背包问题

- 01背包
- 完全背包
- 多重背包
  - 二进制优化
  - 单调队列优化
  - 变式题　链接
  - 可行性多重背包
- 多维背包
  - 小变式
- 混合背包
- 分组背包
- 有依赖的背包
- 泛化物品的背包
- 树上分组背包
- 一些神奇的背包
  - 砝码称重

背包问题

01背包

> O(n*m)

```
#include<iostream>
#include<cstdio>
using namespace std;
int w[105],val[105];
int dp[105][1005];
int main()
{
    int t,m,res=-1;
    scanf("%d%d",&t,&m);
    for(int i=1;i<=m;i++)
    {
        scanf("%d%d",&w[i],&val[i]);
    }
    for(int i=1;i<=m;i++)
        for(int j=t;j>=0;j--)
        {
            if(j>=w[i])
            {
                dp[i][j]=max(dp[i-1][j-w[i]]+val[i],dp[i-1][j]);
            }
            else
            {
                dp[i][j]=dp[i-1][j];
            }
        }
    printf("%d",dp[m][t]);
    return 0;
}
```

> 滚动数组优化

```
#include<iostream>
#include<cmath>
#include<cstdio>
#include<vector>
#include<algorithm>
#include<cstring>
using namespace std;
#define ll long long
const int maxn=1e4+10;
ll dp[maxn];    //dp[i][j]为讨论物品i时,剩余容量为j时的最大价值
```

```
//由于由上而下更新，可以去掉一维直接更新
ll tim[maxn],v[maxn]; //tim物品容量,v物品价值
int main()
{
    int t,m;
    cin>>t>>m;
    for(int i=1;i<=m;i++)
    {
        scanf("%lld%lld",&tim[i],&v[i]);
    }
    for(int i=1;i<=m;i++)
    {
        for(int j=t;j>=tim[i];j--)
        {
            dp[j]=max(dp[j-tim[i]]+v[i],dp[j]);
        }
    }
    cout<<dp[t];
}
```

完全背包

> 有时会出现当重量大于等于h，小于等于h与价格最大值的和时，最小值都有可能出现，不仅存于重量等于h的情况，此时需要扩大dp范围

```
#include<iostream>
#include<cstdio>
#define ll long long
using namespace std;
const int maxn=1e4+10;
int dp[maxn],p[maxn],t[maxn];
int main()
{
    int m,n;
    cin>>m>>n;
    for(int i=1;i<=n;i++)
    {
        cin>>p[i]>>t[i];
    }
    for(int i=1;i<=n;i++)
    {
        for(int j=0;j<=m-t[i];j++)
        {
            dp[j+t[i]]=max(dp[j+t[i]],dp[j]+p[i]);
        }
        //下面的也可以
//    for(int j=t[i];j<=m;j++)
//    {
//      dp[j]=max(dp[j],dp[j-t[i]]+p[i]);
//    }
    }
    cout<<dp[m];
}
```

多重背包

> 多重背包可以看作由很多个01背包组成

[题目链接洛谷P1833]

```
#include<iostream>
#include<cstdio>
#include<fstream>
#define ll long long
using namespace std;
const int maxn=1e4+10;
int t[maxn],c[maxn],p[maxn];
int dp[maxn];
int main()
{
    int h1,m1,h2,m2,n;
    scanf("%d:%d %d:%d %d",&h1,&m1,&h2,&m2,&n);
    int w=(h2-h1-1)*60+(60+m2-m1);
    for(int i=1;i<=n;i++)
    {
        scanf("%d%d%d",&t[i],&c[i],&p[i]);
    }
    for(int i=1;i<=n;i++)
    {
        if(p[i]==0)
        {
            for(int j=0;j<=w-t[i];j++)
            {
                dp[j+t[i]]=max(dp[j+t[i]],dp[j]+c[i]);
            }
        }
        else{ //此处情况为多重背包，有i号物品由p[i]个
            for(int k=0;k<p[i];k++)
            {
                for(int j=w;j>=t[i];j--)
                {
                    dp[j]=max(dp[j],dp[j-t[i]]+c[i]);
                }
            }
        }
    }
    cout<<dp[w];
}
```

> 但当物品由很多个时，这种方法会很慢，因此需要优化

> 第一种为二进制优化，第二种为单调队列优化

二进制优化

讲解链接

> 多重背包转换成 01 背包问题就是多了个初始化，把它的件数C用分解成若干个件数的集合，这里面数字可以组合成任意小于等于C的件数，而且不会重复。

> 之所以叫二进制分解，是因为这样分解可以用数字的二进制形式来解释，比如：

> （1）7的二进制 7 = 111 它可以分解成 001 、010 、100 这三个数可以组合成任意小于等于7 的数，而且每种组合都会得到不同的数

> （2）15 = 1111 可分解成 0001 、 0010 、 0100 、 1000 四个数字

> （3）如果13 = 1101 则分解为 0001 0010 0100 0110前三个数字可以组合成7以内任意一个数，

> 加上 0110 = 6 可以组合成任意一个大于6 小于13的数，虽然有重复但总是能把 13 以内所有的数都考虑到了，基于这种思想去把多件物品转换为，多种一件物品，就可用01 背包求解了。

> 注：该例中0110可以13-1-2-4得到，即1101-0001-0010-0100=0110=6

> 该操作能够把k次01背包转化为logk次01背包

```
#include<iostream>
#include<cstdio>
#include<fstream>
#define ll long long
using namespace std;
const int maxn=1e4+10;
int t[maxn],c[maxn],p[maxn];
int dp[maxn];
int binary[maxn];
int qpow(int n,int m)
{
    int res=1;
    int multi=n;
    while(m!=0)
        {
            if((m&1)==1)
                {
                    res*=multi;
                }
            multi=multi*multi;
            m/=2;
        }
    return res;
}
int main()
{
    int h1,m1,h2,m2,n;
    scanf("%d:%d %d:%d %d",&h1,&m1,&h2,&m2,&n);
    int w=(h2-h1-1)*60+(60+m2-m1);
```

```
    for(int i=1;i<=n;i++)
    {
        scanf("%d%d%d",&t[i],&c[i],&p[i]);  //t为物品大小，c为物品价值，p为物品个数
    }
    for(int i=1;i<=n;i++)
    {
        if(p[i]==0)
        {
            for(int j=0;j<=w-t[i];j++)
            {
                dp[j+t[i]]=max(dp[j+t[i]],dp[j]+c[i]);
            }
        }
        else{
            int cnt=0;
            int sav=p[i];
            while(sav!=0) //转换为二进制
            {
                binary[cnt++]=sav%2;
                sav/=2;
            }
            int newt[maxn],newc[maxn];  //生成新的大件物品
            //以1101为例，此处实现生成0001、0010、0100物品
            //cnt-1为最高位
            for(int j=0;j<cnt-1;j++)
            {
                newt[j]=t[i]*(1<<j);
                newc[j]=c[i]*(1<<j);
            }
            sav=p[i];
            for(int j=0;j<cnt-1;j++)  //此处得到1101-0001-0010-0100=0110
            {
                sav-=(1<<j);
            }
            newt[cnt-1]=t[i]*sav;   //生成0110个物品的合成物品
            newc[cnt-1]=c[i]*sav;
            for(int k=0;k<cnt;k++)
            {
                for(int j=w;j>=newt[k];j--)
                {
                    dp[j]=max(dp[j],dp[j-newt[k]]+newc[k]);
                }
            }
        }
    }
    cout<<dp[w];
}
```

另一版本：

```
for(int i=1;i<=n;i++)
{
  for(int j=1;j<=num[i];j<<=1)
    //二进制每一位枚举.
    //注意要从小到大拆分
  {
    num[i]-=j;//减去拆分出来的
    new_c[++tot]=j*c[i];//合成一个大的物品的体积
    new_w[tot]=j*w[i];//合成一个大的物品的价值
  }
  if(num[i])//判断是否会有余下的部分.
    //就好像我们某一件物品为13,显然拆成二进制为1,2,4.
    //我们余出来的部分为6,所以需要再来一份.
  {
    new_c[++tot]=num[i]*c[i];
    new_w[tot]=num[i]*w[i];
    num[i]=0;
  }
}
```

单调队列优化

链接

> c[i]是体积，w[i]是价值，j是当前更新的已装体积
>
> 令d=c[i],a=j/c[i],b=j%c[i]，其中a为全选状况下的物品个数.则$j=a*d+b$
>
> 则带入原始的状态转移方程中 j−k∗d = a∗d+b−k∗d = (a−k)∗d+b
>
> 我们令$(a-k)=k'$，再回想我们最原始的状态转移方程中第二状态 : f[i][j−k∗c[i]]+k∗w[i] 代表选择k个当前i物品.
>
> j-k*c[i] = (a-k)*d+b = k'*d+b
>
> 而我们要求的状态也就变成了f[i][j]=max(f[i−1][k'∗d+b]+a∗w[i]−k'∗w[i])
>
> 而其中,我们的a∗w[i]为一个常量(因为a已知.)
>
> 所以我们的要求的状态就变成了f[i][j]=max(f[i−1][k'∗d+b]−k'∗w[i])+a∗w[i]
>
> 根据我们的k∈[1,lim]，k'∈[a−k,a]
>
> 当前的f[i][j]求解的就是为lim+1个数对应的f[i−1][k'∗d+b]−k'∗w[i]的最大值.
>
> (之所以为lim+1个数,是包括当前这个j,还有前面的物品数量.)
>
> 将f[i][j]前面所有的f[i−1][k'∗d+b]−k'∗w[i]放入一个队列.
>
> 那我们的问题就是求这个最长为lim+1的队列的最大值+a∗w[i].

```
#include<iostream>
#include<cstdio>
```

```cpp
#include<fstream>
#define ll long long
using namespace std;
const int maxn=1e4+10;
int t[maxn],c[maxn],p[maxn];
int dp[maxn];
int main()
{
    int h1,m1,h2,m2,n;
    scanf("%d:%d %d:%d %d",&h1,&m1,&h2,&m2,&n);
    int w=(h2-h1-1)*60+(60+m2-m1);    //背包容量
    for(int i=1;i<=n;i++)
    {
        scanf("%d%d%d",&t[i],&c[i],&p[i]);  //t为物品大小，c为物品价值，p为物品个数
        //物品个数可以理解为后续维护的单调队列的长度
    }
    for(int i=1;i<=n;i++)
    {
        if(p[i]==0)
        {
            for(int j=0;j<=w-t[i];j++)
            {
                dp[j+t[i]]=max(dp[j+t[i]],dp[j]+c[i]);
            }
        }
        else{
      int q1[maxn],q2[maxn];  //q1记录下标，q2记录数据
      for(int d=0;d<t[i];d++) //枚举余数
      {
        int l=1,r=0,maxp=(w-d)/t[i];
        for(int P=0;P<=maxp;P++)
        {
          int &x=dp[d+t[i]*P];    //相当于上文中单次循环的dp[m]
          while(l<=r && q1[l]<P-p[i]) l++;
          //维护区间长度，保证区间长度小于物品个数
          while(l<=r && x-c[i]*P>=q2[r] ) r--;
          q1[++r]=P;
          q2[r]=x-c[i]*P;
          x=max(x,q2[l]+c[i]*P);
        }
      }
    }
  }
    cout<<dp[w];
}
```

变式题 链接

正在上大学的小皮球热爱英雄联盟这款游戏，而且打的很菜，被网友们戏称为「小学生」。

现在，小皮球终于受不了网友们的嘲讽，决定变强了，他变强的方法就是：买皮肤！

小皮球只会玩 N 个英雄，因此，他也只准备给这N个英雄买皮肤，并且决定，以后只玩有皮肤的英雄。
这N个英雄中，第 i个英雄有 Ki款皮肤，价格是每款Ci Q 币（同一个英雄的皮肤价格相同）。

为了让自己看起来高大上一些，小皮球决定给同学们展示一下自己的皮肤，展示的思路是这样的：对于有皮肤的每一个英雄，随便选一个皮肤给同学看。

比如，小皮球共有 5 个英雄，这 5 个英雄分别有 0,0,3,2,4 款皮肤，那么，小皮球就有 3×2×4=24 种展示的策略。
现在，小皮球希望自己的展示策略能够至少达到M种，请问，小皮球至少要花多少钱呢?

输入格式

第一行，两个整数 N,M。

第二行，N个整数，表示每个英雄的皮肤数量 Ki。

第三行，N个整数，表示每个英雄皮肤的价格 Ci。

输出格式

一个整数，表示小皮球达到目标最少的花费。

```
#include<iostream>
#include<cstdio>
#include<fstream>
#define ll long long
using namespace std;
const int maxn=1e6+10;
ll dp[maxn];
ll k[maxn],c[maxn];
int main(){
  ll n,m;
  cin>>n>>m;
    ll qb=0;
  for(int i=1;i<=n;i++)
  {
    cin>>k[i];
  }
  for(int i=1;i<=n;i++)
  {
    cin>>c[i];
        qb+=c[i]*k[i];
  }
    dp[0]=1;
    for(int i=1;i<=n;i++){
        for(int w=qb;w>=0;w--){ //与模板题不同的是，先遍历容量，再遍历物品个数
            for(int j=0;j<=k[i];j++){
                if(w-j*c[i]>=0)
```

```
                dp[w]=max(dp[w],dp[w-j*c[i]]*j);
            }
        }
    }
    ll ans=0;
    while(ans<=qb&&dp[ans]<m) ans++;
    cout<<ans;
```

可行性多重背包

题目描述

设有1g、2g、3g、5g、10g、20g的砝码各若干枚（其总重≤1000）

输入方式：
a1,a2,a3,a4,a5,a6（表示1g砝码有a1个，2g砝码有a 2 个，...，20g砝码有a6个）

输出方式：
Total=N（N表示用这些砝码能称出的不同重量的个数，但不包括一个砝码也不用的情况）

> 未优化

```
#include <iostream>
#include<cstdio>
#include<string>
#include<vector>
#include<queue>
#include<cstring>
#include<cmath>
using namespace std;
const int maxn=1e3+10;
int dp[maxn];
int num[maxn];
int fama[]={0,1,2,3,5,10,20};
int main(){
  dp[0]=1;
  for(int i=1;i<=6;i++)
  {
    cin>>num[i];
    for(int j=1;j<=num[i];j++)
    {
      for(int k=1000;k>=fama[i];k--)
      {
        dp[k]=dp[k]|dp[k-fama[i]];
      }
    }
  }
  int ans=0;
  for(int i=1;i<=1000;i++){
    if(dp[i]) ans++;
  }
  cout<<"Total="<<ans;
}
```

> bitset加速版

```
int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
```

```
int popcount(unsigned int x) { // 返回x的二进制中1的个数
    int ret = 0;
    for(int i = 0; i < 8; i++) ret += table[x & 15], x >>= 4;
    return ret;
}
或
int count_one(int x)//x的二进制中第0到n-1位中1的个数
{
    int cnt = 0;
    for(int i = 0; i <= n - 1; ++i)   //n为最高位
        if(x & (1 << i))//注意做完按位与后的结果不是1，如可能为000010000，所以不要写成x & (1 <<
i) == 1
            cnt++;
    return cnt;
}
```

```
#include <iostream>
#include<cstdio>
#include<string>
#include<vector>
#include<queue>
#include<cstring>
#include<cmath>
#include<bitset>
using namespace std;
const int maxn=1e3+10;
bitset<maxn> dp;
int num[maxn];
int fama[]={0,1,2,3,5,10,20};
int main(){
  dp[0]=1;
  for(int i=1;i<=6;i++)
  {
    cin>>num[i];
    for(int j=1;j<=num[i];j++)
    {
        dp|=(dp<<fama[i]);
        //bitset对象左移表示为全部左移动，比如0011<<1为0110
    }
  }
  int ans=0;
  for(int i=1;i<=1000;i++){
    if(dp[i]) ans++;
  }
  cout<<"Total="<<ans;
}
```

> 二进制优化

```cpp
#include <iostream>
#include<cstdio>
#include<string>
#include<vector>
#include<queue>
#include<cstring>
#include<cmath>
using namespace std;
const int maxn=1e3+10;
int dp[maxn];
int num[maxn];
int fama[]={0,1,2,3,5,10,20};

int qpow(int n,int m)
{
  int res=1;
  int multi=n;
  while(m!=0)
    {
      if((m&1)==1)
        {
          res*=multi;
        }
      multi=multi*multi;
      m/=2;
    }
  return res;
}
int main(){
  dp[0]=1;
  for(int i=1;i<=6;i++)
  {
    cin>>num[i];
    int cnt=0;
    int sav=num[i];
    int binary[maxn];
    while(sav!=0)
    {
      binary[cnt++]=sav%2;
      sav/=2;
    }
    int newf[maxn];
    for(int j=0;j<cnt-1;j++)
      {
        newf[j]=fama[i]*qpow(2, j);
      }
    sav=num[i];
    for(int j=0;j<cnt-1;j++)
    {
```

```
      sav-=qpow(2,j);
    }
    newf[cnt-1]=fama[i]*sav;
    for(int k=0;k<cnt;k++)
      {
        for(int j=1000;j>=newf[k];j--)
          {
            dp[j]=dp[j]|dp[j-newf[k]];
          }
      }
  }
  int ans=0;
  for(int i=1;i<=1000;i++){
    if(dp[i]) ans++;
  }
  cout<<"Total="<<ans;
}
```

单调队列优化

```
#include <iostream>
#include<cstdio>
#include<string>
#include<vector>
#include<queue>
#include<cstring>
#include<cmath>
using namespace std;
const int maxn=1e3+10;
int dp[maxn];
int num[maxn];
int fama[]={0,1,2,3,5,10,20};

int qpow(int n,int m)
{
  int res=1;
  int multi=n;
  while(m!=0)
    {
      if((m&1)==1)
        {
          res*=multi;
        }
      multi=multi*multi;
      m/=2;
    }
  return res;
}
int main(){
```

```
  dp[0]=1;
  for(int i=1;i<=6;i++)
  {
    cin>>num[i];
    dp[0]=1;
    int q1[maxn],q2[maxn];
    for(int d=0;d<fama[i];d++)
    {
      int l=1,r=0,maxp=(1000-d)/fama[i];
      for(int P=0;P<=maxp;P++)
        {
          int &x=dp[d+fama[i]*P];
          while(l<=r && q1[l]<P-num[i]) l++;
          while(l<=r && x>=q2[r] ) r--;
          q1[++r]=P;
          q2[r]=x;
          x=x|q2[l];
        }
    }

  }
  int ans=0;
  for(int i=1;i<=1000;i++){
    if(dp[i]) ans++;
  }
  cout<<"Total="<<ans;
}
```

多维背包

> 一般的多维背包就是限制条件多了一条，比如背包装东西不仅有体积限制，又有重量限制，这种模板只需要加一层for循环额外便利重量就可以了。

例子：

```
for(int i=1;i<=n;i++)
{
  for(int j=M;j>=m[i];j--)
  {
    for(int k=T;k>=t[i];k--)
    {
      dp[j][k]=max(dp[j][k],dp[j-m[i]][k-t[i]]+1);
    }
  }
}
```

小变式

题目背景

小明过生日的时候，爸爸送给他一副乌龟棋当作礼物。

题目描述

乌龟棋的棋盘是一行N个格子，每个格子上一个分数（非负整数）。棋盘第1格是唯一的起点，第N格是终点，游戏要求玩家控制一个乌龟棋子从起点出发走到终点。

乌龟棋中M张爬行卡片，分成4种不同的类型（M张卡片中不一定包含所有4种类型的卡片，见样例），每种类型的卡片上分别标有1,2,3,4四个数字之一，表示使用这种卡片后，乌龟棋子将向前爬行相应的格子数。游戏中，玩家每次需要从所有的爬行卡片中选择一张之前没有使用过的爬行卡片，控制乌龟棋子前进相应的格子数，每张卡片只能使用一次。

游戏中，乌龟棋子自动获得起点格子的分数，并且在后续的爬行中每到达一个格子，就得到该格子相应的分数。玩家最终游戏得分就是乌龟棋子从起点到终点过程中到过的所有格子的分数总和。

很明显，用不同的爬行卡片使用顺序会使得最终游戏的得分不同，小明想要找到一种卡片使用顺序使得最终游戏得分最多。

现在，告诉你棋盘上每个格子的分数和所有的爬行卡片，你能告诉小明，他最多能得到多少分吗?

输入格式

每行中两个数之间用一个空格隔开。

第1行2个正整数N,M，分别表示棋盘格子数和爬行卡片数。

第2行N个非负整数，a1,a2,...,aN，其中ai表示棋盘第i个格子上的分数。

第3行M个整数，b1,b2,...,bM，表示M张爬行卡片上的数字。

输入数据保证到达终点时刚好用光M张爬行卡片。

输出格式

1个整数，表示小明最多能得到的分数。

```
#include<iostream>
#include<cstdio>
#include<fstream>
#include<algorithm>
#define ll long long
#define re register
using namespace std;
const int maxn=50;
inline int read()
{
    int x=0,f=1;char ch=getchar();
    while (!isdigit(ch)){if (ch=='-') f=-1;ch=getchar();}
        while (isdigit(ch)){x=x*10+ch-48;ch=getchar();}
        return x*f;
}
int dp[maxn][maxn][maxn][maxn];
int num[maxn];
int map[400];
int main()
{
  int n=read(),m=read();
  for(int i=0;i<n;i++)
  {
    map[i]=read();
  }
  for(int i=1;i<=m;i++)
  {
    int card=read();
    num[card]++;
  }
  dp[0][0][0][0]=map[0];
  for(int i=0;i<=num[1];i++)
  {
    for(int j=0;j<=num[2];j++)
    {
      for(int k=0;k<=num[3];k++)
      {
        for(int x=0;x<=num[4];x++)
        {
        int pos=i+2*j+3*k+4*x;
    if(i!=0) dp[i][j][k][x]=max(dp[i][j][k][x],dp[i-1][j][k][x]+map[pos]);
    if(j!=0) dp[i][j][k][x]=max(dp[i][j][k][x],dp[i][j-1][k][x]+map[pos]);
    if(k!=0) dp[i][j][k][x]=max(dp[i][j][k][x],dp[i][j][k-1][x]+map[pos]);
    if(x!=0) dp[i][j][k][x]=max(dp[i][j][k][x],dp[i][j][k][x-1]+map[pos]);
    \\其实就是分类讨论最后一张打的是什么牌
```

```
        }
      }
    }
  }
  cout<<dp[num[1]][num[2]][num[3]][num[4]];
}
```

> 多维背包中，维数一般表现为某几种属性的上限，或是某几样需要满足的需求。

混合背包

> 指一道题中需要使用多个背包的模型跑，将前面的各种背包灵活运用。

分组背包

所谓分组背包，就是将物品分组，每组的物品相互冲突，最多只能选一个物品放进去。

> 伪代码

```
for 所有的组k
    for v=V..0
        for 所有的i属于组k
          f[v]=max{f[v],f[v-w[i]]+c[i]}
```

> 注意这里的三层循环的顺序，“for v=V..0”这一层循环必须在“for 所有的i属于组k”之外。这样才能保证每一组内的物品最多只有一个会被添加到背包中。

有依赖的背包

> 本质为01背包的小变式，根据附件选择情况，分类跑dp就行了。

题目描述

金明今天很开心，家里购置的新房就要领钥匙了，新房里有一间金明自己专用的很宽敞的房间。更让他高兴的是，妈妈昨天对他说：“你的房间需要购买哪些物品，怎么布置，你说了算，只要不超过 n 元钱就行”。今天一早，金明就开始做预算了，他把想买的物品分为两类：主件与附件，附件是从属于某个主件的。
如果要买归类为附件的物品，必须先买该附件所属的主件。每个主件可以有 0 个、1 个或 2 个附件。每个附件对应一个主件，附件不再有从属于自己的附件。金明共有n元，有m个物品，下m行分别描述：使用w元，重要度为v，属于i号主件（如果为0，则为主件）。一个物品的价值为w*v，求能够购买达成的最大价值。

输出格式

输出一行一个整数表示答案。

思路

> 还记得01背包的决策是什么吗?

> 1.不选，然后去考虑下一个

> 2.选，背包容量减掉那个重量，总值加上那个价值。

> 这个题的决策是五个，分别是：

> 1.不选，然后去考虑下一个

> 2.选且只选这个主件

> 3.选这个主件，并且选附件1

> 4.选这个主件，并且选附件2

> 5.选这个主件，并且选附件1和附件2.

> 这五个情况全对dp取一次max就行了

```
#include<iostream>
#include<cstdio>
#include<vector>
#include<algorithm>
using namespace std;
const int maxn=3e4+10;
int dp[maxn];
vector<int> list[maxn];
inline int read()
{
  int x=0,f=1;char ch=getchar();
  while (!isdigit(ch)){if (ch=='-') f=-1;ch=getchar();}
  while (isdigit(ch)){x=x*10+ch-48;ch=getchar();}
  return x*f;
}
int v[maxn],w[maxn],affi[maxn];
int isnt_main[maxn];
int main(){
  int n=read(),m=read();
  n/=10;
  for(int i=1;i<=m;i++)
  {
    w[i]=read();
    w[i]/=10;
    v[i]=read();
    int add=read();
    if(add!=0)
    {
      isnt_main[i]=1;
      list[add].push_back(i);
    }
  }
  for(int i=1;i<=m;i++)
  {
    if(!isnt_main[i])
    for(int j=n;j>=w[i];j--)
    {
      int len=list[i].size();
      if(j>=w[i])
      dp[j]=max(dp[j],dp[j-w[i]]+v[i]*w[i]);
      if(len==1)
      {
        if(j>=w[i]+w[list[i][0]])
          dp[j]=max(dp[j],dp[j-w[i]-w[list[i][0]]]
          +v[i]*w[i]+v[list[i][0]]*w[list[i][0]]);
      }
      if(len==2)
```

```
      {
        if(j>=w[i]+w[list[i][0]])
          dp[j]=max(dp[j],dp[j-w[i]-w[list[i][0]]]
          +v[i]*w[i]+v[list[i][0]]*w[list[i][0]]);
        if(j>=w[i]+w[list[i][1]])
          dp[j]=max(dp[j],dp[j-w[i]-w[list[i][1]]]
          +v[i]*w[i]+v[list[i][1]]*w[list[i][1]]);
        if(j>=w[i]+w[list[i][1]]+w[list[i][0]])
          dp[j]=max(dp[j],dp[j-w[i]-w[list[i][1]]-w[list[i][0]]]
          +v[i]*w[i]+v[list[i][1]]*w[list[i][1]]
          +v[list[i][0]]*w[list[i][0]]);
      }
    }
  }
  cout<<dp[n]*10;
}
```

泛化物品的背包

这种背包，没有固定的费用和价值，它的价值是随着分配给它的费用而定。在背包容量为V的背包问题中，当分配给它的费用为vi时，能得到的价值就是h(vi)。这时，将固定的价值换成函数的引用即可。

树上分组背包

> 分组背包的伪代码如下

```
for (int i = 0到组数)
  for (int j = max_v到0)
    for (int k = 此组所有物品)
      f[j] = max (f[j], f[j - v[k]] + w[k])
```

简单来说就是把每个节点看成一个背包，它的容量就是以这个节点为根的子树大小，组数就是连接的儿子个数。

每组都有很多选择，选一个，两个，三个用户，把这些选择当做组中的元素就好了，容易看出每组中只能选一个元素，比如你选择了选一个用户，就不可能同时选择选两个用户。

> 树上分组背包dfs模板

```
#include<iostream>
#include<algorithm>
#include<cstring>
#include<fstream>
#include<vector>
using namespace std;
const int maxn=3e3+10;
int dp[maxn][maxn];
int v[maxn];
vector<pair<int,int> > G[maxn];
int n,m;
inline int read()
{
  int x=0,f=1;char ch=getchar();
  while (!isdigit(ch)){if (ch=='-') f=-1;ch=getchar();}
  while (isdigit(ch)){x=x*10+ch-48;ch=getchar();}
  return x*f;
}
int dfs(int st)
{
  if(st>n-m)
  {
    dp[st][1]=v[st];
    return 1;
  }
  int sum=1;
  int len=G[st].size();
  for(int i=0;i<len;i++)  //遍历子树，相当于遍历组
  {
    int to=G[st][i].first;
    int cost=G[st][i].second;
    int num=dfs(to);
```

```
    sum+=num;    //把节点看成背包，sum为该节点为根的树的节点总数，为背包容量上限
    for(int j=sum;j>=0;j--)
    {
      for(int t=0;t<=num;t++) //遍历每一种组中选取的每一种情况
      {
        if(j>=t)
          dp[st][j]=max(dp[st][j],dp[st][j-t]+dp[to][t]-cost);
      }
    }
  }
  return sum;
}
int main(){
  n=read(),m=read();
  for(int i=0;i<=n;i++)
    for(int j=0;j<=n;j++)
      dp[i][j]=-9999999;
  int zhuan=n-m;
  for(int i=1;i<=zhuan;i++)
  {
    int size=read();
    for(int j=1;j<=size;j++)
      {
        int to=read(),cost=read();
        G[i].push_back(make_pair(to,cost));
      }
  }
  for(int i=zhuan+1;i<=n;i++)
    v[i]=read();
  for(int i=0;i<=n;i++)
  {
    dp[i][0]=0;
  }
  dfs(1);
  for(int i=n;i>=0;i--)
  {
    if(dp[1][i]>=0)
    {
      cout<<i;
      return 0;
    }
  }
}
```

一些神奇的背包

砝码称重

现有n个砝码，重量分别为 ai，在去掉 m 个砝码后，问最多能称量出多少不同的重量（不包括 0）。

请注意，砝码只能放在其中一边。

输入格式

第 1 行为有两个整数 n 和m，用空格分隔。（n<=20,m<=4）

第 2 行有 n 个正整数 a1 ,a2 a3,...,an，表示每个砝码的重量。

输出格式

仅包括 1 个整数，为最多能称量出的重量数量。

> 第一种，由于n<=20，因此可以dfs枚举所有情况，对符合条件的进行dp操作

```
#include<cstdio>
#include<cstring>
#include<algorithm>
using namespace std;
const int maxn=22;
const int maxm=2010;
int n,m,a[maxn],ans,tot,ret;
bool tf[maxn],f[maxm];
void dp()//不传参，全部定义在全局变量中，就是个01背包
{
    memset(f,0,sizeof f);f[0]=true;ans=0;tot=0;//清零，因为可能要调用多次
    for(int i=0;i<n;i++)//从前到后选取所有的砝码
    {
        if(tf[i])continue;//如果被标记为已经舍弃就跳过
        for(int j=tot;j>=0;j--)if(f[j]&&!f[j+a[i]])f[j+a[i]]=true,ans++;
        //否则dp并且维护ans的值
        tot+=a[i];//这个tot意为当前f[i]为真值的最大的i，极大加快了dp过程
    }
    ret=max(ans,ret);//更新最后的答案
}
void dfs(int cur,int now)
//cur代表当前已经选取/放弃了多少个砝码，now代表已经放弃了多少个砝码
{
    if(now>m)return;//如果已经放弃的砝码数超过了需要放弃的砝码数，剪枝
    if(cur==n){if(now==m)dp();return;}//如果搜索完后正好符合条件，执行一次dp过程
    dfs(cur+1,now);//不放弃当前的砝码，继续向下
    tf[cur]=true;//留下足迹
    dfs(cur+1,now+1);//放弃当前砝码
    tf[cur]=false; //擦除足迹
}
int main()
```

```
{
    scanf("%d%d",&n,&m);
    for(int i=0;i<n;i++)scanf("%d",a+i);
    dfs(0,0);
    printf("%d\n",ret);
    return 0;
}
```

> bitset版状压dp

```
#include <bitset>
#include <cstdio>
int w[25];
int table[16] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4};
int popcount(unsigned int x) { // 返回x的二进制中1的个数
    int ret = 0;
    for(int i = 0; i < 8; i++) ret += table[x & 15], x >>= 4;
    return ret;
}
int main() {
    int n, m, ans = 0;
    scanf("%d%d", &n, &m);
    for(int i = 0; i < n; i++) scanf("%d", w + i);
    for(int i = 0, li = 1 << n; i < li; i++) {
        if(popcount(i) == n - m) {  //符合条件
            std::bitset<2010> S;
            S[0] = 1;
            for(int j = 0; j < n; j++) if(i & (1 << j)) S |= S << w[j];
              //i & (1<<j)为判断对于当前情况i，是否有选第j个背包，如果选了就dp一下
            int siz = S.count();
            ans = ans > siz ? ans : siz;
        }
    }
    printf("%d\n", ans - 1);
    return 0;
}
```