写在前面:因为文档附上代码后有点长，请点击左下角的导航按钮，点击标题直达相应题目的位置。



15.1

题目：

采用DP思想设计算法，判断一个n元正整数集合A中是否存在一个子集B,B中所有元素之和为一个给定的正整数S。

设计思想：

因为题目要求是判断是否存在，所以我们采用一个二维bool数组来维护解的迭代。容易发现，如果我们将所有前i个元素(1≤i≤n)的可能组成的和写出，那么我们就可以判断这n个元素的自然数系数(取值仅限0或1)的线性组合张成的空间中是否有一维向量S了。

因此我们可以这样设计这个二维bool数组T。二维数组的行是0~S的所有可能值，二维数组的列为0,1,2,3,…,n。T(P,i)代表的含义为:前i个元素的自然数系数的线性组合张成的空间中是否有P这个一维向量。

接下来考虑DP中最重要的一环，即状态转移方程。显然，对于任意P(1≤P≤S)而言，T(P,i)是否为真都有三种情况。

1. 第一种，前(i-1)个元素已经构成了P，即T(P,i-1)为真；
2. 第二种，第i个元素自身就可以构成P，即ai=P；
3. 第三种，前(i-1)个元素可以构成P-ai，这样算上第i个元素ai，就可以构成P了。即，T(P-ai,i-1)为真。

T(P,i)为真当且仅当以上三种情况中至少存在一种为真。那么可以得到状态转移方程：

最后，我们采用bottom-up的思想，从前1个元素能构成的所有可能值开始不断迭代，直到我们获得一组可行解，即前k个元素能构成的所有可能值中存在S。

另外，我们需要给出一种可以回溯得到解的方式。其实就是利用状态转移方程不断检查已经得到的二维数组，检查S及其前置的和的构成方式是三种情况的哪一种，并向解向量中更新对应的结果即可。

设计思想图解：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| ***P/i*** | 0 | 1 | 2 | … | n |
| 0 | True | True | True | True | True |
| 1 | False | … |  |  |  |
| 2 | False |  |  |  |  |
| … | False |  |  |  |  |
| S | False |  |  |  |  |

伪代码：

Judge() // To judge whether there exists a solution

{

Firstly sort the collection A in an **ascending order** // It’s not necessary but I’d like to

For each column in the DP table

For each row in the DP table

if current row is the last row and T(P,i) is true

return true // strategy to terminate timely

return false

}

TraceBack() // To get the final solution

{

If Judge() returns false // To save our time since there’s no solution

return

For each column as i in DP table as T

if T(sum,i) is false

continue // find the first column that gains the result

if = sum

// the solution vector is a bool vector

return // we still needs the rest of sum and happens to have it

if T(sum-,i-1) is true

// get two 1-coefficient at one time!

sum -= // update the rest of sum

continue

// this following one is unnecessary but we still keep it for a better understanding

if T(sum,i-1) is true

}

复杂度分析：

先对迭代部分分析。

时间复杂度：

初始化二维表：θ(sum\*n)。

迭代：因为及时终止了迭代过程，所以为O(sum\*n)。

总计： θ(sum\*n)。

空间复杂度：

维护了一个二维表，θ(sum\*n)。

再对回溯部分分析。

时间复杂度：

初始化解向量：θ(n)。

构建解向量：θ(n)。

总计：θ(n)。

空间复杂度：

维护了一个解向量，θ(n)。

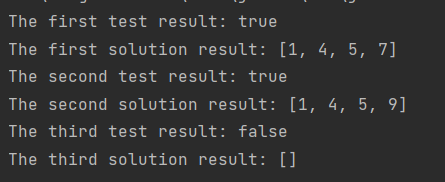
整个问题总计：

时间复杂度：θ(sum\*n)。

空间复杂度：θ(sum\*n)。

编码分析：

采用java语言编写。其中包含了测试部分，经检测结果正确。



代码具体如下：

package com.DP.allocatedSumForSubSet;  
  
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.List;  
  
*/\*\*  
 \* 复杂度上:  
 \* 对于迭代而言:  
 \* 初始化: θ(n\*sum)  
 \* 迭代: O(sum\*n)  
 \* 因此总体为: θ(sum\*n)  
 \* 对于回溯解而言:  
 \* 初始化: θ(n)  
 \* 回溯: θ(n)  
 \* 总体为: θ(n)  
 \*/*public class JudgeWhetherCouldFormTheAllocatedSum {  
 //我不想采取类字段的形式将该类封装为一个单纯的工具类，需要为每一个实例创建对象加以判断  
 private final int[] A;//给定的正整数集合A  
 private final int sum;//给定的和S  
 private boolean[][] DPTable;//用来迭代更新的DP2维数组  
 private boolean[] solution;  
  
 public JudgeWhetherCouldFormTheAllocatedSum(int[] A,int sum){  
 //先按照升序排列一下  
 Arrays.*sort*(A);  
 this.A=A;  
 this.sum=sum;  
 initDPTable();  
 initSolution();  
 }  
  
 private void initDPTable(){  
 this.DPTable = new boolean[sum+1][A.length];  
 for (boolean[] rows : DPTable) {  
 for (boolean grid : rows) {  
 grid = false;  
 }  
 }  
 for (boolean firstRow : DPTable[0]) {  
 firstRow = true;  
 }  
 }  
  
 private void initSolution(){  
 this.solution = new boolean[A.length];  
 for (boolean b : solution) {  
 b=false;  
 }  
 }  
  
 */\*\*  
 \* 状态转移方程:  
 \* T(P,i) = T(P,i-1) || (ai==P) || T(P-ai,i-1) (注意ai范围防止数组越界)  
 \* ∑是累or的意思  
 \** ***@return*** *是否能得到给定和  
 \*/* public boolean Judge(){  
 //应不断迭代列,i为列,P为行  
 for(int i=1;i<=A.length-1;i++){ //A.length为(n+1),规定A零索引弃用  
  
 for(int P=1;P<=sum;P++){  
 //第一个是前一列能不能形成P，第二个是第i个元素能否形成P，第三个是由当前元素和前一列能够形成的所有数中的某一个组合能否形成P  
 DPTable[P][i] = DPTable[P][i-1] || (A[i]==P) || JudgeForeColumnCombineWithCurrentNumberForAllocatedSum(P,i);  
 if(P==sum&& DPTable[P][i]){  
 return true;  
 }  
 }  
  
 }  
 return false;  
 }  
  
 */\*\*  
 \* T(P,i)的形成可能是由当前元素和前一列能够形成的所有数中的某一个组合形成的  
 \** ***@param*** *P 给定的和(local)  
 \** ***@param*** *i 给定的元素索引  
 \** ***@return*** *能否组成给定和P  
 \*/* private boolean JudgeForeColumnCombineWithCurrentNumberForAllocatedSum(int P,int i){  
 if(A[i]>P){  
 return false;  
 }  
 else if(A[i]==P){  
 return true;  
 }  
 else{  
 return DPTable[P-A[i]][i-1];  
 }  
 }  
  
 public List<Integer> getSolution(){  
 generateSolution();  
 ArrayList<Integer> result = new ArrayList<>();  
 for(int i=1;i<=A.length-1;i++){  
 if (solution[i]){  
 result.add(A[i]);  
 }  
 }  
 return result;  
 }  
  
 private void generateSolution(){  
 //没解直接返回  
 if(!Judge()){  
 return;  
 }  
 int tempSum = sum;  
 //检查DPTable的相邻两列  
 for(int i=A.length-1;i>=2;i--){  
 //找到第一个达成sum的列  
 if(!DPTable[tempSum][i]){  
 continue;  
 }  
 //还需要tempSum,结果ai就是tempSum，直接返回了  
 if(A[i]==tempSum){  
 solution[i]=true;  
 return;  
 }  
 //tempSum是由ai和前i-1项的自然数系数的线性组合组合成的  
 if(DPTable[tempSum-A[i]][i-1]){  
 solution[i]=true;  
 solution[i-1]=true;  
 tempSum-=A[i];  
 continue;  
 }  
 //这一段其实可以不用写，但是为了表意，还是写了，意思就是前i-1个的自然数系数的线性组合已经组成了tempSum  
 if(DPTable[tempSum][i-1]){  
 solution[i]=false;  
 }  
 }  
 }  
  
 public static void main(String[] args) {  
 //A test for the DP code  
 //supposed to be {true,true,false}  
 int[] A = {-1,1,4,5,7,9};  
 int S = 17;  
 JudgeWhetherCouldFormTheAllocatedSum judgeWhetherCouldFormTheAllocatedSum = new JudgeWhetherCouldFormTheAllocatedSum(A, 17);  
 System.*out*.println("The first test result: "+ judgeWhetherCouldFormTheAllocatedSum.Judge());  
 System.*out*.println("The first solution result: "+ judgeWhetherCouldFormTheAllocatedSum.getSolution() );  
 S= 19;  
 JudgeWhetherCouldFormTheAllocatedSum judgeWhetherCouldFormTheAllocatedSum1 = new JudgeWhetherCouldFormTheAllocatedSum(A, S);  
 System.*out*.println("The second test result: " + judgeWhetherCouldFormTheAllocatedSum1.Judge());  
 System.*out*.println("The second solution result: "+ judgeWhetherCouldFormTheAllocatedSum1.getSolution());  
 S= 24;  
 JudgeWhetherCouldFormTheAllocatedSum judgeWhetherCouldFormTheAllocatedSum2 = new JudgeWhetherCouldFormTheAllocatedSum(A, S);  
 System.*out*.println("The third test result: " + judgeWhetherCouldFormTheAllocatedSum2.Judge());  
 System.*out*.println("The third solution result: "+judgeWhetherCouldFormTheAllocatedSum2.getSolution());  
 }  
}

15.2

题目：

有n堆棋子(n为正整数)，每堆有正整数个棋子。每次合并只能合并**相邻**的两个堆，每次合并的开销为**两个堆棋子数之和**。分别采用**DP和Greedy**的思想，设计算法，求解开销最小的合并方式。

设计思想：

1. DP思想

这道题和课堂上讲过的“矩阵乘法链”的DP算法如出一辙。

对于任意一个序列，其最小开销为所有两个子序列开销之和再加上将这两个子序列合并的开销中的最小值。

如果我们让**T(i,j)**表示从序列中的**第i个元素开始，到第j个元素结束的子序列的最小合并开销**,则可以得到状态转移方程:

这里两个子序列合并的开销，就是把整个序列中的每个堆的棋子合并到一起，因此合并开销就是从i到j的元素开销之和。

注意这里是要取所有开销中的**最小值(min)**。

因为在计算一个序列的最小的开销时需要使用其所有子序列的开销，因此我们需要使用bottom-up的思想。

显然，一个子序列的长度显然小于等于原序列的长度，因此我们在维护DP的二维表的时候，本质上迭代的结果是一个**带状矩阵**。

如果我们用数字来描述被迭代时for循环中的索引，那么整个过程应该如下图所示。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **i/j** | 1 | 2 | 3 | … | n |
| 1 | 0 | 1 | 2 | … | n |
| 2 | x | 0 | 1 | … | n-1 |
| 3 | x | x | 0 | … |  |
| … | x | x | … | 0 | 1 |
| n | x | x | … | … | 0 |

另外，我们需要在迭代时记录每个子序列的**“断点”**(breaking point)，例如，序列1~7是由子序列1~4和5~7合并而成，那么断点为4。这件事可以在迭代维护DP二维表时完成。

最后，当我们需要回溯一个解时，就要用到这个断点表。由于本题DP做法和矩阵乘法链的DP做法极为相似，我们采用矩阵乘法链的解的形式，即形如，

的解的形式。

括号代表提升这次合并操作的优先级。在这个例子中，先合并a1,a2，再合并a3,a4。将其各自结果合并。再与先做a6,a7合并，再与a5合并的结果合并，可得到最小开销。

为了获得这样的解，我们可以根据解表格递归回溯这个解。如果我们想回溯序列(i,j)的解，那么相当于回溯**子序列(i,断点)和子序列(断点索引+1,j)的解**。

1. Greedy思想

Greedy算法一般比较直观，在这里显然每一步最优是选择两个**相邻和最小**的堆进行合并。算法正确性在后面部分证明。

一般我们采用数组来存放每次合并后的新序列，这导致索引与最初的数组索引不同，因此我们很难用一种方式存放断点，然后在求得最优解后回溯得到解向量。因此我们需要在迭代新序列时同时更新每一步得到的解向量。

下面以测试时的一个用例为例来解释解向量的形成过程。这个例子中的数据为**{a1,a2,a3,a4,a5,a5,a7} = {2,3,2,5,7,1,6}**

首先，解向量表中的每一个元素都是一个字符串，不断拼接得到最终的解向量。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **2** | **3** | **2** | **5** | **7** | **1** | **6** |
|  | **(2+3)** | **2** | **5** | **7** | **1** | **6** |
|  |  | **((2+3)+2)** | **5** | **7** | **1** | **6** |
|  |  |  | **((2+3)+2)** | **5** | **7** | **(1+6)** |
|  |  |  |  | **…** | **…** | **…** |
|  |  |  |  |  | **…** | **…** |
|  |  |  |  |  |  | **…** |

在每次数组发生实际合并时，同时将解向量表中同样位置的字符串拼接合并，放到下一行中，不断迭代形成解向量。（整张表的最右下角的元素）

设计思想图解：

1. DP思想

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **i/j** | 1 | 2 | 3 | … | n |
| 1 | 0 | a1+a2 |  |  |  |
| 2 | x | 0 | a2+a3 |  |  |
| 3 | x | x | 0 | … |  |
| … | x | x | … | 0 |  |
| n | x | x | … | … | 0 |

显然，我们需要维护的DP二维表是一个**三角矩阵（可以通过映射函数减少内存空间申请）**，因为从i到j的开销本质上就是从j到i的开销。同时，一堆棋子是没有合并的概念的，因此对角线上的元素都为0。

1. Greedy思想

DP思想中画出设计图是因为需要一个图解。但Greedy思想实在太过于显然，因此在这里不再浪费篇幅画出图解。

伪代码：

1. DP思想

Calculate\_Minimal\_Cost() //

{

For every possible length as l of the subsequence

For each row as i in the DP table as T

j = i+l-1 // set the end index

min,mink // min is the minimal cost and mink is the breaking point

For each possible offset as k

cost =

if cost < min

min = cost // update minimal cost

mink = i+k // update the breaking point

T(i,j) = min // the final minimal cost for sequence(i,j)

S(i,j) = mink // **S is the solution breaking point table**

return T(1,n) // final answer for the whole sequence

}

Trace\_Back\_Solution(start,end) // get the solution of the **subsequence(start,end)**

{

if start = end

return ‘a’ + start // It’s ok to select one of {start,end} since they are equal

else

// **recursively** get the solution

return ‘(‘ +

Trace\_Back\_Solution(start,S(i,j))+

Trace\_Back\_Solution(S(i,j)+1,end)

+ ‘)’

}

1. Greedy思想

Calculate\_Minimal\_Cost()

{

totalCost = 0; // final minimal cost

row = 1; //solution table’s updating index

while(rest of the elements in weights[] are more than 1)

totalCost = totalCost + merge(row)

row = row + 1

return totalCost

}

merge(row) // we update the **solution** table and the **weights[]** array **concurrently**

{

initialize variable min,formerIndex // to **record the two merged elements**

For each possible position as i in weights[]

if(weights[i]+weights[i+1] < min)

min = weights[i]+weights[i+1]

formerIndex = i

//**merge the two selected elements**

weights[formerIndex] = weights[formerIndex]+ weights[formerIndex+1]

For each element as e whose index is behind latterIndex=formerIndex+1

// keep the **elements’ positions right**

weights[e.index-1] = weights[e.index]

// update the solution table

nextRow = row+1

// there’re a lot of boring shitty **hard codes** in this part

// but as a **coder** we have to **tolerate all of these**

For each string as s in solutionTable[row] whose index before formerIndex

solutionTable[nextRow][s.index+1] = s

//concat the solution string

solutionTable[nextRow][latterIndex] = ‘(‘+

solutionTable[row][formerIndex]+

solutionTable[row][latterIndex]+

‘)’

For each string as s in solutionTable[row] whose index behind latterIndex

solutionTable[nextRow][s.index] = s

// since we merge two elements ,there’re less elements than before

weights[].length-=1

return min

}

复杂度分析：

1. DP思想

时间复杂度:

**初始化DP表、解断点表：**两张表都是n\*n的表，θ(n²)

**迭代DP表、解断点表:** 最外层循环θ(n)次，次外层循环θ(n-l)次,最内层循环θ(l)次，计算子序列和θ(l)次。

如果仅仅从代数角度上考虑这个计算问题，那么步骤将是浩繁的。

但关键在于，这个算法做了一个实际的操作：迭代一个矩阵。

所以如果结合矩阵的wilkinson图（如上述的图解）来看，这个问题会简化很多。

**=**

**= = O**

**回溯解：**鉴于解断点表的断点个数可能是不确定的，所以一定是O的复杂度。

同时，断点个数最多为(n-1)个，（不断地向一个堆合并）则一个断点产生2个 分支，（前、后子序列的回溯）前者为θ(1)，后者为T 通过递归树，可以得 到，时间复杂度为θ(n)。

在**最坏情况下**时间复杂度为θ(n)，则在所有情况下为O(n)。

**总计：**O

空间复杂度：

维护两个表：θ(n²)

1. Greedy思想

时间复杂度：

**初始化解向量表：**表是n\*n的，因此为θ(n²)

**合并相邻和最小的两个元素：**

找到这两个元素需要遍历数组θ(r),r为当前数组剩余元素的个数。

合并需要θ(1)

这样的过程共计(n-1)次，因此总计:

= O(n2)。

**迭代更新解向量表：**这个过程相当于把上一行的临时解向量中的两项合并(θ(1)时间)后将整行移动到下一行。所以和合并并更新数组的操作所需时间的计算公式是完全一致的，结果也为O(n2)。

**总计：**θ(n2)。

空间复杂度：

**迭代更新了一个数组，并维护了一个解向量表：**θ(n2)

算法正确性分析：

1）DP思想

对于DP而言，这是显然的，因为**状态转移方程等价于整个题目的最优子结构的迭代求解方法**。则DP思想一般不需要证明正确性。于是重点来到Greedy思想的算法正确性证明上。

2）Greedy思想

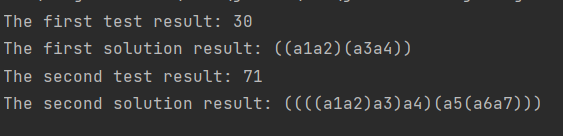
1.每次合并时，在总代价中产生的额外代价为当前选择的两个元素的代价和,weights[i]+weights[i+1]。因此每次选择相邻和最小的两个元素进行合并会在总代价中产生最小的额外开销。

2.对于任意的一个序列，其最小总代价为两个代价最小的子序列的代价和再加上整个序列中所有元素的开销的和，注意到这一事实对整个序列也成立。

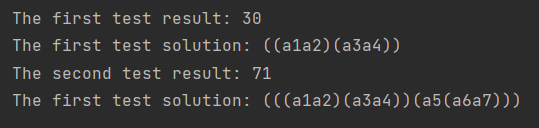
综合1、2，由于总代价为每次合并的代价的总和，且每次合并的代价又最小，则该算法会导致最小的总代价。

编码分析：

java语言编写，包含了测试结果。经检验两种算法结果均正确。



（result by greedy algorithm）



(result by DP algorithm)

这里最小开销形成方式的答案不一致是很正常的，因为**解本来就不止一个**。

DP code:

package com.DP.moveChess;  
  
*/\*\*  
 \* 这题完完全全就是矩阵乘法链的改版  
 \*/*public class moveChess {  
 private final int[] weights;//零索引弃用  
 private int[][] DPTable;  
 private int[][] solutionTable;//保存解的二维表  
 private final int n;//方阵行数  
  
 public moveChess(int[] weights){  
 this.weights=weights;  
 this.n = weights.length-1;  
 initDPTable();  
 initSolutionTable();  
 }  
  
 private void initDPTable(){  
 //其实DP表是个三角矩阵，可以通过一些映射函数缩小申请的内存空间，但我懒得做了，因为渐进复杂度都是n²  
 this.DPTable = new int[weights.length][weights.length];  
 //一堆棋子没有合并的必要，对角线元素均为0  
 for(int i=1;i<=n;i++){  
 DPTable[i][i]=0;  
 }  
 }  
  
 private void initSolutionTable(){  
 this.solutionTable = new int[weights.length][weights.length];  
 }  
  
 //其实一堆int相加不一定也是int，但这里主要考虑的内容是算法，溢出无所谓，就当是脏数据  
  
 */\*\*  
 \* 状态转移方程:  
 \* T(i,j) = min(0≤k≤(j-i)/2) { T(i,i+k) + T(i+k+1,j) + ∑(p start from i,j) weight[p] }  
 \** ***@return*** *最小耗费  
 \*/* public int calculateMinimalCost(){  
 //先两个两个，再三个三个，最后n个n个  
 //eg. 1-2,2-3,...,n-1-n; 1-3,2-4,......  
 //l为本次迭代的序列长度  
 for(int l=2;l<=n;l++){  
  
 //i为本次迭代的行号  
 for(int i=1;i<=n-l+1;i++){  
 int j = i+l-1;//j为对应的列号,(i,j)意即从第i堆棋子到第j堆棋子  
 int subSum = sumOfItoJ(i, j);  
 //DP方程计算计算最小消费  
 int min = DPTable[i][i]+DPTable[i+1][j]+ subSum;  
 int minK = i;//记录断点位置  
 for(int k=0;k<=l-2;k++){  
 int cur = DPTable[i][i + k] + DPTable[i + k + 1][j] + subSum;  
 if(cur < min){  
 min = cur;  
 minK = i+k;  
 }  
 }  
 //迭代二维表  
 DPTable[i][j]=min;  
 //更新解表断点  
 solutionTable[i][j] = minK;  
 }  
  
 }  
 return DPTable[1][n];//返回T(1,n)的值  
 }  
  
 private int sumOfItoJ(int i,int j){  
 int sum=0;  
 for(int index=i;index<=j;index++){  
 sum+=weights[index];  
 }  
 return sum;  
 }  
  
 public String getSolution(){  
 return traceBackSolution(1, n);  
 }  
  
 private String traceBackSolution(int start,int end){  
 if(start==end){  
 return "a"+start;  
 }  
 else{  
 return "("+  
 traceBackSolution(start,solutionTable[start][end])+  
 traceBackSolution(solutionTable[start][end]+1,end)+  
 ")";  
 }  
 }  
  
 public static void main(String[] args) {  
 //test for the segment of DP code  
 //answer supposed to be 30,71(by the greedy algorithm)  
 int[] test = {-1,5,2,7,1};  
 moveChess moveChess = new moveChess(test);  
 int i = moveChess.calculateMinimalCost();  
 System.*out*.println("The first test result: "+i);  
 System.*out*.println("The first test solution: "+moveChess.getSolution());  
 int[] test2 = {-1,2,3,2,5,7,1,6};  
 moveChess moveChess1 = new moveChess(test2);  
 int i1 = moveChess1.calculateMinimalCost();  
 System.*out*.println("The second test result: "+i1);  
 System.*out*.println("The first test solution: "+moveChess1.getSolution());  
 }  
}

Greedy code:

package com.Greedy.moveChess;  
  
import java.util.Arrays;  
  
*/\*\*  
 \* 贪心相对于DP在这道题上的优势是更加直观  
 \* 但是缺点是需要证明正确性  
 \* 简单来说，策略就是每次挑选两个相邻的且和最小的元素合并，累计cost  
 \*/*public class moveChess {  
 private final int[] weights;//零索引弃用  
 private int n;//dynamic length  
 private String[][] solutionTable;//和合并过程一起迭代  
  
 public moveChess(int[] weights) {  
 this.weights = weights;  
 this.n = weights.length-1;  
 initSolutionTable();  
 }  
  
 private void initSolutionTable(){  
 this.solutionTable = new String[weights.length][weights.length];  
 //为了防空指针我还是全部初始化成空串了，其实不需要的  
 for(int i=1;i<=n;i++){  
 for(int j=1;j<=n;j++){  
 solutionTable[i][j]="";  
 }  
 }  
 for(int j=1;j<=n;j++){  
 solutionTable[1][j] = "a"+j;  
 }  
 }  
  
 public int calculateMinimalCost(){  
 int totalCost=0;  
 int row =1;//记录解表行数  
 while(n>1){ //合并到最后一个值  
 totalCost+=mergeAdjacentMinimalSum(row);  
 row+=1;  
 }  
 return totalCost;  
 }  
  
 */\*\*  
 \* 合并相邻和最小的两个元素  
 \** ***@return*** *本次合并的开销  
 \*/* private int mergeAdjacentMinimalSum(int row){  
 if(n<=1){  
 return weights[1];  
 }  
 int min = weights[1]+weights[2];  
 int formerIndex = 1;  
 //找到当前合并的两个元素的前面的索引，并更新最小值  
 for(int i=1;i<=n-1;i++){  
 int sum = weights[i] + weights[i + 1];  
 if(sum <min){  
 formerIndex=i;  
 min = sum;  
 }  
 }  
 weights[formerIndex]=weights[formerIndex]+weights[formerIndex+1];//merge  
 for(int i=formerIndex+2;i<=n;i++){  
 weights[i-1]=weights[i];//update  
 }  
 //下一行索引  
 int nextRow = row+1;  
 //把formerIndex之前的全部顺延挪到下一行  
 for(int i=row;i<formerIndex+row-1;i++){  
 solutionTable[nextRow][i+1] = solutionTable[row][i];  
 }  
 //formerIndex和formerIndex+1合并到下一行的formerIndex+1处  
 solutionTable[nextRow][formerIndex+1+row-1] = "("+solutionTable[row][formerIndex+row-1]+solutionTable[row][formerIndex+1+row-1]+")";  
 //formerIndex+1之后的直接挪到下一行  
 for(int i = formerIndex+1+1+row-1;i<= weights.length-1;i++){  
 solutionTable[nextRow][i] = solutionTable[row][i];  
 }  
 n-=1;  
 return weights[formerIndex];  
 }  
  
 public String getSolution(){  
 return solutionTable[weights.length-1][weights.length-1];  
 }  
  
 public static void main(String[] args) {  
 //test for the segment of DP code  
 //answer supposed to be 30,71(by DP)  
 int[] test = {-1,5,2,7,1};  
 com.Greedy.moveChess.moveChess moveChess = new com.Greedy.moveChess.moveChess(test);  
 int i = moveChess.calculateMinimalCost();  
 System.*out*.println("The first test result: "+i);  
 System.*out*.println("The first solution result: "+moveChess.getSolution());  
 int[] test2 = {-1,2,3,2,5,7,1,6};  
 com.Greedy.moveChess.moveChess moveChess1 = new com.Greedy.moveChess.moveChess(test2);  
 int i1 = moveChess1.calculateMinimalCost();  
 System.*out*.println("The second test result: "+i1);  
 System.*out*.println("The second solution result: "+moveChess1.getSolution());  
 }  
}