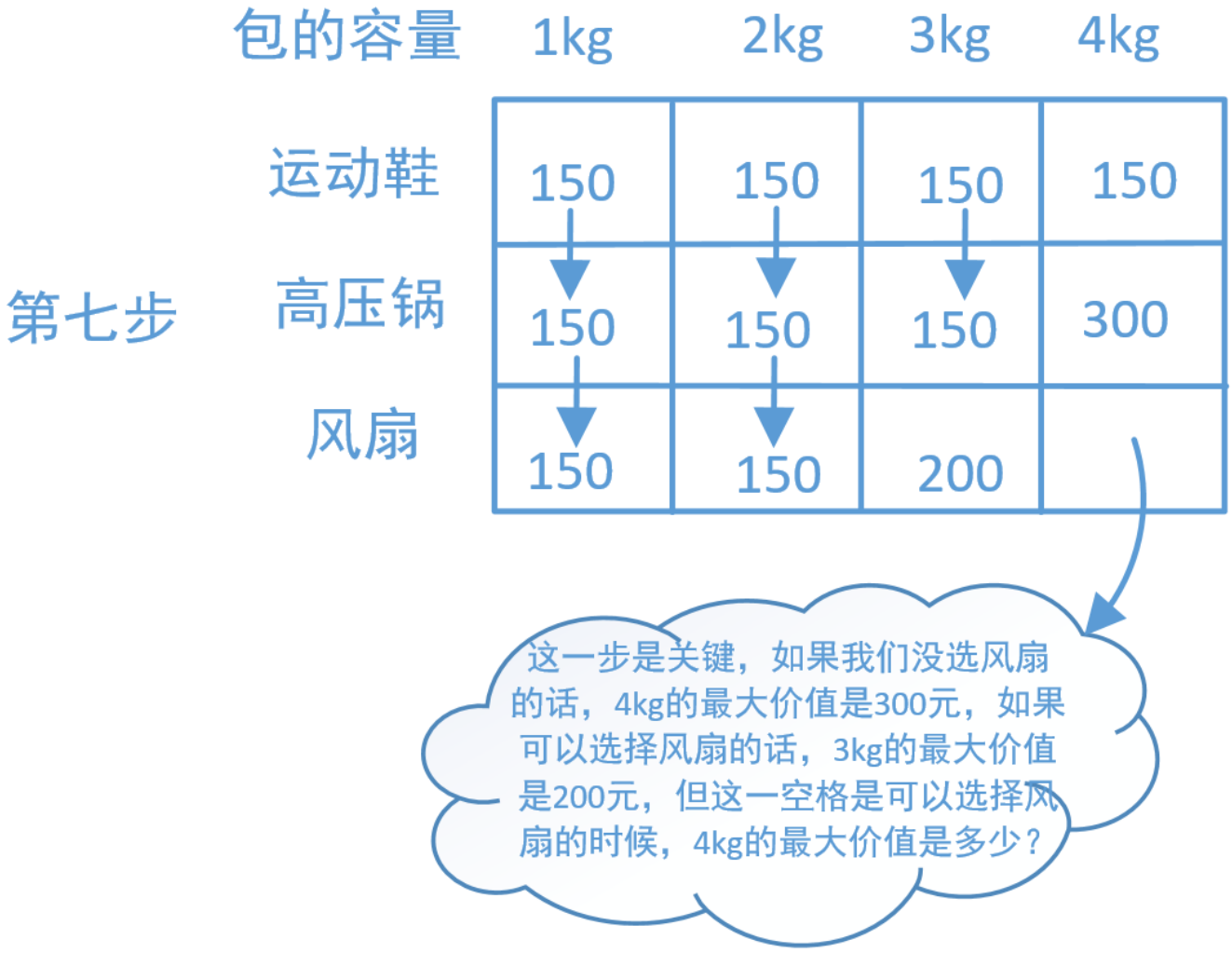

容量为1kg的时候就能装下运动鞋，容量大于1的时候就更能装下了，所以后面全是150（第一行我们只装运动鞋）
包的容量
1kg
2kg
3kg
4kg
运动鞋
150
150
150
150
第二步
高压锅
风扇

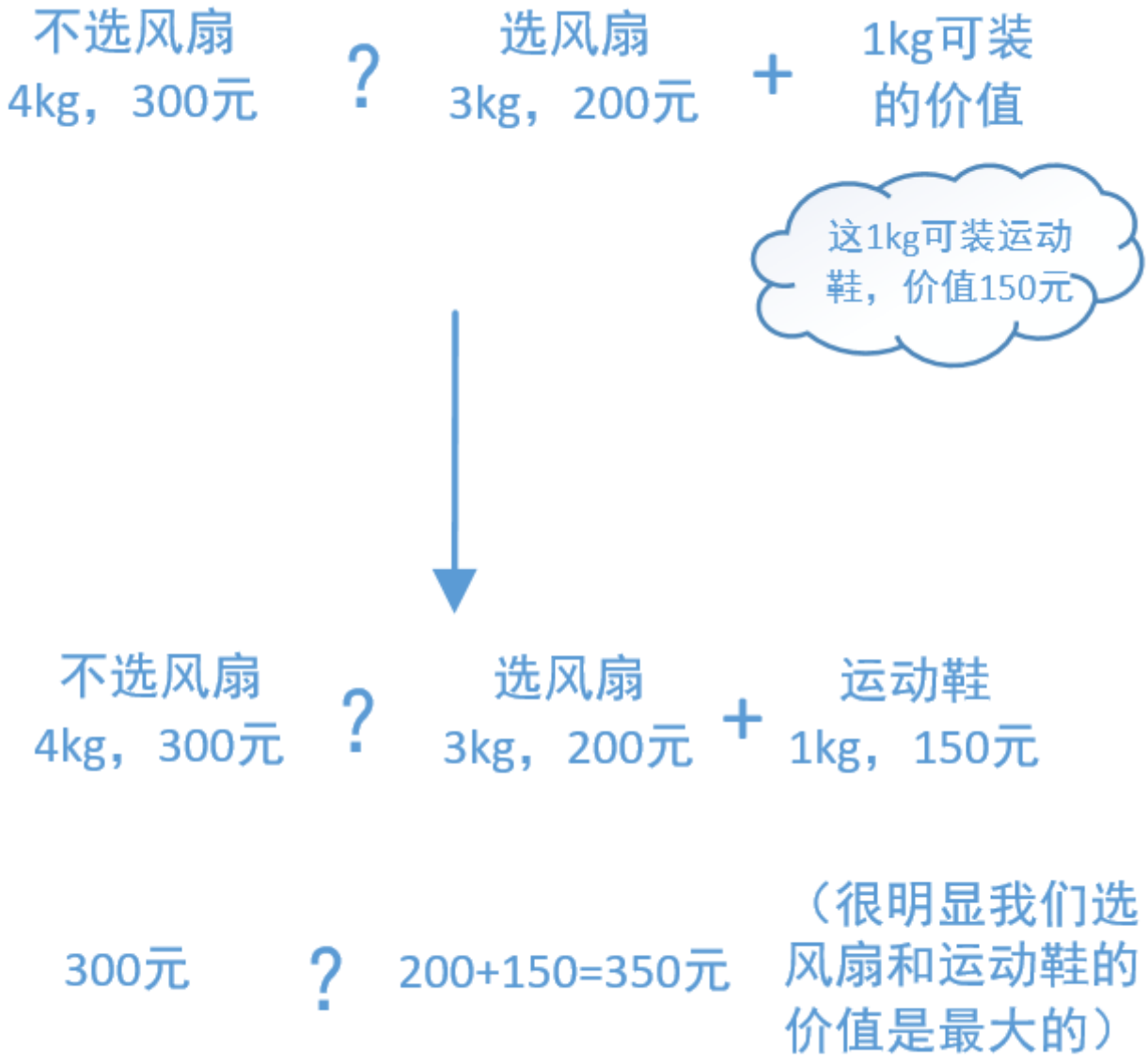

第二行我们可以选择高压锅了，因为1kg的容量我们装不下高压锅，只能装运动鞋，所以最大值还是150
包的容量
1kg
2kg
3kg
4kg
运动鞋
150
150
150
150
第三步
高压锅
150
风扇


同理当包的容量是2，3的时候还是装不下高压锅的（在这一行我们还没到风扇这一步，所以还不能选择风扇，只能选择运动鞋和高压锅），只能装运动鞋，所以最大值还是150
包的容量
1kg
2kg
3kg
4kg
运动鞋
150
150
150
150
第四步
高压锅
150
150
150
风扇


包的容量为4的时候我们可以装下高压锅了，如果装高压锅的话，最大价值是300元，比运动鞋的价值大，所以我们选择装高压锅
包的容量 1kg 2kg 3kg 4kg
运动鞋 150 150 150 150
第五步 高压锅 150 150 150 300
风扇


第三行我们可以3个都能选择了，当包容量为1，2的时候，我们只能选择运动鞋。但当包容量为3的时候我们可以选择风扇了，风扇的价值大于150，所以当包容量为3kg的时候我们选择风扇。
包的容量 1kg 2kg 3kg 4kg
运动鞋 150 150 150 150
第六步 高压锅 150 150 150 300
风扇 150 150 200


包的容量
1kg
2kg
3kg
4kg
第七步
运动鞋
高压锅
风扇
150
150
150
150
150
150
150
300
150
150
200
这一步是关键，如果我们没选风扇的话，4kg的最大价值是300元，如果可以选择风扇的话，3kg的最大价值是200元，但这一空格是可以选择风扇的时候，4kg的最大价值是多少？


不选风扇
4kg，300元
?
选风扇
3kg，200元
+
1kg可装的价值
这1kg可装运动鞋，价值150元
不选风扇
4kg，300元
?
选风扇
3kg，200元
+
运动鞋
1kg，150元
300元
?
200+150=350元
（很明显我们选风扇和运动鞋的价值是最大的）

所以递推公式我们很容易就能得出

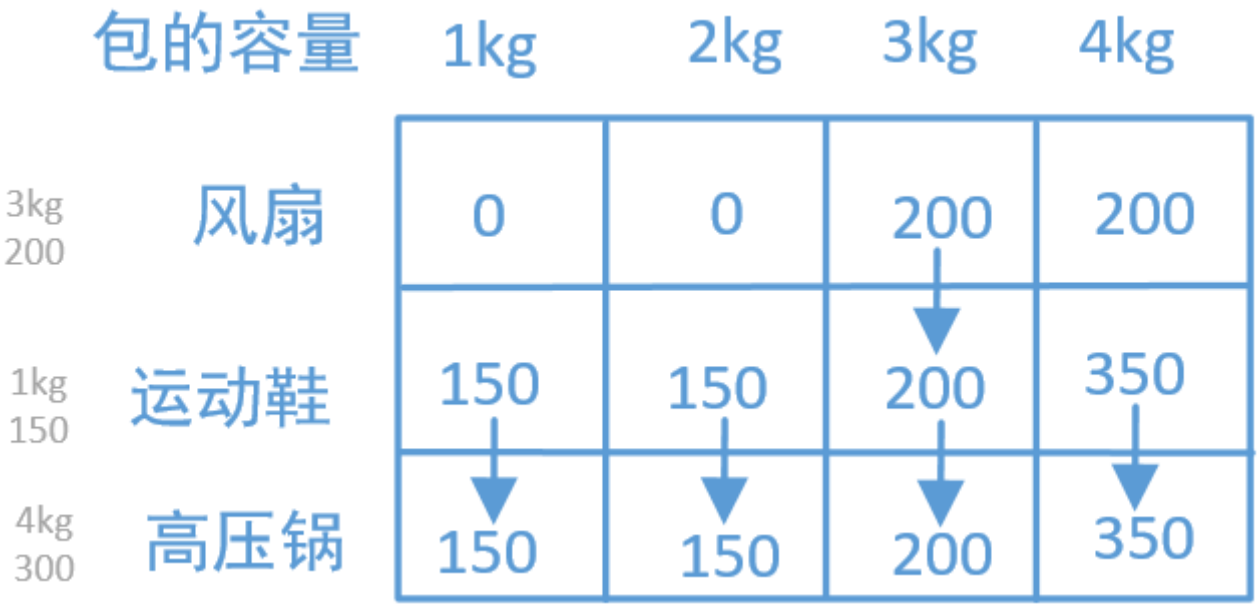
dp[i][j] = max
- dp[i-1][j]（不选当前商品）
- 当前商品的价值+剩余空间可容纳的价值(选当前商品)

转换成公式

dp[i][j] = max
- dp[i-1][j]（不选当前商品）
- dp[i-1][j-weight(i)]+value[i]（选当前商品）

02 改变选择的顺序

> 我们上面选择的顺序是：运动鞋→高压锅→风扇，如果我们改变选择的顺序，结果会不会改变，比如我们选择的顺序是：风扇→运动鞋→高压锅，我们还是来画个图看一下



> 我们发现无论选择顺序怎么改变都不会改变最终的结果。

数据测试：

> 我们就用上面的图形分析的数据来测试一下，看一下最终结果

```
public static void main(String[] args) {
   System.out.println("最终结果是: " + packageProblem1());
}

public static int packageProblem1() {
   int packageContainWeight = 4;//包最大可装重量
   int[] weight = {1, 4, 3};//3个物品的重量
   int[] value = {150, 300, 200};//3个物品的价值
   int[][] dp = new int[weight.length + 1][packageContainWeight + 1];
   for (int i = 1; i <= value.length; i++) {
      for (int j = 1; j <= packageContainWeight; j++) {
         if (j >= weight[i - 1]) {
            dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - weight[i - 1]] + value[i - 1]);
         } else {
            dp[i][j] = dp[i - 1][j];
         }
      }
   }
   Util.printTwoIntArrays(dp);//这一行仅做打印观测数据，也可以去掉
   return dp[weight.length][packageContainWeight];
}
```

03 运行结果

```
0   0   0   0   0
0   150 150 150 150
0   150 150 150 300
0   150 150 200 350
最终结果是: 350
```

> 和我们上面分析的完全一致。（为了测试方便，这里的所有数据我都是写死的，我们也可以把这些数据提取出来，作为函数参数传进来。）

空间优化：

其实这题还可以优化一下，这里的二维数组我们每次计算的时候都是只需要上一行的数字，其他的我们都用不到，所以我们可以用一维空间的数组来记录上一行的值即可，但要记住一维的时候一定要逆序，否则会导致重复计算。我们来看下代码

```
1   public static int packageProblem2() {
2      int packageContainWeight = 4;
3      int[] weight = {1, 4, 3};
4      int[] value = {150, 300, 200};
5      int[] dp = new int[packageContainWeight + 1];
6      for (int i = 1; i <= value.length; i++) {
7         for (int j = packageContainWeight; j >= 1; j--) {
8            if (j - weight[i - 1] >= 0)
9               dp[j] = Math.max(dp[j], dp[j - weight[i - 1]] + value[i - 1]);
10        }
11        Util.printIntArrays(dp);
12        System.out.println();
13     }
14     return dp[packageContainWeight];
15  }
```

注意：

我们看到第7行在遍历重量的时候采用的是逆序的方式，因为第9行在计算dp[j]的值的时候，数组后面的值会依赖前面的值，而前面的值不会依赖后面的值，如果不采用逆序的方式，数组前面的值更新了会对后面产生影响。

04 运行结果

```
0    150 150 150 150
0    150 150 150 300
0    150 150 200 350
最终结果是：350
```

C++:

```
1   #include<iostream>
2   #include <algorithm>
3
4   using namespace std;
5
6   int main()
7   {
8      int weight[] = { 1,4,3 };
9      int value[] = {150, 300, 200 };
10     int packageContainWeight = 4;
```

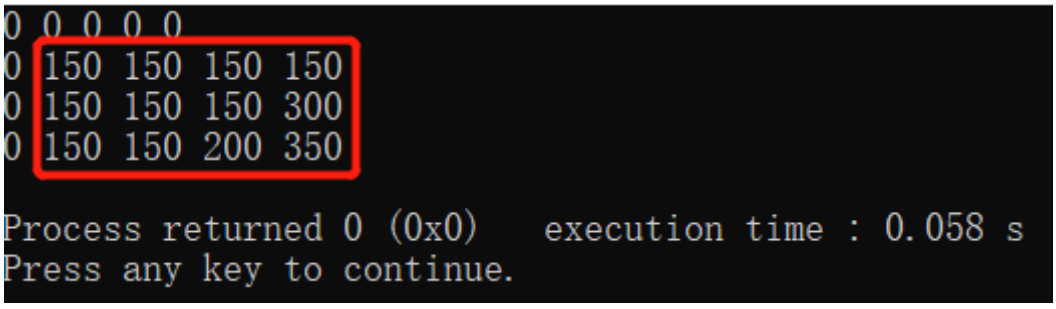
```
    int dp[4][5]= { { 0 } };
    for (int i = 1; i <4 ; i++)
    {
      for (int j = 1; j < 5; j++)
      {
        if (j >= weight[i - 1])
          dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i - 1]] + value[i - 1]);
        else
          dp[i][j] = dp[i - 1][j];
      }
    }

    for (int i = 0; i < 4; i++)
    {
      for (int j = 0; j < 5; j++)
      {
        cout << dp[i][j] << ' ';
      }
      cout << endl;
    }

    return 0;
  }

```

05 运行结果



递归写法：

除了上面的两种写法以外，我们还可以使用递归的方式，代码中有注释，有兴趣的可以自己看，就不在详细介绍。

```
int[] weight = {1, 4, 3};//3个物品的重量
int[] value = {150, 300, 200};//3个物品的价值

// i: 处理到第i件物品，j可容纳的重量
public int packageProblem3(int i, int j) {
    if (i == -1)
        return 0;
    int v1 = 0;
    if (j >= weight[i]) {//如果剩余空间大于所放的物品
        v1 = packageProblem3(i - 1, j - weight[i]) + value[i]; //选第i件
```

```
11      }
12      int v2 = packageProblem3(i - 1, j);//不选第i件
13      return Math.max(v1, v2);
14    }
```

长按上图，识别图中二维码之后即可关注。

如果喜欢这篇文章就点个"在看"吧

收录于合集 #算法图文分析 164

上一篇
372，二叉树的最近公共祖先

下一篇
370，最长公共子串和子序列