2015 级算法第四次上机解题报告（助教版）

目 录

2015 级算法第四次上机解题报告（助教版）

马国瑞

一、引言

本次上机题目整体难度相较第一、第二次上机而言大了些，但比第三次上机难度小一点。这次上机平均分 2.72 分，大部分同学过题数目在 2~3 题之间（占总人数的 58.79%）

本次上机重点考查贪心算法及动态规划的知识。据一位同学的解题报告的总结，是“贪心算法初步&动态规划进阶”。这两种算法实际上是有共同之处的，二者都具备重叠子问题性质。贪心算法算是动态规划的一种特殊情况，因为贪心算法子问题得出的最优解一定被包含在整体最优解中，动态规划子问题的最优解不一定被包含在整体最优解中——其子问题的最优解是包含其下一层子子问题的最优解的，并不一定包含更深层子问题的最优解。因此，贪心算法不一定能得到最优解。例子参见《算法导论》第 16 章的 01 背包问题和分数背包问题的对比，这里不做解释。

这一份解题报告同样包含同学们上交的解题报告的内容，在此向提交解题报告的同学们表示感谢。

此外，欢迎大家对这份解题报告内容中出现的问题批评指正。

二、解题报告

A 怠惰的王木木 II

思路分析

贪心算法

当所有的纸币数目均为正整数时，可使用“先给 100 元，再给 50、20……最后再找一块钱”的贪心策略，肯定能得到最优解。

但是，当纸币数目变为非负整数时，由于 50 不是 20 的整数倍，因此在某些情况下贪心算法得不到最优解。比如给出的纸币数目分别为 10、0、0、3、1、0，要求找 60 元，实际的最小纸币数目为 3，而经过上述贪心算法得到的结果为 11.

因此，我们可以先进行把所有的纸币种类贪心求解，再忽略 50 元、20 元各做一次，取数据合法的情况下（即方案能完成该面值找零）三种方案的纸币数目最小值即可。时间复杂度为 O(n).

具体代码实现参见参考代码一。

动态规划法 Source: 15211051 王子烈

假设 f[i][v]表示选了前 i 种纸币，凑足 v 的面值所需的最小纸币数量，amount[i]为这种纸币的数量。

对每种纸币决策有 amount[i]+1 种选择办法，分别为不选，选 1 张，选 2 张……选 amount[i]张。在所有的选择方案里选择张数最小的那个。时间复杂度为 $O(n^2)$

转移方程为 $f[i][v]=min\{f[i-1][v-k\cdot value[i]],\ 0\le k\le amount[i]\}$

参考代码一

```
//Source: Magry
//事实上三次贪心部分的代码长度可以缩小一些
#include<iostream>
#include<cstdio>
#define INF 0x7FFFFFFF
using namespace std;
int main()
{
	int a[6],money,ans;
	while(cin>>a[0]){
	   //此处可以注意多组数据第一行输入多个数的代码姿势
	      for(int i=1;i<6;i++) cin>>a[i];
	      int x;
	      cin>>money;
	      x=money;
	      int ans0=0,ans1=0,ans2=0;
	   //flag1,flag2,flag3 分别代表三种情况的数据合法性变量
	      bool flag1=true,flag2=true,flag3=true;
	      int buf[6]={1,5,10,20,50,100};
	   //全部纸币遍历一遍
	      for(int i=5;i>=0;i--){
	            int cnt=money/buf[i];
	            if(cnt<a[i]){
	                  ans0+=cnt;
	                  money%=buf[i];
	            }
	            else{
	                  ans0+=a[i];
	                  money-=(buf[i]*a[i]);
	            }
	      }
	      if(money>0) flag1=false;
	   //忽略 50 元
```

```
    money=x;
    for(int i=5;i>=0;i--){
        if(i!=4){
            int cnt=money/buf[i];
            if(cnt<a[i]){
                ans1+=cnt;
                money%=buf[i];
            }
            else{
                ans1+=a[i];
                money-=(buf[i]*a[i]);
            }
        }
    }
    if(money>0) flag2=false;
//忽略 20 元
    money=x;
    for(int i=5;i>=0;i--){
        if(i!=3){
            int cnt=money/buf[i];
            if(cnt<a[i]){
                ans2+=cnt;
                money%=buf[i];
            }
            else{
                ans2+=a[i];
                money-=(buf[i]*a[i]);
            }
        }
    }
    if(money>0) flag3=false;
    //不考虑没法找钱的情况，题设 3 种情况至少有一种存在
    int ans=ans0;
    if(!flag1) ans=INF;
    if(ans1<ans&&flag2){
        ans=ans1;
    }
    if(ans2<ans&&flag3){
        ans=ans2;
    }
    cout<<ans<<endl;
```

```
    }
}
```

参考代码二

```
//Source: 15151165 马宇航
#include <cstdio>
#define INF 99999999;
int money[6];
int Dp[1007];
const int value[]={1,5,10,20,50,100};
int main(){
    int x;
    while(~scanf("%d",&money[0])){
        for(int i=1;i<6;i++)
            scanf("%d",&money[i]);
        scanf("%d",&x);
        for(int i=1;i<=x;i++)
            Dp[i]=INF;
        //多重背包的三重 for 循环方法
        for(int i=0;i<6;i++)
            for(int j=1;j<=money[i];j++)
                for(int k=x;k>=value[i];k--){
                    if(Dp[k]>Dp[k-value[i]]+1)
                        Dp[k]=Dp[k-value[i]]+1;
                }
        printf("%d\n",Dp[x] );
    }
}
```

B Magry 的朋友很多 - Wonderland 的邀请篇

思路分析

本题的贪心策略是：优先选择结束时间最早的活动，则选中的活动一定在最大兼容活动子集中。这个结论可以使用数学归纳法证明，具体过程不在此赘述。

确定贪心策略之后，在代码实现层面有一处难点是排序问题。由于 $O(n^2)$的排序时间复杂度对于 n 比较大的情况花费的时间很多，因此需要定义结构体，自定义比较函数，然后使用 sort()对结构体排序。具体实现参见参考代码。

算法分析

本算法排序时间复杂度 O(nlgn), 贪心选择时间为 O(n).

参考代码

```
//Source: Magry
#include<iostream>
#include<algorithm>
#include<cstdio>
using namespace std;
typedef struct
{
    int begin;
    int end;
}Time;
Time t[100010];
bool cmp2(Time t1, Time t2)    //比较函数，结束时间越早优先级越高
{
    if(t1.end<t2.end)
        return 1;
    else
        return 0;
}
int main()
{
    int n;
    while(cin>>n)
    {
        for(int i=0;i<n;i++)
            cin>>t[i].begin>>t[i].end;
        //排序
```

```
        sort(t,t+n,cmp2);
        //贪心选择
        int cnt=1;
        int minend=t[0].end;
        for(int i=1;i<n;i++){
            if(t[i].begin>=minend){
                cnt++;
                minend=t[i].end;
            }
        }
        cout<<cnt<<endl;
    }
}
```

C 在下废灵根

思路分析

根据题意我们发现，F 是取得的每种灵物优劣程度的最小值，C 是取得的每种灵物的获取难度之和，F 与 C 并没有线性相关性，且每种灵物的优劣程度和获取难度一一对应。于是我们可以枚举所有输入的优劣程度（或者针对给定 V 的数据范围一一枚举），设为 F_i，然后针对各种灵物优劣程度不小于 Fi 的获取方式贪心/动态规划求解最小获取难度，最后迭代取最大值。

这里需要特别注意需要剔除某种灵物所有的 V 均小于 F_i 的情况。

针对各种灵物优劣程度不小于 Fi 的获取方式求解最小获取难度的方法如下：

动态规划法 – 时间复杂度大致为 $O(n^3)$

Source: 15211041 朱辉

我们可以使优劣程度暂定。本题中我们可以一边输入一边处理，因为在处理输入时只需要用到前 i-1 种灵根的信息。

状态数组 dp [i][j] 表示选择了 i 个灵根中优劣程度达到 j 的最小难度。 很容易得到转移方程 ：dp[i][j]=min（dp[i][j]，dp[i-1][k]+H），这里特别注意选择 j 的时候的大小情况。

具体实现方式参见参考代码一。

贪心法 – 时间复杂度大致为 $O(n^3)$

对每种灵物取在优劣程度不小于 F_i 的情况下获取难度的最小值，然后相加即可（思路同第三次上机 A 题 Nintendo Switch 生产车间）。

具体实现方式参见参考代码二。

参考代码一

```
//Source: 15211041 朱辉
#include<cstdio>
#include<cstring>
#include<iostream>
using namespace std;
const int INF=0x3f3f3f3f;

int dp[105][1005];//记录前 i 个灵根中优劣程度达到 j 的最小难度

int main()
{
```

```
int n;//n种灵物
int MaxV;//记录最大的优劣程度
while(~scanf("%d",&n))
{
    MaxV=0;
    for(int i=1;i<=n;i++)//初始化
        for(int j=0;j<1005;j++)
            dp[i][j]=INF;

    for(int i=1;i<=n;i++)
    {
        int num;
        scanf("%d",&num);
        for(int j=1;j<=num;j++)
        {
            int V,H;
            scanf("%d%d",&V,&H);
            if(MaxV<V) MaxV=V;
            if(i==1)//第一个灵物
                dp[1][V]=min(dp[1][V],H);
            else
            {
                for(int k=0;k<=MaxV;k++)
                {
                    if(dp[i-1][k]!=INF)//前 i-1 个灵根中优劣程度为 k 的难度
                    {
                        if(k<=V)//程度优，更新 dp[i][k]
                            dp[i][k]=min(dp[i][k],dp[i-1][k]+H);
                        else//程度劣，更新 dp[i][V]
                            dp[i][V]=min(dp[i][V],dp[i-1][k]+H);
                    }
                }
            }
        }
    }
    double ans=0.0;
    for(int i=0;i<=MaxV;i++)
    {
        if(dp[n][i]!=INF)//优劣程度 i
        {
            double k=(double)i/dp[n][i];//计算优劣程度为 i/其最小难度
```

```
            if(k>ans) ans=k;
        }
    }
    printf("%.2lf\n",ans);
  }
  return 0;
}
```

参考代码二

```
//Source: Magry
#include<iostream>
#include<iomanip>
#include<cstdio>
#include<cstring>
using namespace std;
struct G
{
    int v;
    int h;
};
G g[101][101];
int fs[10010];
int gb[101];
bool flag[101];
int main()
{
    int n;
    while(cin>>n){
        int cnt=0;
        for(int i=0;i<n;i++){
            cin>>gb[i];
            for(int j=0;j<gb[i];j++){
                cin>>g[i][j].v>>g[i][j].h;
                fs[cnt++]=g[i][j].v;  //fs 数组用于存储输入所有的 V
            }
        }
        double ans=0;
        for(int ti=0;ti<cnt;ti++){
            int c=0;
            memset(flag,false,sizeof(flag));
            for(int i=0;i<n;i++){
                //贪心法，选取获取难度的最小值
```

```
            int minc=0x7FFFFFFF;
            for(int j=0;j<gb[i];j++){
                if(g[i][j].v>=fs[ti]&&g[i][j].h<minc){
                    minc=g[i][j].h;
                    flag[i]=true;
                }
            }
            c+=minc;
        }
        //合法性检查，如果有一种灵物的优劣程度始终小于 fs[ti]则舍去
        bool ansflag=true;
        for(int i=0;i<n;i++){
            ansflag&=flag[i];
        }
        if(ansflag){
            double buf=(double)fs[ti]/(double)c;
            if(buf>ans) ans=buf;
            //printf("%.2lf %d %d\n",buf,fs[ti],c);
        }
    }
    cout<<setprecision(2)<<fixed<<ans<<endl;
  }
}
```

D 机器人装配

思路分析

典型的动态规划经典问题——流水线调度问题，不过需要大家仔细根据样例解释读懂样例输入代表的含义。这里仅展示自顶向下的分析方法。

令 cost[i][j]为到达第 i 条线的第 j 个装配站时所需最小的时间总和，a[i][j]存下各个点的耗费，b[i][k][j]表示从第 i 条流水线 k 道工序转到第 j 条流水线的耗费。

这里对流水线 i 进行分析：

1）、通过流水线 i 装配站 n-1，然后直接通过流水线 i 的 n；

2）、通过流水线 i 的装配站 n-1，然后从流水线 i 切换到流水线 j，之后通过流水线 k 的 n；

所以得出下面的状态转移方程：

cost[i][j]=min(cost[i][j]，cost[k][j-1]+b[k][j-1][i]+a[i][j]) (k=1,2,3.....n)

最后结果是所有 cost[k][m]的最小值，其中 k=1,2,3,…,n.

（部分参考自：15211041 朱辉）

算法分析

时间复杂度为 $O(mn^2)$，空间复杂度 $O(mn^2)$.

参考代码

```
//Source: Magry
#include<cstdio>
#include<iostream>
#include<cstring>
#include<cmath>
#define INF 0x7FFFFFFF
using namespace std;
int a[1001][11][11];
int dp[1001][11];
bool ok[1001][11];
int t[1001][11];
int main()
{
    int n,m;
    while(cin>>n>>m){
        memset(dp,0,sizeof(dp));
```

```
        memset(ok,false,sizeof(ok));
        for(int mi=1;mi<m;mi++){
            for(int i=1;i<=n;i++)
                cin>>t[mi][i];
            for(int i=1;i<=n;i++){
                for(int j=1;j<=n;j++)
                    cin>>a[mi][i][j];
            }
        }
        for(int i=1;i<=n;i++)
            cin>>t[m][i];
        //导入第一个装配站的数据
        //这里与思路分析不同的是 dp[i][j]代表第 i 个装配站和第 j 条线
        for(int i=1;i<=n;i++)
            dp[1][i]=t[1][i];
        //处理后续数据
        for(int mi=2;mi<=m;mi++){
            for(int i=1;i<=n;i++){
                dp[mi][i]=INF;
            }
            for(int i=1;i<=n;i++){
                for(int j=1;j<=n;j++){
                    if(dp[mi][i]>dp[mi-1][j]+a[mi-1][j][i]+t[mi][i])
                        dp[mi][i]=dp[mi-1][j]+a[mi-1][j][i]+t[mi][i];
                }
            }
        }
        //得出最后结果
        int ans=dp[m][1];
        for(int i=2;i<=n;i++){
            if(ans>dp[m][i]) ans=dp[m][i];
        }
        cout<<ans<<endl;
    }
}
```

E 可得长生否

思路分析

本题可通过枚举每一个起始点进行 DP+DFS（深度优先搜索）求该点出发的最长合法路径长度，取最大值。最大值可以用公式表示为：

ans=max(ans,dfs(i,j))，1≤i≤n，1≤j≤m

DP+DFS 过程基于如下考虑：

- 最长合法路径的长度对于每个点肯定都是唯一的值
- 以其他点任何点为起点的最长路径如果经过这个点，必然经过这个点的其中一条最长合法路径。

这一过程的状态转移方程如下：

dp[x][y]=max(dp[x-1][y],dp[x+1][y],dp[x][y-1],dp[x][y+1])+1

需要注意的是：这一过程边走边保存当前点的最长合法路径长度，以后再经过这个点时，直接返回这个值即可，不必重复递归。

（部分参考自：15211090 杨承昊）

算法分析

本题算法总的时间复杂度最好情况 $O(n^2)$，最坏情况 $O(n^4)$

参考代码

```
//Source: Magry
#include<iostream>
#include<cstring>
#include<cmath>
using namespace std;
int a[101][101];
int s[101][101];
int px[]={1,0,-1,0};
int py[]={0,1,0,-1};
bool sol[101][101];
bool ok(int i, int j, int n, int m)  //边界检测函数
{
    if(i<0||j<0||i>=n||j>=m)
        return 0;
    else
        return 1;
```

```
}
int solve(int i, int j, int n, int m)
{
    //不重复递归。大家会发现 n*m 次 dfs 过程没有分别对这个数组初始化，
    //是因为以其他点任何点为起点的最长路径如果经过这个点，
    //必然经过这个点的其中一条最长合法路径，
    //并且这个点出发的最长合法路径唯一，因此可借此消除重复递归。
    if(sol[i][j])
        return s[i][j];
    int ps=0,pt=0;
    int sum=1;
    for(int r=0;r<4;r++){
        int s=i+px[r];
        int t=j+py[r];
        if(ok(s,t,n,m)&&a[s][t]>a[i][j]){
            int buf=1+solve(s,t,n,m);
            if(sum<buf) sum=buf;
        }
    }
    sol[i][j]=true;
    s[i][j]=sum;
    return sum;
}

int main()
{
    int n,m;
    while(cin>>n>>m){
        memset(s,0,sizeof(s));
        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++)
                cin>>a[i][j];
        }
        memset(sol,false,sizeof(sol));
        int ans=-1;
        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                int buf=solve(i,j,n,m);
                if(buf>ans)
                    ans=buf;
            }
```

```
        }
        cout<<ans<<endl;
    }
}
```

F Magry 的烦恼

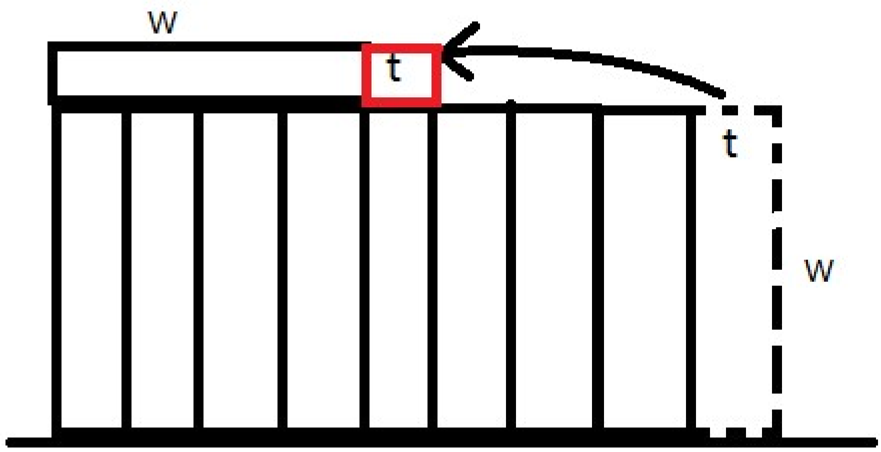
思路分析

本题解题方法很多，01 背包法、贪心算法、搜索剪枝等方法均可做。这里介绍其中的三种方法。

方法一：套用 01 背包问题

Source: 15211027 李熙

这个问题可以看成是一个 01 背包问题。



首先分析一下问题，我们可已经将问题抽象为：将全部书竖着放在下面，记所有书的厚度和为 sumT。如果从下面拿一本书放到上面，相当于，下面的总厚度减去 t_i，上面宽度加上 w_i，**等价于下面的厚度不变，上面的厚度加 w_i+t_i**，最后需要下面所有书的厚度和最小，等同于拿到上面的书的总厚度最大。

可以看到这和 01 背包是有一些类似的，每本书就是我们需要放到背包里的东西，每本书只有一个，**背包的总容量就是 sumT**，也就是我们能横放图书的最大长度，**我们每次向上放一本书，所需要的花费是 t_i+w_i**（为保证 **sumT** 不变），我们**获取的价值就是 t_i**，我们获取的 t_i 之和的最大值，就是我们能拿到上面的书的之和，**当价值最大时，也就是下面剩余的树厚度最小的时候，就是我们最终需要求的东西。**

时间复杂度为 $O(n^2)$。具体代码实现参见参考代码一。

方法二：贪心策略

Source: 15211095 吴星哲

首先把所有的书都放在下面，求出厚度总和。随后不断拿走书本放到上面，直到再拿书就会下面的宽度小于上面的宽度为止。拿书时，当书厚度相等时，优先拿宽度小的书，这就是贪心选择性质。当下面的

书只有一种厚度时，贪心性质很简单。而当 1、2 二种厚度都有时，分为以下几种情况：

1.当厚度为 1 的书中，二本宽度最小的书宽度之和大于等于厚度为 2 的书宽度最小值时。

a.上面剩余宽度足够放厚度为 2 的书最小宽度，把书放上去。

b.上面宽度仅够放厚度为 1 的书最小宽度，把它放上去。

c.都不够，说明已到极限，结束贪心过程。

2.当厚度为 1 的书中，二本宽度最小的书宽度之和小于厚度为 2 的书宽度最小值时。

a.上面宽度足够放厚度 1 最小宽度，把它放上去。（只放 1 本，不放 2 本，这是最大的坑。考虑特殊情况，上面宽度足够放厚度为 1 和 2 的二本最窄的书，但是放了二本厚度为 1 最窄的书，就不能再放了。显然第一种方法更优。）

b.达到极限，结束贪心过程。

依次反复，直到不能再放。此时下面宽度就是最小宽度。

给书按升序排序，时间复杂度为 O(nlogn)。贪心过程时间复杂度为 O(n)。所以总的时间复杂度为 O(nlogn)。

具体代码实现参见参考代码二。

方法三：贪心算法和迭代法结合起来的方法

我们可以把书分成这两类：厚度为 1 的一类，厚度为 2 的一类。

由此，这道题的题意可以转化为：对于 n 本书，挑走 i 本第一类书和 j 本第二类书竖着摆，剩下的 n-i-j 本书横着摆，求竖着摆的书厚度总和为 i+2j 时余下 n-i-j 本书的最小宽度之和（设为 K[i+2j]）。

这里满足一点贪心策略：优先将每类书宽度大的拿出来竖着摆，剩下的书宽度相对小一些，因此宽度之和也小些。因此，在 i 与 j 相同的情况下，当取走的竖着摆的 i 本第一类书和 j 本第二类书宽度分别是第一类书的前 i 大和第二类书的前 j 大时，剩下的书的宽度之和最小。

对于宽度为 i+2j 的情况：由于很可能不止一种情况会出现厚度之和相同的情况，因此可以通过如下状态转移方程迭代求解得到当宽度为 i+2j 时余下书的最小宽度之和 K[i+2j]：

```
K[i+2j] = min( K[i+2j], (s1[cnt1]-s1[s])+(s2[cnt2]-s2[t]) )
```

这里 s1[i]代表厚度为 1 的所有书的集合按宽度从大到小排序后前 i 本书之和，s2[i] 代表厚度为 2 的所有书的集合按宽度从大到小排序后前 i 本书之和。

最后，求出满足 i+2j≥K[i+2j] 的最小 i+2j 的值即可。

同样的，给书按升序排序需要的最小时间复杂度为 O(nlgn)，求前 i 项和需要的时间复杂度为 O(n)，迭代求解需要的时间复杂度为 $O(n^2)$，比较的时间复杂度为 O(n)，总的时间复杂度为 $O(n^2)$.

这里如果求解区间和过程使用暴力方法，则时间复杂度会变为 $O(n^3)$.

具体代码实现参见参考代码三。

参考代码一

```
//Source: 15211027 李熙
#include<iostream>
#include<cstdio>
#include<cstring>
#define MAXSIZE 110
#define MAXN 10010
using namespace std;

int t[MAXSIZE], w[MAXSIZE];//厚度，宽度
int f[MAXN];//f[i],总容量为 i 时的最大价值，即拿上去的书的总厚度

int main() {
    int n;
    while (~scanf("%d", &n)) {
        int sumT = 0;//总容量
        for (int i = 1; i <= n; i++) {
            scanf("%d%d", &t[i], &w[i]);
            sumT += t[i];
        }
        memset(f, 0, sizeof(f));
        //0-1 背包
        for (int i = 1; i <= n; i++) {
            //t[i] + w[i],花费
            for (int j = sumT; j >= (t[i] + w[i]); j--) {
                if (f[j - (t[i] + w[i])] + t[i] > f[j]) {
                    f[j] = f[j - (t[i] + w[i])] + t[i];
                }

            }
        }
        int ans = sumT - f[sumT];
        printf("%d\n", ans);
    }
}
```

参考代码二

```
//Source: 15211095 吴星哲
#include<cstdio>
#include<algorithm>
using namespace std;

int book1[105];//厚度为 1 的书宽度
int book2[105];//厚度为 2 的书宽度

int main()
{
    int n;
    int t,w;
    while( scanf( "%d",&n )!=EOF )
    {
        int sum=0;//下面书总宽度
        int tmp=0;//上面书总宽度
        int top1=0,top2=0;

        for( int i=0;i<n;i++ )
        {
            scanf( "%d %d",&t,&w );
            if( t==1 )
                book1[top1++]=w;
            else
                book2[top2++]=w;
            sum+=t;//所有书厚度之和
        }
        sort(book1,book1+top1);
        sort(book2,book2+top2);//宽度升序排列

        int cnt1=0,cnt2=0;
        while( sum>tmp )
        {
            if( cnt1<top1&&cnt2<top2 )//二种厚度的书下面都有
            {
            if( book1[cnt1]+book1[cnt1+1]<book2[cnt2])
            { if( sum-1>=tmp+book1[cnt1] )
                {
                    sum--;
                    tmp+=book1[cnt1];
```

```
        cnt1++;
    }//厚度为 1 宽度最小的书，放到上面
    else
        break;
}

else if( sum-2>=tmp+book2[cnt2] )
{
    sum-=2;
tmp+=book2[cnt2];
cnt2++;
}//厚度为 2 最窄的书放到上面

else if( sum-1>=tmp+book1[cnt1] )
{
    sum--;
    tmp+=book1[cnt1];
    cnt1++;
}
else
    break;
}

else if( cnt2<top2 )//只有厚度为 2 的书
{
    if( sum-2>=tmp+book2[cnt2] )
    {
        sum-=2;
        tmp+=book2[cnt2];
        cnt2++;
    }
    else
        break;
}
else if( cnt1<top1 )//只有厚度为 1 的书
{
    if( sum-1>=tmp+book1[cnt1] )
    {
        sum--;
        tmp+=book1[cnt1];
        cnt1++;
```

```
                }
                else
                    break;
            }
        }

        printf( "%d\n",sum);
    }
}
```
参考代码三

```
//Source: Magry
#include<iostream>
#include<algorithm>
#include<cstring>
#include<cstdio>
using namespace std;
struct Book
{
    int t;
    long long w;
};
Book b[101];
bool cmp3(long long a, long long b)
{
    if(a>b) return 1;
    else return 0;
}
long long dp[210];
bool solve[210];
int main()
{
    int n;
    while(cin>>n){
        long long v1[101],v2[101]; //v1,v2 分别代表第一、第二类书宽度
        int cnt1=0,cnt2=0;  //cnt1,cnt2 分别代表第一、第二类书个数
        int sumt=0;      //统计所有书本的厚度之和
        for(int i=0;i<n;i++){
            cin>>b[i].t>>b[i].w;
            sumt+=b[i].t;
            if(b[i].t==1) v1[cnt1++]=b[i].w;
            else v2[cnt2++]=b[i].w;
```

```
        }
        //使用了贪心策略，两类书从大到小排序
        sort(v1,v1+cnt1,cmp3);
        sort(v2,v2+cnt2,cmp3);
        memset(dp,0,sizeof(dp));
        memset(solve,false,sizeof(solve));
        //求解区间和
        long long s1[101],s2[101];
        s1[1]=v1[0];
        s2[1]=v2[0];
        s1[0]=0;
        s2[0]=0;
        for(int i=2;i<=cnt1;i++){
            s1[i]=s1[i-1]+v1[i-1];
        }
        for(int i=2;i<=cnt2;i++){
            s2[i]=s2[i-1]+v2[i-1];
        }
        //迭代法求解
        for(int i=0;i<=cnt1;i++){
            for(int j=0;j<=cnt2;j++){
                int s=i;
                int t=j;
                if(!solve[s+t*2]||dp[s+t*2]>((s1[cnt1]-s1[s])+(s2[cnt2]-
s2[t]))){
                    dp[s+t*2]=(s1[cnt1]-s1[s])+(s2[cnt2]-s2[t]);
                    solve[s+t*2]=true;
                }
            }
        }
        int ans=-1;   //求解满足 dp[i]<=i 的最小的 i
        for(int i=1;i<=sumt;i++){
            if(dp[i]<=i&&solve[i]){
                ans=i;
                break;
            }
        }
        if(ans==-1) ans=sumt;
        cout<<ans<<endl;
    }
}
```