初识动态规划：如何解决双十一购物时的凑单问题

双十一，总有很多比如满200减50的活动，假设女朋友购物车中有n个（n>100)商品，在凑够满减条件的前提下，让选出的商品价格总和最大程度上接近满减条件200元，如何使用程序搞定呢?

动态规划学习路线

动态规划适合求解最优问题，比如最大值，最小值等等;

三部分学习： 第一节通过两个经典问题展示为什么要使用动态规划以及动态规划解题方法是如何演化过来的?

第二节总结动态规划适合解决问题的特征，以及动态规划解题思路;

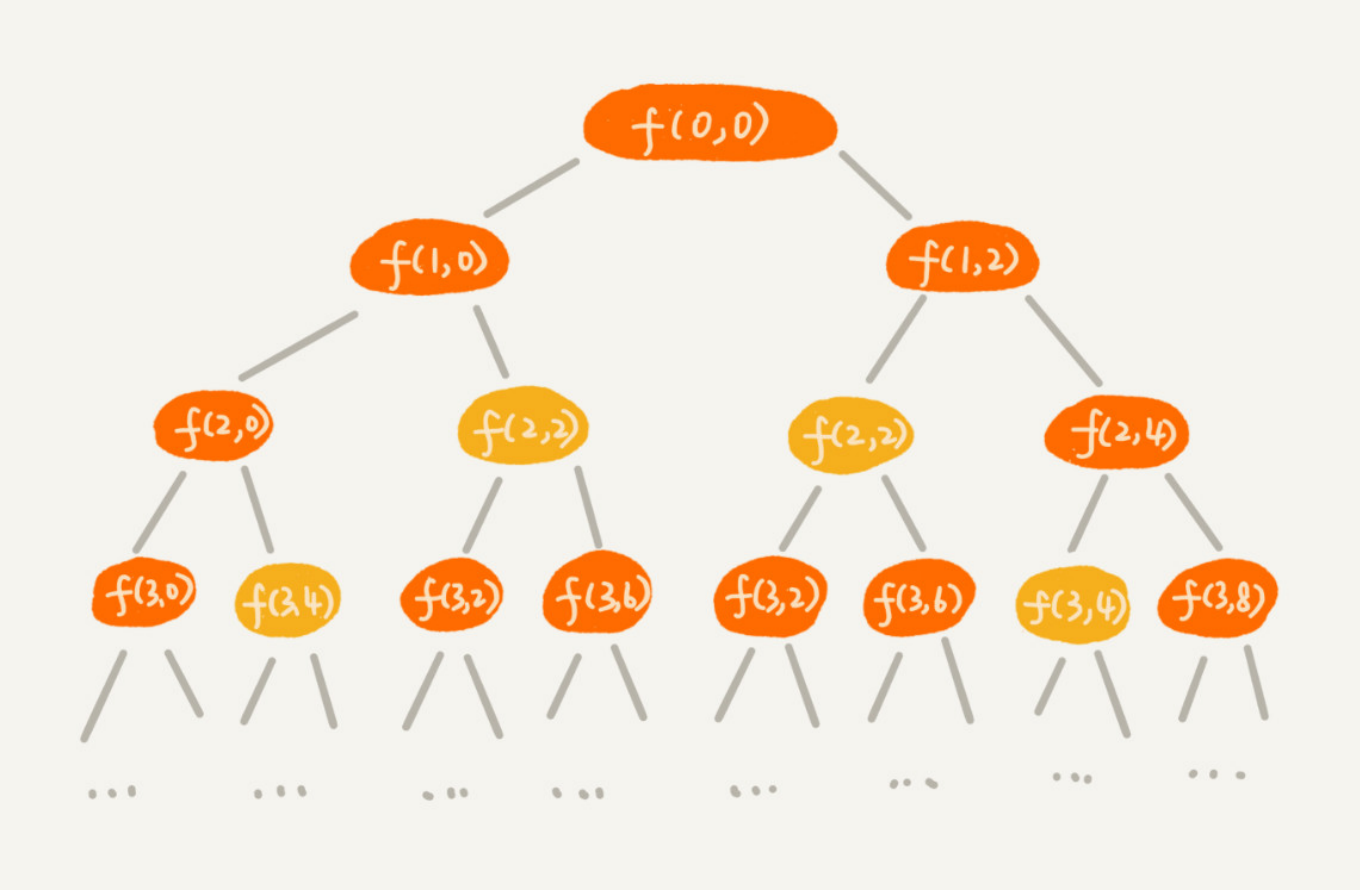
第三节动态规划的实际应用;

0-1背包问题

贪心算法中的背包问题物品是可以分割的，回溯算法中物品不可以分割，只是回溯算法中我们要穷举所有可能的方法，时间复杂度是指数级的，回溯算法的实现方法如下:

```
private int maxW = Integer.MIN_VALUE; //结果放在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; //背包容量
public void f(int i,int cw){ //调用方法f(0,0)
    if(cw == w || i == n){
        if(cw > maxW) maxW = cw;
        return;
    }
    f(i+1,cw);//选择不装第i件物品
    if(cw + weight[i] <= w){
        f(i+1,cw + weight[i]); //选择装下第i件物品
    }
}
```

将上述的过程画成递归树的形式就是：



其中每一个节点表示一种状态，(i,cw)表示，i表示第i件物品是否装入背包，cw表示当前背包中物品的总重量。

在上述图中发现有些子问题是重复的，可以借助递归那一节讲的备忘录，将计算好的f(i,cw)存储起来，下次直接取用，改造之后的代码是：

```
private int maxW = Integer.MIN_VALUE;
private int[] weight = {2,2,4,6,3};
private int n = 5;
private int w = 9;
private boolean[][] mem = new boolean[5][10];//5个物品，10个重量

public void f(int i,int cw){
    if(cw == w || i == n){
        if(cw > maxW){
            maxW = cw;
        }
    }
    if(mem[i][cw]) return; //判断是否是重复状态
    mem[i][cw] = true; //及时记录已经访问过的状态
    f(i+1,cw);//第i个物品不放入
    if(cw + weight[i] <= w){
        f(i+1,cw + weight[i]);//第i个物品放入
    }
}
```

这种解决方法已经和动态规划执行效率差不多了。

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入背包，每个物品的决策之后，背包中的物品重量会有很多情况，也就是说会达到多种不同的状态，对应到递归树中有很多不同的节点；

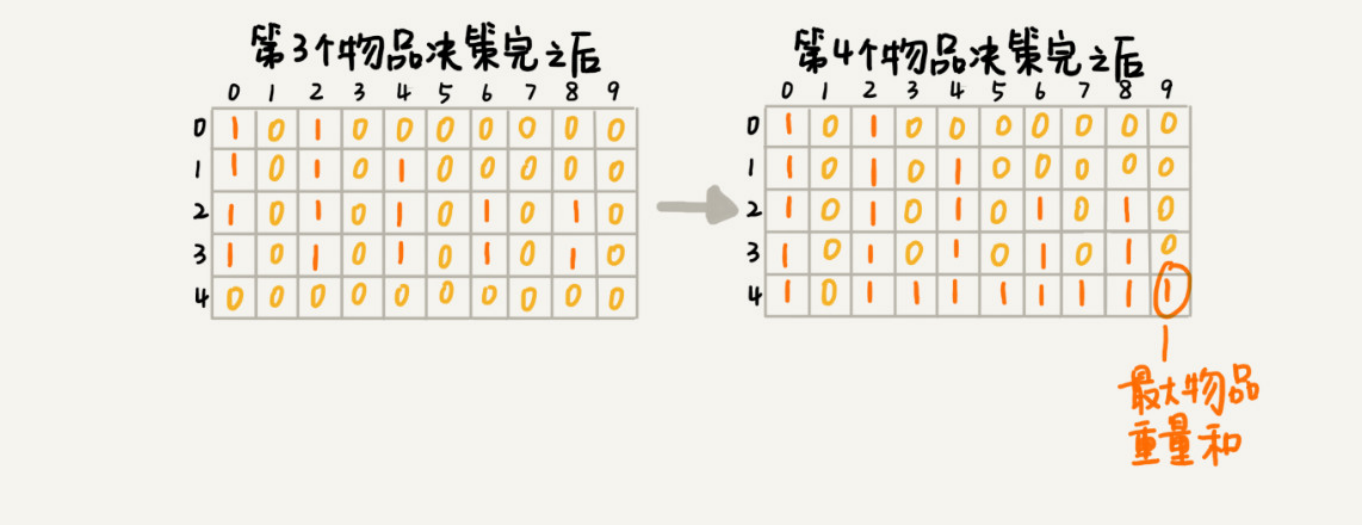
我们把每一层的重复状态进行合并，只记录不同的状态，然后基于上一层的状态集合，推导下一层的状态集合，通过合并每一层的状态，保证了每一层不同状态的个数不会超过w个；

我们使用一个二维数组states[n][w+1],来记录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；

以此类推考察完所有的物品之后，整个states数组就计算好了，整个过程如图所示:



上述的过程比较翻译成代码就是：

```
public int knapsack(int[] weight,int n,int w){
    boolean[][] states = new boolean[n][w+1]; //默认为false
    states[0][0] = true; //第一行数据特殊处理
    if(weight[0] <= w){
        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件物品放入
            if(states[i-1][j]){
                states[i][j] = states[i-1][j];
            }
        }
        for(int j = 0;j < w - weight[i];j++){//把第i件物品放入
            if(states[i-1][j]){
                states[i][j+weight[i]] = true;
```

```
            }
        }
    }
    for(int i = w; i >= 0; i--){
        if(states[n-1][i]){
            return i;
        }
    }
    return 0;
}
```

实际上这是一种记录每一个阶段可达的状态集合，然后通过当前阶段的动态集合，来推导下一阶段的状态集合；

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。

如果物品很多，怎么办？ 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;

改进方法使用一个一维数组就可以了：

```
public int knapsack1(int[] weight,int n,int w){
    boolean[] states = new boolean[w+1]; //默认为false
    states[0] = true; //第一行数据特殊处理
    if(weight[0] <= w){
        states[weight[0]] = true;
    }
    for(int i = 1; i < n; i++){ //动态规划状态转移
        for(int j = w - weight[i]; j >= 0;--j) { //把第i件物品放入
            if (states[j]) {
                states[j + weight[i]] = true;
            }
        }
    }
    for(int i = w; i >= 0; i--){
        if(states[i]){
            return i;
        }
    }
    return 0;
}
```

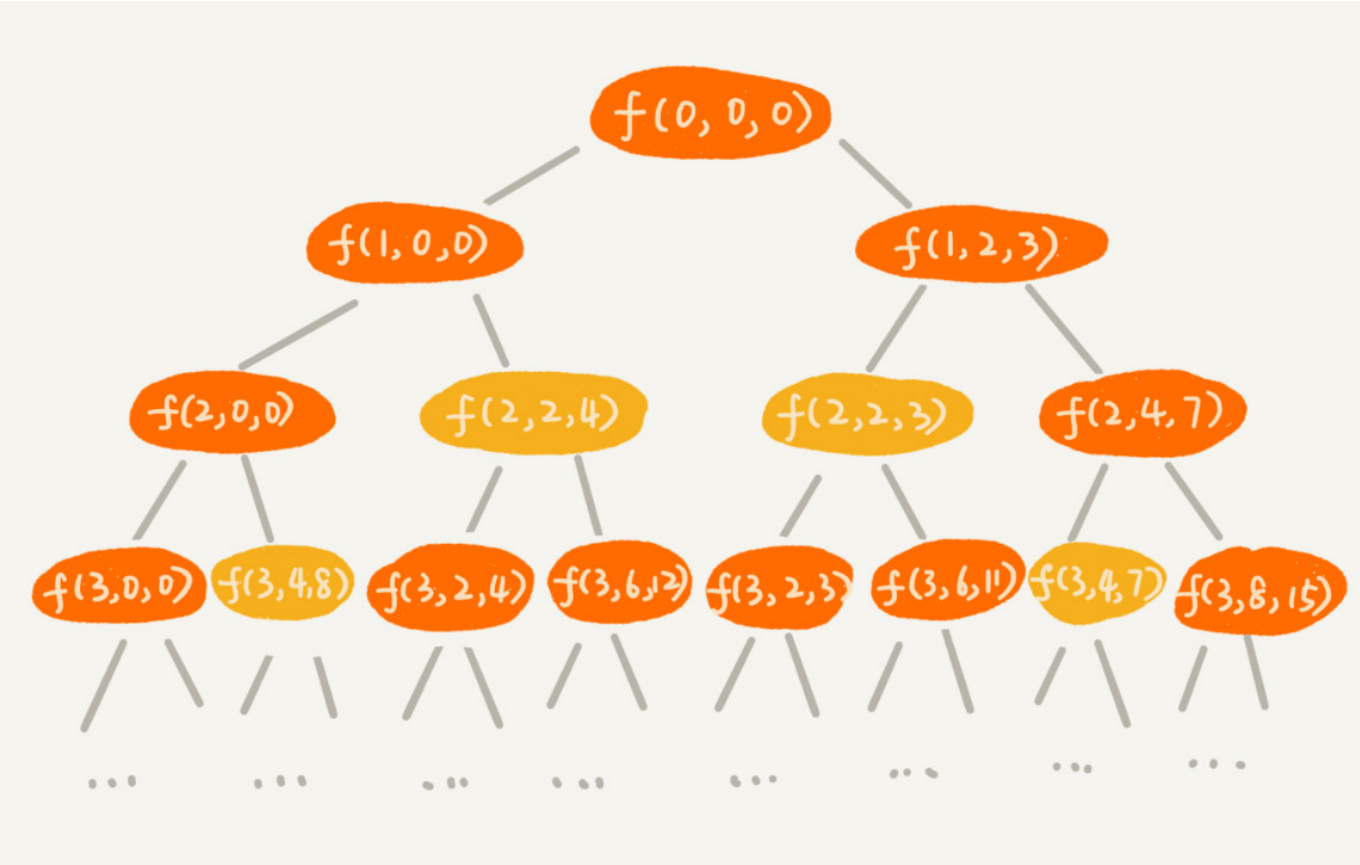
0-1背包问题升级版

如果在上述问题的基础上引入物品价值这一变量，在满足最大重量的前提下，背包中可以装入的总价值最大是多少呢?

```
private int maxV = Integer.MIN_VALUE;
private int[] items = {2,2,4,6,3};
private int[] value = {3,4,8,9,6};
private int n = 5;
private int w = 9;

/**
 * 回溯算法
 * @param i
 * @param cw
 * @param cv
 */
public void f(int i,int cw,int cv){
    if(cw == w || i == n){
        if(cv > maxV){
            maxV = cv;
        }
    }
    f(i+1,cw,cv);
    if(cw + items[i] <= w){
        f(i + 1,cw + items[i],cv + value[i]);
    }
}
```

画出相应的递归树为：



在上述的过程中存在很多的重复节点，我们每次都在记录相应的状态集；

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。

```
public static int knapsack3(int[] weight,int[] value,int n,int w){
    int[][] states = new int[n][w+1];
    //初始化
    for(int i = 0; i < n; i++){
        for(int j = 0; j < w + 1;j++){
            states[i][j] = -1; //此时记录的状态图应该是价格值，并且相同的情况取最大价格
        }
    }
    states[0][0] = 0;
    if(weight[0] <= w){
        states[0][weight[0]] = value[0];
    }
    for(int i = 1; i < n; i++){
        for(int j = 0; j <= w; j++){
            if(states[i-1][j] >= 0){
                states[i][j] = states[i-1][j];//不放入i物品
            }
        }
        for(int j = 0; j < w-weight[i]; j++){
            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;
}
```

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

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。

实际上是0-1背包的变形问题，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。就这个实际问题而言，如果要买的东西比200多很多，就没有了意义；

其次就是还有找到相应的物品是那些；

```
/**
 * 双十一凑单
 * @param items
 * @param n
 * @param w
 */
public static void double11advance(int[] items,int n,int w){
    boolean[][] states = new boolean[n][3*w+1];
    states[0][0] = true;
    if(items[0] <= 3*w){
        states[0][items[0]] = true;
    }
    for(int i = 1; i < n; i++){
        for(int j = 0; j <= 3*w;j++){
            if(states[i-1][j]) {
                states[i][j] = states[i-1][j];
            }
        }
        for(int j = 0; j < 3*w-items[i]; j++){
            if(states[i-1][j]){
                states[i][j+items[i]] = true;
            }
        }
    }

    int j;
    for(j = w; j < 3*w+1;j++){ //从w开始，输出结果大于w的最小值
        if(states[n-1][j]){
            break;
        }
    }
    if(j == 3*w+1) return;//没有可行解
    for(int i = n-1;i >= 1;--i){
        if(j-items[i] >= 0 && states[i-1][j-items[i]]){
            System.out.println(items[i] + " ");
            j = j-items[i];
        }
    }
    if(j != 0){
        System.out.println(items[0]);
    }
}
```

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。

课后思考

杨辉三角； 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。