**[0/1背包问题](http://www.cnblogs.com/DiaoCow/archive/2010/04/17/1714312.html)**

0/1背包问题是背包问题中最基本的一种，其状态转移方程：m[i][j] = Max(m[i+1][j] , m[i+1][j - w[i]] + v[i])

对比了自己的代码和王晓东书上的代码，感觉在两方面值得自己学习：

1. jMax = Min(w[i] - 1 , c); 这样写法使得在（0 ~ jMax)可以直接赋值，无需判断，因此减少了判断次数；

2. 对于求m[1][c],并不是放在循环体里，而是单独拎出来求，这样就避免了求解很多“无意义”情况；

#include <stdio.h>  
  
#define Min(a,b) ((a) < (b) ? (a) : (b))  
#define Max(a,b) ((a) > (b) ? (a) : (b))  
#define N 20  
  
  
int m[N][N] , v[N] , w[N] , n , c;  
  
int Knapsack()  
{  
 int i , j , jMax;  
  
 jMax = Min(w[n] - 1 , c);  
 for(j = 0 ; j <= jMax ; j++) m[n][j] = 0;  
 for(j = w[n] ; j <= c ; j++) m[n][j] = v[n];  
  
 for(i = n - 1 ; i > 1 ; i--)  
 {  
 jMax = Min(w[i] - 1 , c);  
 for(j = 0 ; j <= jMax ; j++) m[i][j] = m[i+1][j];  
 for(j = w[i] ; j <= c ; j++) m[i][j] = Max(m[i+1][j] , m[i+1][j - w[i]] + v[i]);  
 }  
 m[1][c] = m[2][c];  
 m[1][c] = Max(m[2][c] , m[2][c - w[1]] + v[1]);  
  
 return m[1][c];  
}  
  
void TraceBack(int \*x)  
{  
 int i , j;  
  
 for(i = 1 , j = c ; i < n ; i++)  
 {  
 if(m[i][j] == m[i+1][j])   
 {  
 x[i] = 0;  
 }  
 else  
 {  
 x[i] = 1;  
 j -= w[i];  
 }  
 }  
 x[n] = m[n][j] ? 1 : 0;  
}  
  
int main(void)  
{  
 int x[N] , i;  
  
 scanf("%d%d", &c , &n); //背包容量，背包个数  
 for(i = 1 ; i <= n ; i++) scanf("%d", w + i); //物品重量  
 for(i = 1 ; i <= n ; i++) scanf("%d", v + i); //物品价值  
  
 printf("Max Value %d\n", Knapsack());  
 TraceBack(x);  
 printf("选择的物品：");  
 for(i = 1 ; i <= n ; i++)  
 {  
 if(x[i]) printf("重量(%d)\_价值(%d) ", w[i] ,v[i]);  
 }  
  
 return 0;   
}

上面程序的时间复杂度和空间复杂度都是O(nc)。时间复杂度我们已经无法再优化了，但是空间复杂度我们还是可以降低到O(c)。

理由就是:每次求m[i][j]只需要利用前一行数据m[i+1][0~c],因此只要维护一个一维数组m[0~c]即可（也可以维护一个二维数组m[2][c],然后循环滚动赋值）

int Knapsack()  
{  
 int i , j , jMax;   
   
 jMax = Min(w[n] - 1 , c);   
 for(j = 0 ; j <= jMax ; j++) m[j] = 0;   
 for(j = w[n] ; j <= c ; j++) m[j] = v[n];  
   
 for(i = n - 1 ; i > 1 ; i--)   
 {   
 for(j = c ; j >= w[i] ; j--)   
 {  
 m[j] = Max(m[j] , m[j - w[i]] + v[i]);   
 }  
 }   
 m[c] = Max(m[c] , m[c - w[1]] + v[1]);  
 return m[c];  
}

1.01-package （最直接应用）

题目描述：给定一个背包的容量k，给定n个物品的体积和价值，物品不可分割，将n个物品中选若干个物品放入背包，求背包内物品的最大价值总和，在价值总和最大的前提下求背包内的最小物品个数c。

分析：求最大价值的方法我们在前面已经分析过了，现在只要知道如何求最小物品个数。如果放入当前物品i 使总价值增加，那么当前物品数为：cnt[j] = cnt[j-w[i]] + 1(j为背包容量) ；如果放入当前物品不会对总价值造成影响，那么我们就要找“最小物品”即，cnt[j] = Min(cnt[j] , cnt[j - w[i]] + 1) ；

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

#include <stdio.h>  
#include <string.h>  
  
#define Min(a,b) ((a) < (b) ? (a) : (b))  
#define N 2000  
  
int m[N] , cnt[N] , w[N] , v[N] , n , c;  
  
void Knapsack()  
{  
 int i , j , jMax;   
   
 jMax = Min(w[n] - 1 , c);   
 for(j = 0 ; j <= jMax ; j++) {m[j] = 0; cnt[j] = 0;}   
 for(j = w[n] ; j <= c ; j++) {m[j] = v[n]; cnt[j] = 1;}  
   
 for(i = n - 1 ; i > 1 ; i--)   
 {   
 for(j = c ; j >= w[i] ; j--)   
 {  
 if(m[j] < m[j - w[i]] + v[i])  
 {  
 m[j] = m[j - w[i]] + v[i];  
 cnt[j] = cnt[j-w[i]] + 1;  
 }  
 else if(m[j] == m[j - w[i]] + v[i])  
 {  
 cnt[j] = Min(cnt[j] , cnt[j - w[i]] + 1);  
 }   
 }  
 }   
   
 if(m[c] < m[c - w[1]] + v[1])  
 {  
 m[c] = m[c - w[1]] + v[1];  
 cnt[c] = cnt[c-w[1]] + 1;  
 }  
 else if(m[c] == m[c - w[1]] + v[1])  
 cnt[c] = Min(cnt[c] , cnt[c - w[1]] + 1);  
}  
  
  
int main(void)  
{   
 int z , i;  
   
 scanf("%d", &z);  
 while(z-- > 0)  
 {  
 scanf("%d%d", &n,&c);  
 for(i = 1 ; i <= n ; i++)  
 scanf("%d%d", v + i , w + i);  
  
 Knapsack();  
 printf("%d %d\n", m[c] ,cnt[c]);   
 }  
 return 0;

2.Incredible Cows  
这道题实际可以抽象成：把一组数分成两个集合，使得这两个集合和的绝对值差最小。  
分析：分成两个集合，那么一个数要么放在集合1，要么放在集合2，也就是，  
x[i] = 0 : 第i个数放入第1个集合  
x[i] = 1 ：第i个数放入第2个集合  
这显然是个0/1背包问题，这里的物体重量w[i]就是i本身，且w[i]==v[i] ,(这题由于C较大用回溯法解决01背包而不是DP)

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifhttp://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif代码

#include <stdio.h>  
  
#define Max(a,b) ((a) > (b) ? (a) : (b))  
#define N 35  
  
int m[N] , w[N] , n , c , cw , best;  
  
int Bound(int i)  
{  
 int cleft = c - cw;  
 int b = cw;  
  
 while(i <= n && w[i] <= cleft)  
 {  
 cleft -= w[i];  
 b += w[i];  
 i++;  
 }  
 if(i <= n) b += cleft;  
  
 return b;  
}  
  
void Knapsack(int i)  
{  
 if(i > n)  
 {  
 best = Max(cw , best);  
 return;  
 }  
 if(cw + w[i] <= c) //left  
 {  
 cw += w[i];  
 Knapsack(i + 1);  
 cw -= w[i];  
 }  
 if(Bound(i+1) > best) //right（剪枝）  
 {  
 Knapsack(i + 1);  
 }  
}  
  
void QuickSort(int \*arr , int left , int right)  
{  
 int i , j , x , nTemp;  
  
 if(left >= right) //边界条件检查  
 return;  
 else  
 {   
 //Partition  
 i = left; j = right + 1; x = arr[i];  
 while(1)  
 {  
 do i++; while(i < j && arr[i] > x);  
 do j--; while(arr[j] < x);  
 if(i > j) break;  
 //swap(i,j)  
 nTemp = arr[i]; arr[i] = arr[j]; arr[j] = nTemp;  
 }  
 //swap(left,j)  
 nTemp = arr[left]; arr[left] = arr[j]; arr[j] = nTemp;  
  
 QuickSort(arr,left,j-1);  
 QuickSort(arr,j+1,right);  
 }  
}  
  
  
int main(void)  
{   
 int z , i , k , j;  
   
 scanf("%d", &z);  
 while(z-- > 0)  
 {  
 k = cw = best = 0;  
 scanf("%d", &n);  
 for(i = 1 ; i <= n ; i++)  
 {  
 scanf("%d" , w + i);  
 k += w[i];  
 }  
  
 QuickSort(w,1,n);  
  
 c = k / 2 + (k & 1); //背包容量  
 Knapsack(1);  
 j = k - best;  
 printf("%d\n", j > best ? j - best : best - j);   
 }  
 return 0;   
}

总结：以后凡是遇到这种子集选取的问题，都可以抽象成01背包问题来解决（其中v[i]往往等于w[i])。