# 状态DP-动态规划

## 一：枚举子集的动态规划+存储两个值的动态规划

### 题目描述：



### 题目分析：

* //根据条件1<=n<=14;  
    
   //该题用贪心解决不了  
    
   //状态DP安排上  
    
   //法1：枚举子集的动态规划  
    
   //动态规划的转移方程：f[mask]=min(f[mask/subset]+1){枚举mask的所有自己subset}
* **使用位运算来简化数据结构；用位运算中的1表示任务已完成**
* **通过位运算枚举子集的技巧**

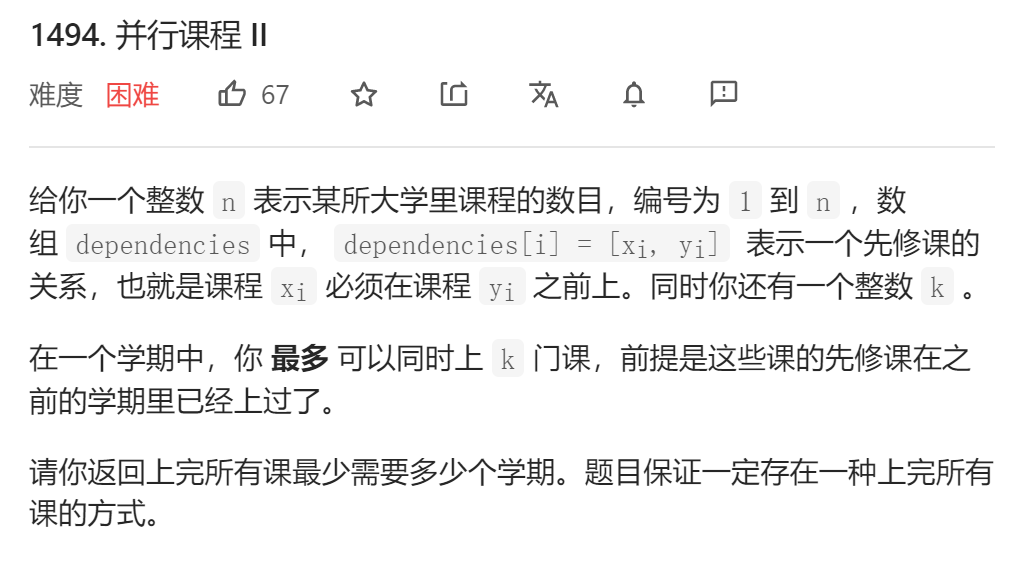
subset = mask  
while subset != 0 do  
 // subset 是 mask 的一个子集，可以用其进行状态转移  
 ...  
 // 使用按位与运算在 O(1) 的时间快速得到下一个（即更小的）mask 的子集  
 subset = (subset - 1) & mask  
end while  
  
作者：LeetCode-Solution  
链接：https://leetcode-cn.com/problems/minimum-number-of-work-sessions-to-finish-the-tasks/solution/wan-cheng-ren-wu-de-zui-shao-gong-zuo-sh-tl0p/  
来源：力扣（LeetCode）  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

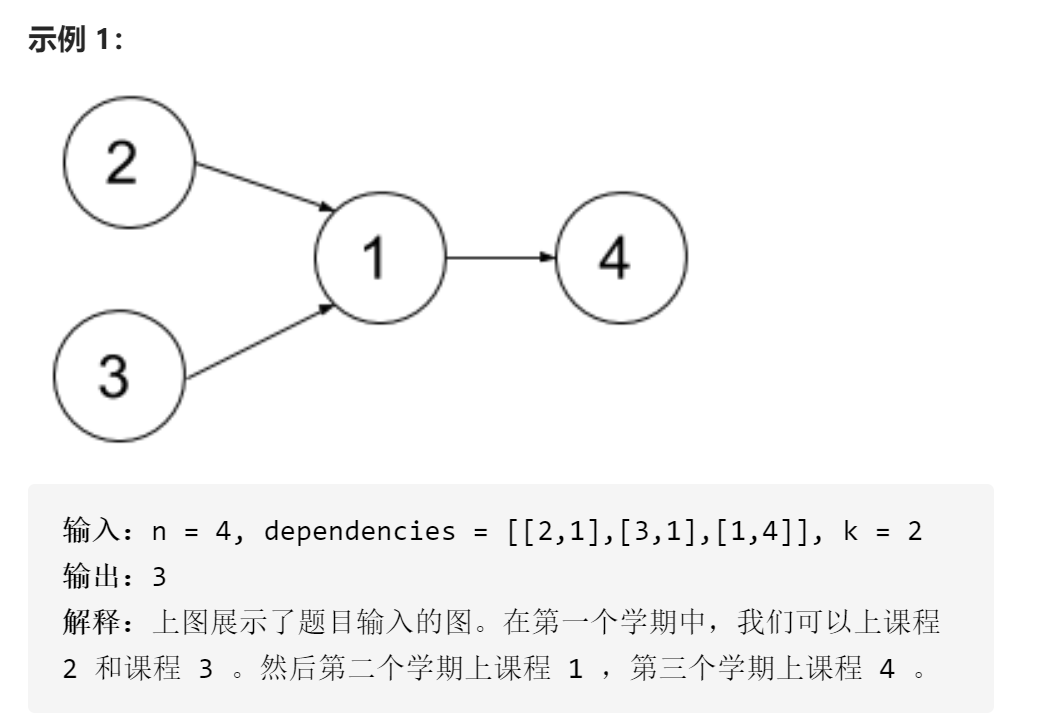
### 代码描述：

class Solution {  
public:  
 int minSessions(vector<int>& tasks, int sessionTime) {  
 //根据条件1<=n<=14;  
 //该题用贪心解决不了  
 //状态DP安排上  
 //法1：枚举子集的动态规划  
 //动态规划的转移方程：f[mask]=min(f[mask/subset]+1){枚举mask的所有自己subset}  
 int n=tasks.size();  
 vector<bool>valid(1<<n,false);  
 //检查各种情况下是否可以独立的处于一个时间段  
 for(int mask=0;mask<(1<<n);++mask){  
 int tempsum=0;  
 for(int i=0;i<n;i++){  
 if(mask&(1<<i))tempsum+=tasks[i];  
 }  
 if(tempsum<=sessionTime)valid[mask]=true;  
 }  
  
 //开始动态的一个转移  
 //设置一个动态的dp  
 vector<int>f(1<<n,INT\_MAX/2);  
 f[0]=0;  
 for(int mask=0;mask<(1<<n);mask++){  
 //枚举mask的自己subset  
 //枚举技巧  
 for(int subset=mask;subset;subset=(subset-1)&mask){  
 if(valid[subset])  
 f[mask]=min(f[mask],f[mask^subset]+1);  
 }  
 }  
 return f[(1<<n)-1];  
   
 }  
};

## 二：枚举子集的动态规划--根据不同情况进行子集的枚举

### 题目描述：





### 题目分析：

1. **这道题和上一道题差不多。差别在于“如何枚举子集上”**
2. **上一道题元素之间没有约束关系，故可以从头到尾进行一个枚举**
3. **这道题元素之间存在约束关系，不是每个随机组成的元素都能构成一个子集。**
4. **该题元素构成子集的条件：**

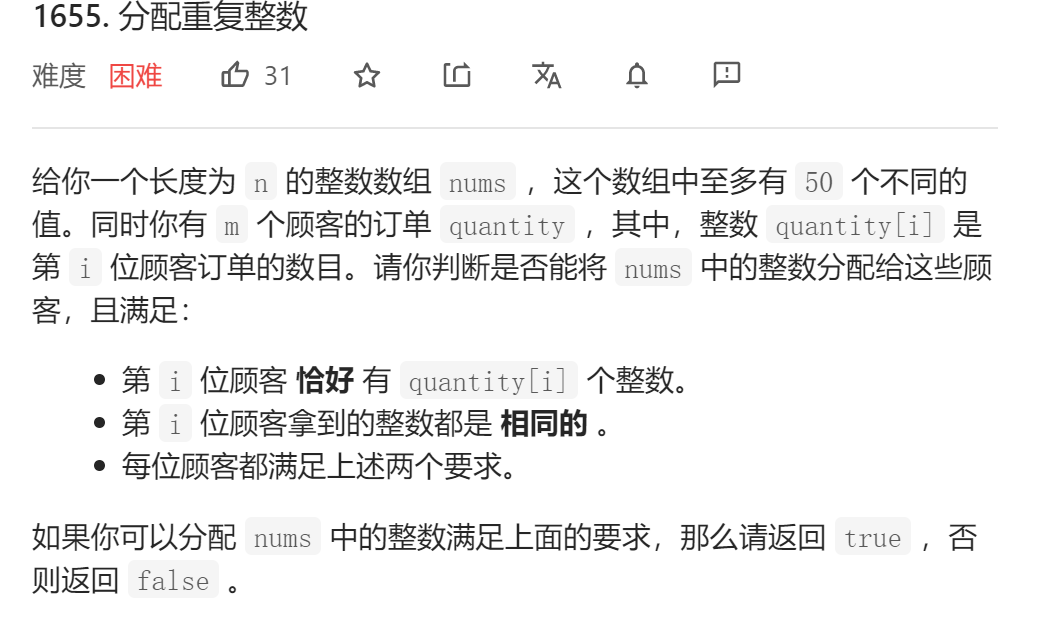
* 子集中元素的个数小于k
* 子集中各个元素存在的先决条件所构成的子集的并集与该子集没有交集；（即子集中个元素之间没有约束关系）

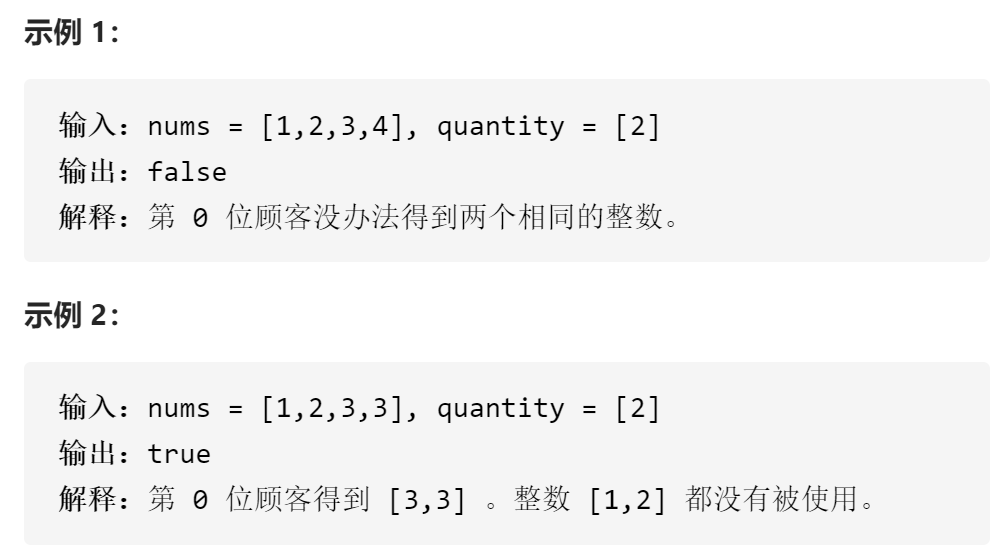
### 代码分析：

class Solution {  
public:  
 int minNumberOfSemesters(int n, vector<vector<int>>& relations, int k) {  
 //试试枚举子集的动态规划  
 vector<bool>valid(1<<n,false);  
 vector<int>set\_fomer(1<<n,0);  
 //枚举子集的可行性  
 //先枚举每个元素所必须的先决条件  
 vector<int>exist(n,0);  
 for(auto temp:relations){  
 exist[temp[1]-1]|=(1<<(temp[0]-1));  
 }  
 //在枚举元素的可行性  
 for(int mask=0;mask<(1<<n);++mask){  
 //\_\_builtin\_popcount用于计算二进制数中1的个数  
 if(\_\_builtin\_popcount(mask)<=k){  
 //当K个元素之间没有先后制约关系时，可以作为一个有效的子集  
 for(int i=0;i<n;i++){  
 if(mask&(1<<i)){  
 set\_fomer[mask]|=exist[i];  
 }  
 }  
  
 if((set\_fomer[mask]&mask)==0)valid[mask]=true;  
 }  
 }  
  
 //枚举子集的动态规划寻找答案  
 vector<int>f(1<<n,INT\_MAX/2);  
 f[0]=0;  
 for(int mask=0;mask<(1<<n);++mask){  
 //划分出去的子集的先决条件要在原子集中存在  
 for(int subset=mask;subset;subset=(subset-1)&mask){  
 if(valid[subset]&&(set\_fomer[subset]&mask)==set\_fomer[subset]){  
 f[mask]=min(f[mask],f[mask^subset]+1);  
 }  
 }  
 }  
  
 return f[(1<<n)-1];  
  
 }  
};

## 三：枚举子集的动态规划

### 题目描述：





### 题目分析：

1. **如何划分子集：首先确定动态转化方法：dp[i] [j]表示用前i个容器来装满足前j个顾客的需求**
2. **枚举子集：第i个容器可以满足的j的某个子集s;而剩下的容器可以满足j-s,即剩下的子集，进行动态的转移**

### 代码分析：

class Solution {  
public:  
 bool canDistribute(vector<int>& nums, vector<int>& quantity) {  
 //状态压缩DP  
 //nums的值不重要，nums中相同值的个数比较重要  
 unordered\_map<int,int>mymap;  
 for(auto temp:nums)++mymap[temp];  
 vector<int>vec;  
 for(auto it=mymap.begin();it!=mymap.end();it++){  
 vec.push\_back(it->second);  
 }  
 //问题转换为m个顾客，n种不同数量的订单的最佳匹配问题  
 //动态规划：dp[i][j]表示前i种订单能否满足前j个顾客  
 //状态转移：j的某个子集s由i完成，而剩下的由0..i-1完成  
 //求子集的和  
 int m=quantity.size();  
 vector<int>sum(1<<m,0);  
 for(int i=1;i<(1<<m);i++){  
 for(int j=0;j<m;j++){  
 if(i&(1<<j)){  
 int left=i-(1<<j);  
 sum[i]=sum[left]+quantity[j];  
 break;  
 }  
 }  
 }  
 //动态规划  
 int n=vec.size();  
 vector<vector<bool>>dp(n,vector<bool>(1<<m,false));  
 //初始化  
 for(int i=0;i<n;i++)dp[i][0]=true;  
 for(int i=0;i<n;i++){  
 for(int j=0;j<(1<<m);j++){  
 //剪枝  
 if(i>0&&dp[i-1][j]){  
 dp[i][j]=true;  
 continue;  
 }  
  
 //枚举子集  
 for(int subset=j;subset;subset=(subset-1)&j){  
 int left=j-subset;//left是剩下的集合  
 bool flag1=(sum[subset]<=vec[i]);  
 //i等于0的时候，只有子集为0才成  
 bool flag2=((i==0)?(left==0):dp[i-1][left]);  
 if(flag1&&flag2){  
 dp[i][j]=true;  
 break;  
 }  
 }  
 }  
 }  
 return dp[n-1][(1<<m)-1];  
 }  
};