01背包问题

有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 w[i] ，价值是 v[i] ，求将哪些物品装入背包可使价值总和最大。

特点是：每种物品仅有一件，可以选择放或不放。

有 N 件物品和一个容量为 V 的背包。 第 i 件物品的费用是 w[i] ， 价值是 v[i] 。 求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量， 且价值总和最大。

```
int w[5] = {2,2,6,5,4};//物品的体积
int v[5] = {6,3,5,4,6};//物品的价值
int C=10;//背包最大容量
```

思路：一个完整的 n 个物品的问题比较难，无法解决。但是如果只有一个物品 i=0 的时候，答案确实很明显的。

1. 将 i=0 物品放入背包。
2. 不将 i=0 物品放入背包。 注意：如果是 1 的情况，这时背包剩余的空间为 C-w[i] ，减小了。

这样，归纳基础就讨论好了。解决这个问题的整个过程就是对所有的物品进行逐个的实验（放入或不放入），每一次的决策都有两种可能。那么假设前 i-1 个物品都已经决策好了，考虑第 i 个物品。

定义状态： f(i,j) （ f[i][j] ）表示前 i 件物品恰放入一个容量为 j 的背包可以获得的最大价值。则其状态转移方程便是：

$$f(i,j) = max\{f(i-1,j), f(i-1,j-w[i]) + v[i]\}$$

若只考虑第 i 件物品的策略(放或不放,前 i-1 已经决策完成)，那么就可以转化为一个只牵扯前 i-1 件物品的问题。如果不放第 i 件物品，那么问题就转化为**前** i-1 **件物品放入容量为** j **的背包中**，价值为 f[i-1][j] (价值没变)；如果放第 i 件物品，那么问题就转化为**前** i-1 **件物品放入剩下的容量为** j-c[i] **的背包中**，此时能获得的最大价值就是 f[i-1][j-w[i]] 再加上通过放入第 i 件物品获得的价值 v[i] 。

1. 前 i 个物品中挑选放入承重为 0 的背包中和没有物品放入承重为 j 的背包中是相等为 0 。 f(i,0)=f(0,j)=0
2. 如果第 i 个物品的体积量大于背包的容量，则装人前 i 个物品得到的最大价值和装入前 i-1 个物品得到的最大价是相同的，即物品 i 不能装入背包。 f(i,j)=f(i-1,j); //j<w[i]
3. 如果第 i 个物品的体积小于背包的容量，则会有以下两种情况：
   A. 如果把第 i 个物品装入背包，则背包物品的价值等于第 i-1 个物品装入容量位 j-w[i] 的背包中的价值加上第 i 个物品的价值 v[i]；
   B. 如果第 i 个物品没有装入背包，则背包中物品价值就等于把前 i-1 个物品装入容量为 j 的背包中所取得的价值。显然，取二者中价值最大的作为把前 i 个物品装入容量为 j 的背包中的最优解。

有了前面的讨论，完全可以使用函数的递归调用（ dfs ）找到问题的解。只是会有很多的重复计算，这时候就需要使用记忆化搜索优化。

注意函数有两个参数 i,j ，这不是天然的就对应与二维数组的下表?

In [1]:
```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    int max(int a, int b){
        return a>=b?a:b;
    }
}
#endif
```

Out[1]:

/

In [2]:
```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    int KnapSack01v1(int n, int w[], int v[], int x[], int C){
        int V[200][200];//前i个物品装入容量为j的背包中获得的最大价值
        int i, j;
        for (i = 0; i <= n; i++)  //第一种情况  这个其实可以省略
            V[i][0] = 0;
        for (j = 0; j <= C; j++)  //第一种情况  这个其实可以省略
            V[0][j] = 0;
        //上面两个for可以省略是因为整个计算并不需要从两个边递推。
        for (i = 0; i < n; i++){   //物品，一个一个的选
            for (j = 0; j < C+1; j++){   //容积，一个一个的推
                if (j<w[i]){       //第二种情况，新的i放不进去，所以保持原来的值
                    if(i>0)
                        V[i][j] = V[i-1][j];   //如果不是第一行，新的物品无法放入，那就将上一行的数字落下来。
                    else  //如果是最上面一行，i-1就会为-1，则是不对的，没有上一行。有些编辑器也可以用-1下标，但有隐患。
                        V[i][j] = 0;
                    //printf("-------->%d  %d  %d\n",i-1,j,V[i-1][j]);
                }else    //当可以放
                    V[i][j] = max(V[i-1][j], V[i-1][j-w[i]] + v[i]);     //第三种情况
            }
        }
        j = C;
        for (i = n-1; i >= 0; i--){
            if (V[i][j]>V[i-1][j]){
                x[i] = 1;
                j = j-w[i];
            }else
                x[i] = 0;
        }
        printf("选中的物品是:\n");
        for (i = 0; i<n; i++)
            printf("%d ", x[i]);
        printf("\n0\t1\t2\t3\t4\t5\t6\t7\t8\t9\t10       <------------所选物品的体积\n");
        printf("=================================================================================\n");
        for (int i = 0; i < n; i++){
            for (int j = 0; j < C+1; j++){
                printf("%d\t", V[i][j]);
                if (j == C){
                    printf(" <------------前%d个物品的价值，纵坐标为前%d个物品中选中物品的体积和\n",i+1,i+1);
                }
            }
        }
        return V[n-1][C];
    }
}
#endif
```

Out[2]:

In [3]:
```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    void get01v1(){
        int s;//获得的最大价值
        int w[5] = {2,2,6,5,4};//物品的重量
        int v[5] = {6,3,5,4,6};//物品的价值
        int x[5];//物品的选取状态
        int n = 5;
        int C=10;//背包最大容量

        s = KnapSack01v1(n, w, v, x, C);

        printf("最大物品价值为:\n");
        printf("%d\n", s);
        return ;
    }
}
#endif
```

Out[3]:

In [4]:
```
get01v1();
```

```
选中的物品是:
1 1 0 0 1
0       1       2       3       4       5       6       7       8       9       10      <------------所选物品的体积
=================================================================================
0       0       6       6       6       6       6       6       6       6       6       <------------前1个物品的价值，纵坐标为前1个物品中选中物品的体积和
0       0       6       6       9       9       9       9       9       9       9       <------------前2个物品的价值，纵坐标为前2个物品中选中物品的体积和
0       0       6       6       9       9       9       9       11      11      14      <------------前3个物品的价值，纵坐标为前3个物品中选中物品的体积和
0       0       6       6       9       9       9       10      11      13      14      <------------前4个物品的价值，纵坐标为前4个物品中选中物品的体积和
0       0       6       6       9       9       12      12      15      15      15      <------------前5个物品的价值，纵坐标为前5个物品中选中物品的体积和
最大物品价值为:
15
```

Out[4]: (void) nullptr

仔细观察发现，前两个 for 可以省略，重点是第三个双层 for 循环。还有一个特点就是计算每一行数据都只需要前面的一行数据，所以其实只需要两行数组就可以完成所有的计算，这个叫滚动数组。同学们自己研究一下吧!

dfs 的递归调用解决背包问题，采用的是从右下角的 15 向上搜索的过程。如果换一个方向（从上向下），则可以递推了，可以滚动数组了。有时候换一个方向，世界变得完全不一样。

优化空间复杂度（滚动数组）

In [5]:
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    int KnapSack01v2(int n, int w[], int v[], int x[], int C){
        int V[2][200];//前i个物品装入容量为j的背包中获得的最大价值
        int i, j;
        memset(V,0,sizeof(V));   //初始化数组为0
        for (i = 1; i <= n; i++){
            for (j = 0; j < C+1; j++){
                if (j<w[i-1]){           //装不下，没有等好
                    V[i%2][j] = V[(i-1)%2][j];
                }else    //当可以放
                    V[i%2][j] = max(V[(i-1)%2][j], V[(i-1)%2][j-w[i-1]] + v[i-1]);     //第三种情况
            }
            printf("------> %d %d:",i,i%2);
            for (int k = 0; k < C+1; k++){
                printf("%d\t",V[i%2][k]);
            }
            printf("\n");
        }

        return V[n%2][C];
    }
}
#endif
```

Out[5]:

In [6]:
```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    void get01v2(){
        int s;//获得的最大价值
        int w[5] = {2,2,6,5,4};//物品的重量
        int v[5] = {6,3,5,4,6};//物品的价值
        int x[5];//物品的选取状态
        int n = 5;
        int C=10;//背包最大容量

        s = KnapSack01v2(n, w, v, x, C);
        printf("%d\n", s);
        return ;
    }
}
#endif
```

Out[6]:

/

In [7]:
```
get01v2();
```

```
------> 1 1:0   0       6       6       6       6       6       6       6       6       6
------> 2 0:0   0       6       6       9       9       9       9       9       9       9
------> 3 1:0   0       6       6       9       9       9       9       11      11      14
------> 4 0:0   0       6       6       9       9       9       10      11      13      14
------> 5 1:0   0       6       6       9       9       12      12      15      15      15
15
```

Out[7]: (void) nullptr

同学们要观察取余的操作，也可以试试看有没有更好的办法。

优化空间复杂度（滚动都不滚了）

如果只用一个数组 f[0...V]，能不能保证第i次循环结束后 f[j] 中表示的就是我们定义的状态 f[i][j] 呢？

f[i][j] 是由 f[i-1][j] 和 f[i-1][j-w[i]] 两个子问题递推而来，能否保证在推 f[i][j] 时（也即在第i次主循环中推 f[j] 时）能够得到 f[i-1][j] 和 f[i-1][j-w[i]] 的值呢？事实上，这要求在每次主循环中我们以 j=V...0 的顺序推 f[j]，这样才能保证推 f[j] 时 f[j-w[i]] 保存的是状态 f[i-1][j-w[i]] 的值。至于为什么下面有详细解释。代码如下：

In [8]:
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    int KnapSack01v3(int n, int w[], int v[], int x[], int C){
        int V[200];//前i个物品装入容量为j的背包中获得的最大价值
        int i, j;
        memset(V,0,sizeof(V));   //初始化数组为 0
        for (int i = 0; i < n; i++){
            for (int j = C; j >= 0; j--){
                if(j>=w[i]){   //能装下，有等号
                    V[j] = max(V[j], V[j - w[i]] + v[i]);
                }else{
                    V[j]=V[j];
                }

            }
            for (int k = 0; k < C+1; k++){
                printf("%d\t",V[k]);
            }
            printf("\n");
        }
        return V[C];
    }
}
#endif
```

Out[8]:

In [9]:

```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    void get01v3(){
        int s;//获得的最大价值
        int w[5] = {2,2,6,5,4};//物品的重量
        int v[5] = {6,3,5,4,6};//物品的价值
        int x[5];//物品的选取状态
        int n = 5;
        int C=10;//背包最大容量

        s = KnapSack01v3(n, w, v, x, C);
        printf("%d\n", s);
        return ;
    }
}
#endif
```

Out[9]:

In [10]:

```
get01v3();
```

```
0       0       6       6       6       6       6       6       6       6       6
0       0       6       6       9       9       9       9       9       9       9
0       0       6       6       9       9       9       9       11      11      14
0       0       6       6       9       9       9       10      11      13      14
0       0       6       6       9       9       12      12      15      15      15
15
```

Out[10]:

```
(void) nullptr
```

其中的 f[j]=max(f[j],f[j-w[i]]) 一句恰就相当于我们的转移方程 f[i][j]=max(f[i-1][j],f[i-1][j-w[i]]) ，因为现在的 f[j-w[i]] 就相当于原来的 f[i-1][j-w[i]] 。如果将 V 的循环顺序从上面的逆序改成顺序的话，那么则成了 f[i][j] 由 f[i][j-w[i]] 推知，与本题意不符，但它却是另一个重要的背包问题（完全背包）最简捷的解决方案，故学习只用一维数组解 01 背包问题是十分必要的。

事实上，使用一维数组解 01 背包的程序在后面会被多次用到，以后的代码中就直接调用不再加以说明。

下面再给出一段代码，注意和上面的代码有什么不同

In [2]:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    int KnapSack01v4(int n, int w[], int v[], int x[], int C){
        int V[200];//前i个物品装入容量为j的背包中获得的最大价值
        int i, j;
        memset(V,0,sizeof(V));   //初始化数组为 0
        for (int i = 0; i < n; i++){
            for (int j = C; j >= w[i]; j--){    //循环的结束下线，是和上一个代码的主要区别，一个优化
                if(j>=w[i]){   //能装下，有等号
                    V[j] = max(V[j], V[j - w[i]] + v[i]);
                }else{
                    V[j]=V[j];
                }

            }
            for (int k = 0; k < C+1; k++){
                printf("%d\t",V[k]);
            }
            printf("\n");
        }
        return V[C];
    }
}
#endif
```

Out[2]:

In [3]:

```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    void get01v4(){
        int s;//获得的最大价值
        int w[5] = {2,2,6,5,4};//物品的重量
        int v[5] = {6,3,5,4,6};//物品的价值
        int x[5];//物品的选取状态
        int n = 5;
        int C=10;//背包最大容量

        s = KnapSack01v4(n, w, v, x, C);
        printf("%d\n", s);
        return ;
    }
}
#endif
```

Out[3]:

In [4]:
```
get01v4();
```

```
0       0       0       5       5       5
0       0       10      10      10      15
15
```

Out[4]:
```
(void) nullptr
```

注意这个过程里的处理与前面给出的代码有所不同。前面的示例程序写成 j=V...0 是为了在程序中体现每个状态都按照方程求解了，避免不必要的思维复杂度。而这里既然已经抽象成看作黑箱的过程了，就可以加入优化。费用为 w[i] 的物品不会影响状态 f[0...j-1] ，这是显然的。

初始化的细节问题

我们看到的求最优解的背包问题题目中，事实上有两种不太相同的问法。有的题目要求"恰好装满背包"时的最优解，有的题目则并没有要求必须把背包装满。这两种问法的区别是在初始化的时候有所不同。
如果是第一种问法，要求恰好装满背包，那么在初始化时除了 f[0] 为0其它 f[1...V] 均设为$-\infty$，这样就可以保证最终得到的 f[N] 是一种恰好装满背包的最优解。
如果并没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将 f[0...V] 全部设为0。

为什么呢？可以这样理解：初始化的f数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满，那么此时只有容量为0的背包可能被价值为0的nothing“恰好装满”，其它容量的背包均没有合法的解，属于未定义的状态，它们的值就都应该是$-\infty$了。如果背包并非必须被装满，那么任何容量的背包都有一个合法解“什么都不装”，这个解的价值为0，所以初始时状态的值也就全部为0了。这个小技巧完全可以推广到其它类型的背包问题，后面也就不再对进行状态转移之前的初始化进行讲解。

一个常数优化

前面的代码中有 for(j=V...w[i]) ，还可以将这个循环的下限进行改进。

由于只需要最后 f[j] 的值，倒推前一个物品，其实只要知道 f[j-w[n]] 即可。以此类推，对以第j个背包，其实只需要知道到 f[j-sumw[j...n]] 即可，即代码可以改成

```
for (int i = 1; i <= n; i++) {
    int bound = max(V - sum{w[i]...w[n]}, v[i]);
    for (int j = V; j >= bound, j--)
        f[j] = max(f[j], f[j - w[i]] + v[i]);
```

对于求sum可以用前缀和，这对于V比较大时是有用的。

下面的代码现在是错的

In [2]:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    int KnapSack01v5(int n, int w[], int v[], int x[], int C){
        int V[200];//前i个物品装入容量为j的背包中获得的最大价值
        int sum[200];
        sum[0]=w[0];
        for(int z=1;z<n;++z){
            sum[z]=sum[z-1]+w[z];
        }
        int i, j;
        memset(V,0,sizeof(V));   //初始化数组为 0
        for (int i = 0; i < n; i++){
            int bound = max(C - (sum[n]-(i>0?sum[i-1]:sum[0])), v[i]);
            for (int j = C; j >= bound; j--){   //这个答案是错的，下界太大了
                if(j>=w[i]){   //能装下，有等号
                    V[j] = max(V[j], V[j - w[i]] + v[i]);
                }else{
                    V[j]=V[j];
                }

            }
            for (int k = 0; k < C+1; k++){
                printf("%d\t",V[k]);
            }
            printf("\n");
        }
        return V[C];
    }
}
#endif
```

Out[2]:

In [3]:

```
#include <stdio.h>
#include <stdlib.h>
#ifdef __cplusplus  //曾经的C/C++，使用这个宏
extern "C" {
    void get01v5(){
        int s;//获得的最大价值
        int w[5] = {2,2,6,5,4};//物品的重量
        int v[5] = {6,3,5,4,6};//物品的价值
        int x[5];//物品的选取状态
        int n = 5;
        int C=10;//背包最大容量

        s = KnapSack01v5(n, w, v, x, C);
        printf("%d\n", s);
        return ;
    }
}
#endif
```

Out[3]:

In [4]:

```
get01v5();
```

```
0       0       6       6       6       6       6       6       6       6       6
0       0       6       6       9       9       9       9       9       9       9
0       0       6       6       9       9       9       9       11      11      14
0       0       6       6       9       9       9       10      11      13      14
0       0       6       6       9       9       12      12      15      15      15
15
```

Out[4]:

```
(void) nullptr
```

小结

01背包问题是最基本的背包问题，它包含了背包问题中设计状态、方程的最基本思想，另外，别的类型的背包问题往往也可以转换成01背包问题求解。故一定要仔细体会上面基本思路的得出方法，状态转移方程的意义，以及最后怎样优化的空间复杂度。

练习

Luogu 2925 干草出售 (https://www.luogu.org/problemnew/show/P2925)

Luogu 1616 疯狂的采药 (https://www.luogu.org/problemnew/show/P1616)

HDU 3466 Proud Merchants (http://acm.hdu.edu.cn/showproblem.php?pid=3466)

[USACO08DEC]干草出售Hay For Sale

题目描述

农民john面临一个很可怕的事实，因为防范失措他存储的所有稻草给澳大利亚蟑螂吃光了，他将面临没有稻草喂养奶牛的局面。在奶牛断粮之前，john拉着他的马车到农民Don的农场中买一些稻草给奶牛过冬。已知john的马车可以装的下C(1 <= C <=50,000)立方的稻草。
农民Don有H(1 <= H <= 5,000)捆体积不同的稻草可供购买，每一捆稻草有它自己的体积(1 <= V_i <= C)。面对这些稻草john认真的计算如何充分利用马车的空间购买尽量多的稻草给他的奶牛过冬。
现在给定马车的最大容积C和每一捆稻草的体积Vi，john如何在不超过马车最大容积的情况下买到最大体积的稻草？他不可以把一捆稻草分开来买。

输入输出格式

输入格式：

第一行两个整数，分别为C和H
第2..H+1行:每一行一个整数代表第i捆稻草的体积Vi

输出格式：

一个整数，为john能买到的稻草的体积。

输入输出样例

输入样例#1：

```
7 3
2
6
5
```

输出样例#1：

```
7
```

疯狂的采药

题目背景

此题为NOIP2005普及组第三题的疯狂版。

此题为纪念LiYuxiang而生。

题目描述

LiYuxiang是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同种类的草药，采每一种都需要一些时间，每一种也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”

如果你是LiYuxiang，你能完成这个任务吗？

此题和原题的不同点：

1.每种草药可以无限制地疯狂采摘。

2.药的种类眼花缭乱，采药时间好长好长啊！师傅等得菊花都谢了！

输入输出格式

输入格式：

输入第一行有两个整数T（1 <= T <= 100000）和M（1 <= M <= 10000），用一个空格隔开，T代表总共能够用来采药的时间，M代表山洞里的草药的数目。接下来的M行每行包括两个在1到10000之间（包括1和10000）的整数，分别表示采摘某种草药的时间和这种草药的价值。

输出格式：

输出一行，这一行只包含一个整数，表示在规定的时间内，可以采到的草药的最大总价值。

输入输出样例

输入样例#1：

```
70 3
71 100
69 1
1 2
```

输出样例#1：

```
140
```

[NOIP2005] 采药(原题)

★ 输入文件：medic.in 输出文件：medic.out 简单对比
时间限制：1 s 内存限制：128 MB

问题描述

辰辰是个天资聪颖的孩子，他的梦想是成为世界上最伟大的医师。为此，他想拜附近最有威望的医师为师。医师为了判断他的资质，给他出了一个难题。医师把他带到一个到处都是草药的山洞里对他说：“孩子，这个山洞里有一些不同的草药，采每一株都需要一些时间，每一株也有它自身的价值。我会给你一段时间，在这段时间里，你可以采到一些草药。如果你是一个聪明的孩子，你应该可以让采到的草药的总价值最大。”

如果你是辰辰，你能完成这个任务吗？

输入文件

输入文件的第一行有两个整数 T （ 1 <= T <= 1000 ）和 M （ 1 <= M <= 100 ），用一个空格隔开， T 代表总共能够用来采药的时间， M 代表山洞里的草药的数目。接下来的 M 行每行包括两个在 1 到 100 之间（包括 1 和 100 ）的整数，分别表示采摘某株草药的时间和这株草药的价值。

输出文件

输出文件包括一行，这一行只包含一个整数，表示在规定的时间内，可以采到的草药的最大总价值。

样例输入

```
70 3
71 100
69 1
1 2
```

样例输出

```
3
```

数据规模

对于 30% 的数据， M <= 10 ；

对于全部的数据， M <= 100 。

In [ ]: