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)。

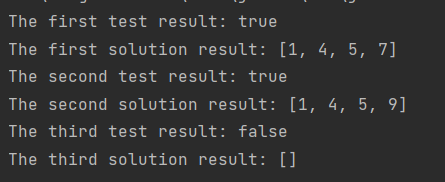
整个问题总计：

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());  
 }  
}