40|初识动态规划：如何巧妙解决“双十一”购物时的凑单问题？

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。

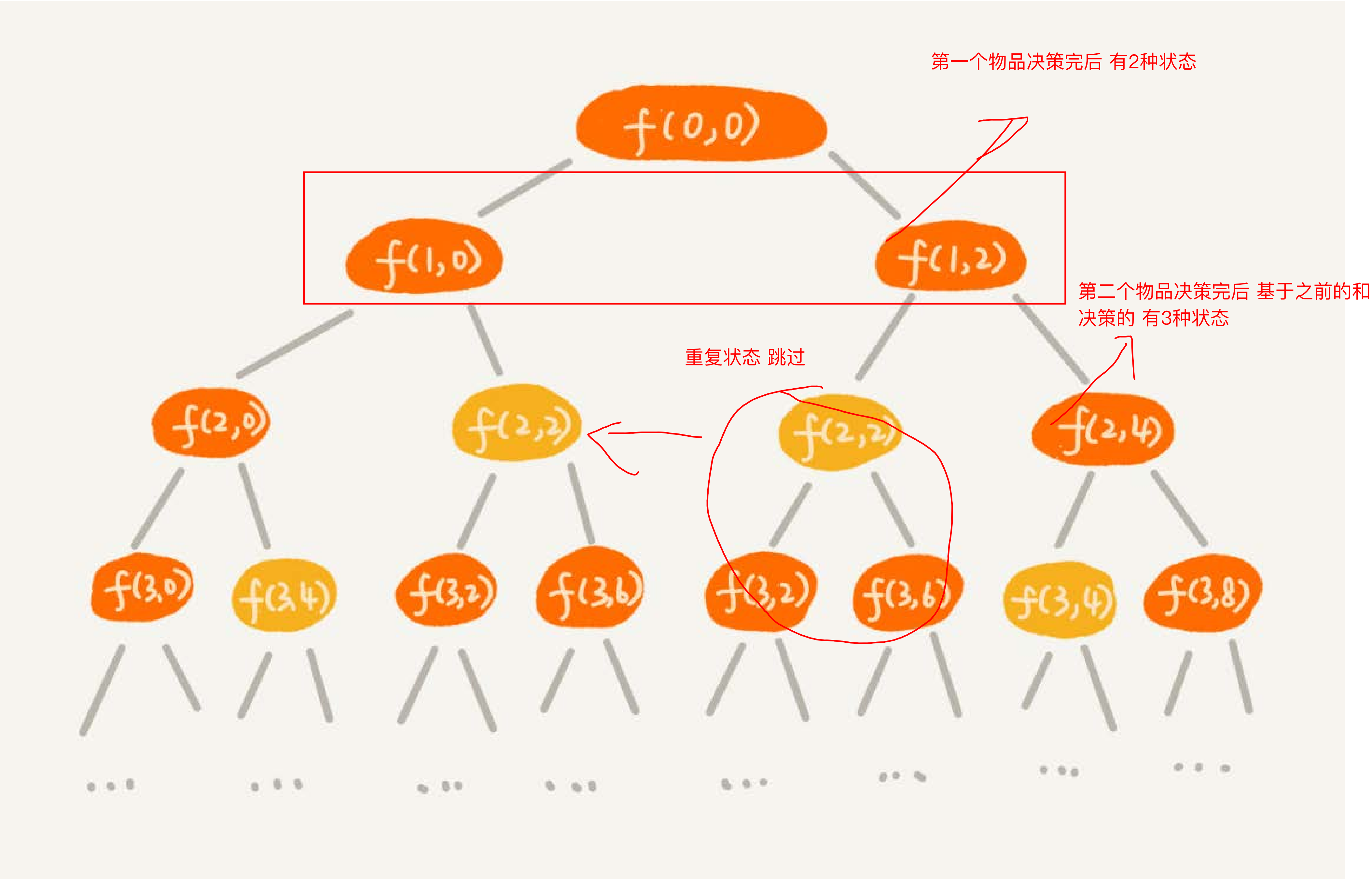
对于一组不同重量、不可分割的物品，我们需要选择一些装入背包，在满足背包最大重量限制的前提下，背包中物品总重量的最大值是多少呢？

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。

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

```
    f(i+1,cw + weight[i]); // 选择装第i个物品
  }
}
```

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。

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

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。

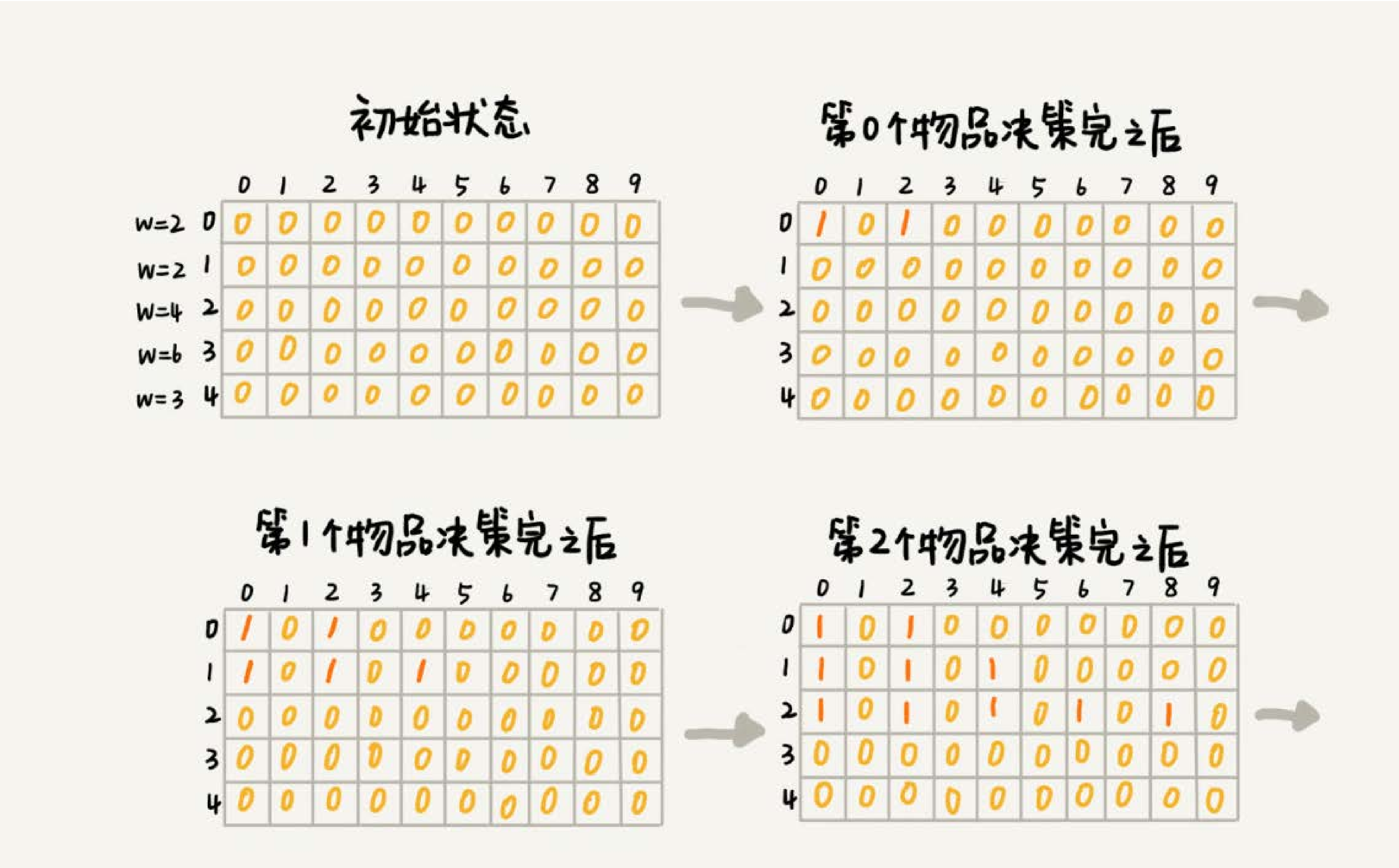
我们把每一层重复的状态（节点）合并，只记录不同的状态，然后基于上一层的状态集合，来推导下一层的状态集合。我们可以通过合并每一层重复的状态，这样就保证每一层不同状态的个数都不会超过w个（w表示背包的承载重量），也就是例子中的9。于是，我们就成功避免了每层状态个数的指数级增长。

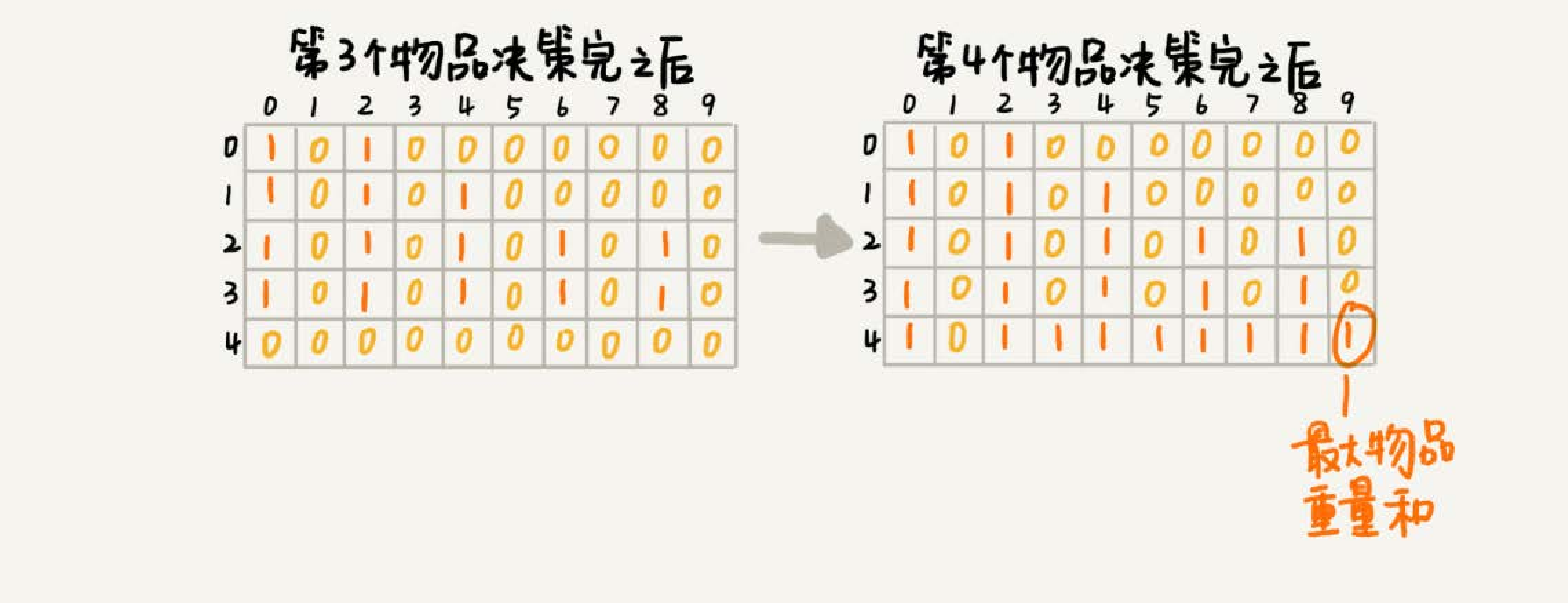
我们用一个二维数组states[n][w+1]，来记录每层可以达到的不同状态。

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 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。


初始状态
第0个物品决策完之后
第1个物品决策完之后
第2个物品决策完之后
w=2
w=2
w=4
w=6
w=3



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。

```
weight:物品重量，n:物品个数，w:背包可承载重量
public int knapsack(int[] weight, int n, int w) {
  boolean[][] states = new boolean[n][w+1]; // 默认值false
  states[0][0] = true;  // 第一行的数据要特殊处理，可以利用哨兵优化
  states[0][weight[0]] = true;
  for (int i = 1; i < n; ++i) { // 动态规划状态转移
    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;
    }
  }
  for (int i = w; i >= 0; --i) { // 输出结果
    if (states[n-1][i] == true) return i;
  }
  return 0;
}
```

stage n个物品对应n个阶段 可能有w+1种状态(每一个可能的重量对应一个状态)

初始化第0行

输出最接近w的

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？

O(2^n)

前面我们讲到，用回溯算法解决这个问题的时间复杂度 ，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。

空间复杂度优化

实际最后只需要最后一层的状态

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

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？

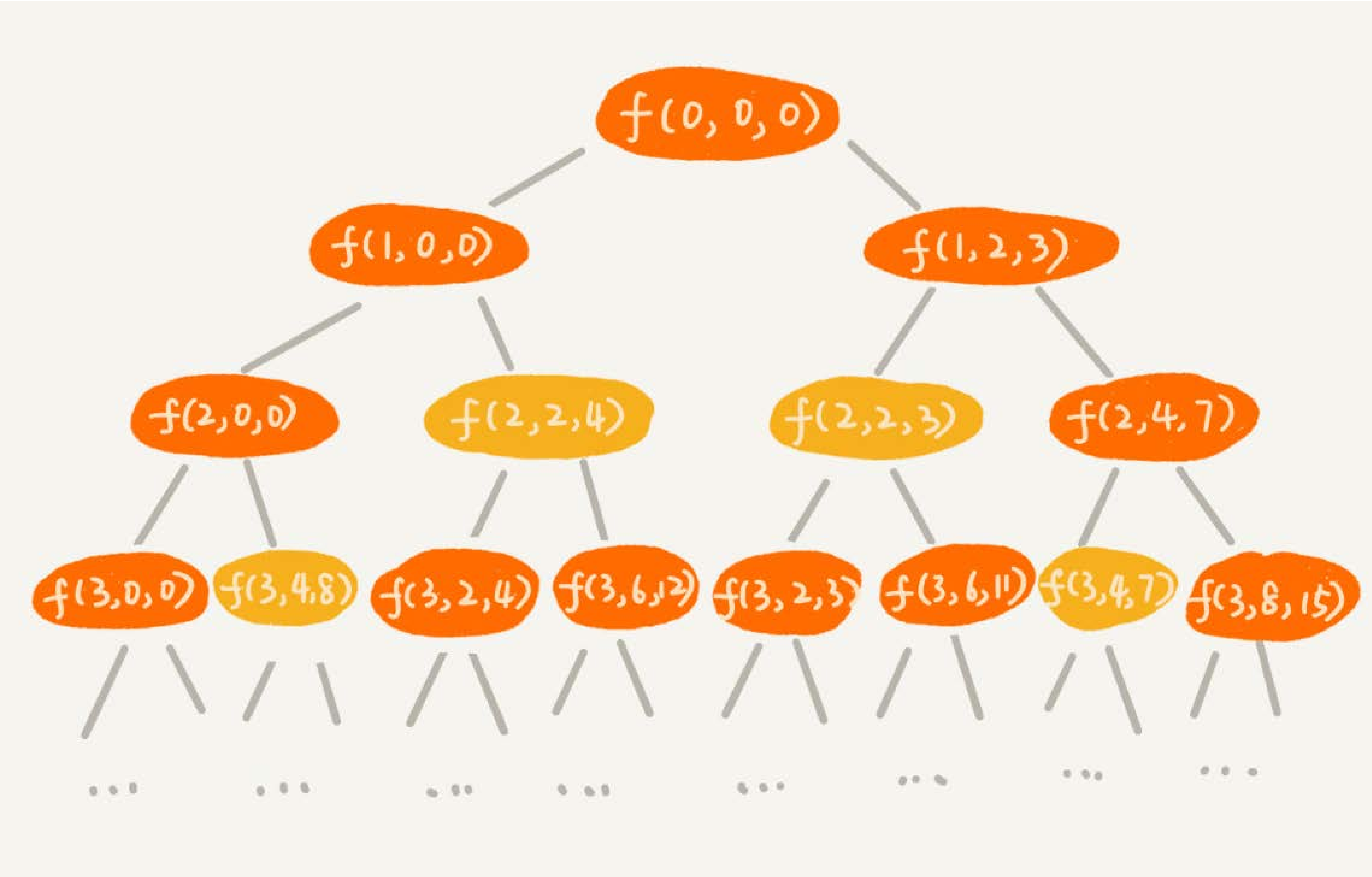
我们刚刚讲的背包问题，只涉及背包重量和物品重量。我们现在引入物品价值这一变量。对于一组不同重量、不同价值、不可分割的物品，我们选择将某些物品装入背包，在满足背包最大重量限制的前提下，背包中可装入物品的总价值最大是多少呢？

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。

```
private int maxV = Integer.MIN_VALUE; // 结果放到maxV中
private int[] items = {2，2，4，6，3}; // 物品的重量
private int[] value = {3，4，8，9，6}; // 物品的价值
private int n = 5; // 物品个数
private int w = 9; // 背包承受的最大重量
public void f(int i, int cw, int cv) { // 调用f(0, 0, 0)
  if (cw == w || i == n) { // cw==w表示装满了，i==n表示物品都考察完了
```

```
    if (cv > maxV) maxV = cv;
    return;
  }
  f(i+1, cw, cv); // 选择不装第i个物品
  if (cw + weight[i] <= w) {
    f(i+1,cw+weight[i], cv+value[i]); // 选择装第i个物品
  }
}
```

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：

```
public static int knapsack3(int[] weight, int[] value, int n, int w) {
  int[][] states = new int[n][w+1];
  for (int i = 0; i < n; ++i) { // 初始化states
    for (int j = 0; j < w+1; ++j) {
      states[i][j] = -1;
    }
  }
  states[0][0] = 0;
  states[0][weight[0]] = value[0];
  for (int i = 1; i < n; ++i) { //动态规划，状态转移
    for (int j = 0; j <= w; ++j) { // 不选择第i个物品
      if (states[i-1][j] >= 0) states[i][j] = states[i-1][j];
    }
    for (int j = 0; j <= w-weight[i]; ++j) { // 选择第i个物品
      if (states[i-1][j] >= 0) {
        int v = states[i-1][j] + value[i];
        if (v > states[i][j+weight[i]]) {
          states[i][j+weight[i]] = v;
        }
      }
    }
  }
  // 找出最大值
  int maxvalue = -1;
  for (int j = 0; j <= w; ++j) {
    if (states[n-1][j] > maxvalue) maxvalue = states[n-1][j];
  }
  return maxvalue;
}
```

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。

```
// items商品价格，n商品个数, w表示满减条件，比如200
public static void double11advance(int[] items, int n, int w) {
  boolean[][] states = new boolean[n][3*w+1];//超过3倍就没有薅羊毛的价值了
  states[0][0] = true;  // 第一行的数据要特殊处理
  states[0][items[0]] = true;
  for (int i = 1; i < n; ++i) { // 动态规划
    for (int j = 0; j <= 3*w; ++j) {// 不购买第i个商品
      if (states[i-1][j] == true) states[i][j] = states[i-1][j];
    }
    for (int j = 0; j <= 3*w-items[i]; ++j) {//购买第i个商品
      if (states[i-1][j]==true) states[i][j+items[i]] = true;
    }
  }

  int j;
  for (j = w; j < 3*w+1; ++j) {
    if (states[n-1][j] == true) break; // 输出结果大于等于w的最小值
  }
  if (j == -1) return; // 没有可行解
  for (int i = n-1; i >= 1; --i) { // i表示二维数组中的行，j表示列
    if(j-items[i] >= 0 && states[i-1][j-items[i]] == true) {
      System.out.print(items[i] + " "); // 购买这个商品
      j = j - items[i];
    } // else 没有购买这个商品，j不变。
  }
  if (j != 0) System.out.print(items[0]);
}
```

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

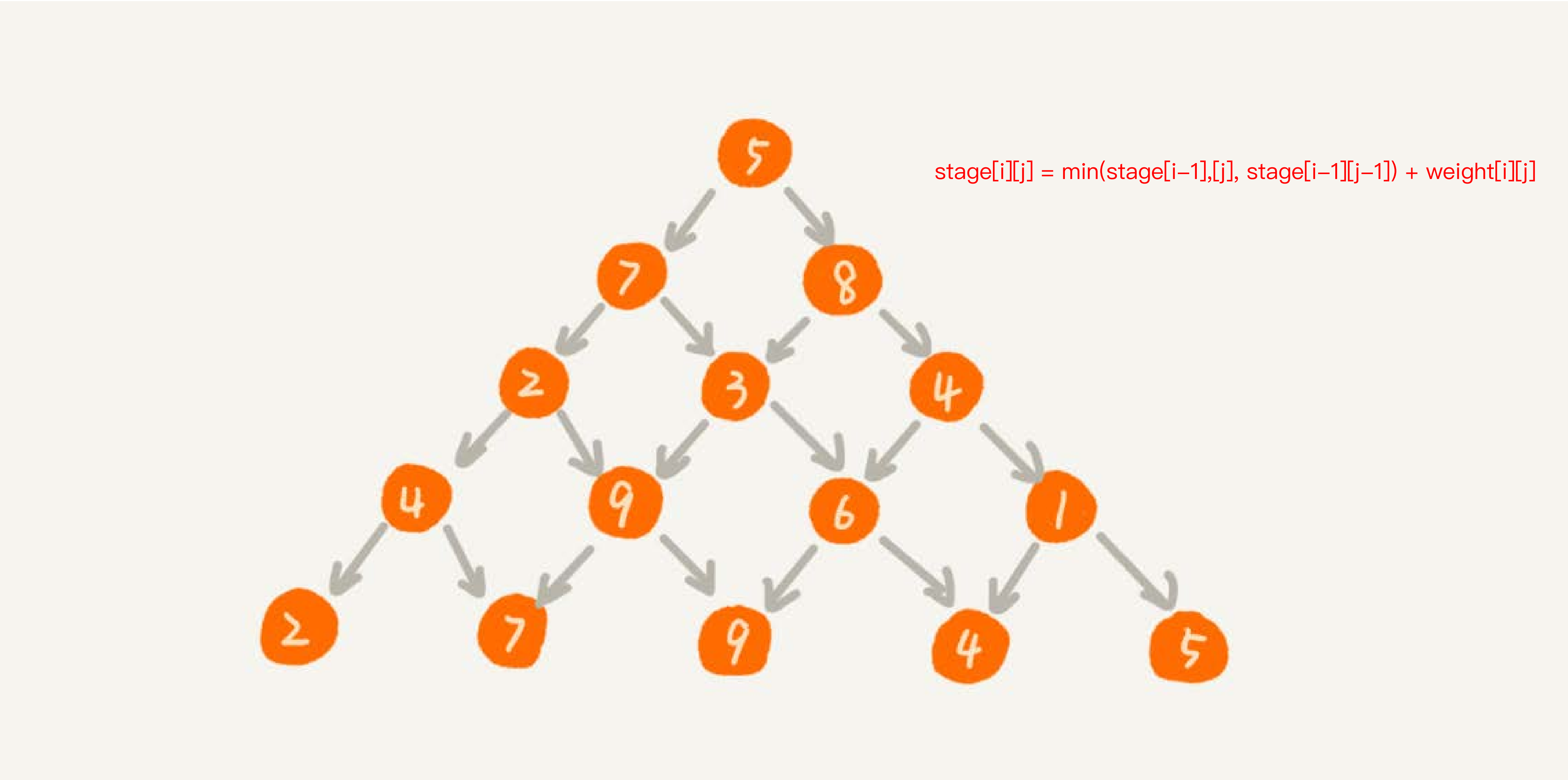
算法思想。

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，然后才用算法理论知识，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。

精选留言：

- zixuan 2018-12-30 09:57:47
  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) [17赞]

- 茴香根 2018-12-26 00:44:18
  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完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。 [14赞]

- hfy 2018-12-26 00:36:57
  首先得有个女朋友 [8赞]

- feifei 2018-12-28 08:29:55
  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！

  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);
```

[5赞]

- Monday 2018-12-28 12:25:33

  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值。

  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。
  [4赞]

  作者回复2019-01-02 08:37:54
  不着急你慢慢学就是了 不用非得跟的那么紧

- Andylee 2018-12-26 05:18:51

  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 [4赞]

  作者回复2018-12-26 12:05:16
  是的 我改下

- P@tricK 2018-12-26 01:12:09

  老师你这个只能精确到元，女朋友羊毛精说要求精确到0.01元，时间空间复杂度增大100倍 [4赞]

  作者回复2018-12-26 12:05:55
  说的没错

- 郭霖 2019-01-02 09:14:00

  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;
} [3赞]
```

- 煦暖 2018-12-28 15:33:46
  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？ [3赞]
- 失火的夏天 2018-12-27 06:28:30
  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。 [2赞]