状态压缩动规例题选讲

Helang

例1：传球游戏 NKOJ1752

例2：校长的烦恼 NKOJ3700

例3：玉米地 NKOJ1902

例4：炮兵阵地 NKOJ1184

例5：广场铺砖 NKOJ1466

例6：硬木地板 NKOJ2310

例7：商品促销 NKOJ1903

例1：传球游戏 NKOJ1752

例1：传球游戏 NKOJ1752

n（2<=n<=16）个人在做传球的游戏,编号为1-n。

游戏规则是这样的：开始时球可以在任意一人手上，他可把球传递给其他人中的任意一位；下一个人可以传递给未接过球的任意一人,球不能自己传给自己。

即球只能经过同一个人一次，而且每次传递过程都有一个代价；不同的人传给不同的人的代价值可能不同；

求当球经过所有n个人后，整个过程的最小总代价是多少。

例1：传球游戏 问题分析

n个人，每个人只有两种状态，即传经了与未传经，因此可用一个二进制数表示已经传球的人的集合，若当前i的第k位为0则第 k人未传经，1则已传经。比如i=25,其对应的二进制数为(11001),表示已经传经1、4、5号人。

本题数据规模较小，考虑集合类型的动规，设定状态：

f[i][j]：表示当前球已传经集合i中的所有人，球现在第j号人手中，所需的最小总代价。

转移方程：

f[i|(1<<(k-1)][k]=min{ f[i|1<<(k-1)][k] , f[i][j]+cost[j][k] }

将球从j传给k，k为之前没有传过球的人

$0<=i<2^n-1$ $1<=j,k<=n$

初始：f[1<<(j-1)][j]=0 (1<=i<=n)表示一开始球在第j号人手上（j已经传球），其余f[ ][ ]为无穷大。

答案：min{f[(1<<n)-1][j]} ,(1<=j<=n), (1<<n)-1的作用是将n个二进制位全部置为1，表示n个人全都传经了。j表示最后球在j手中。

例1：传球游戏 参考代码

```
t=(1<<n)-1;          //目标集合，对应二进制数全为1
for(i=0;i<=t;i++)
  for(j=1;j<=n;j++)f[i][j]=inf;
 for(i=1;i<=n;i++) f[1<<(i-1)][i]=0;  //初始化，一开始球在i号人手上


 //dp
 for(i=0;i<=t;i++)          //枚举集合
   for(j=1;j<=n;j++)        //当前求在第j号人手上
     for(k=1;k<=n;k++)    //枚举球将传到的下一个人的编号
         if((i>>(k-1)&1)==0)    //i的第k位为0，表示第k个人没有传经
         {
              v=1<<(k-1);
              if (f[i][j]+cost[j][k]<f[i|v][k])   //i|v作用是把第k个人的状态变为1
                   f[i|v][k]=f[i][j]+cost[j][k];
         }
```

例1：传球游戏 方法2 搜索

经典深搜：

搜索程序**dfs(int Sum,int Left,int Now)**

Sum表示之前传过球的人的代价总和。

Left表示还剩多少个人没有讨论

Now表示当前讨论的人的编号

怎么优化? 我们用全局变量Ans记录搜索出的最佳答案

可行性剪枝：当Sum>=Ans，则return;

表示当前得到的总和已经超过了之前求出的最优值Ans,那么没有必要再讨论下去了。

最优性剪枝：near[i]到达i号点(第i个人)的所有边中最短的那条的长度。

BestDis表示最理想的传递路线的总长度。最理想的传递路线就是传到每个人手中都是走的最短路线(near[i])

当球传经了第i个人，那么BestDis=BestDis-near[i]; 表示剩下的人传递的最理想路线。

当Sum+BestDis>=Ans，则return;

表示当剩下的未讨论过的人都以最小代价来传递都无法得到更优的结果，那就没必要讨论下去了。

例1：传球游戏 搜素 参考代码1

```
main() ……
  Ans=inf;
    for(i=1;i<=n;i++)
        for(j=1;j<=n;j++)
        {
            scanf("%d",&cost[i][j]);          //near[ ]记录到达i号点的最短的边的长度
            if(cost[i][j]>=0&&cost[i][j]<near[j])near[j]=cost[i][j];
        }
    for(i=1;i<=n;i++)BestDis+=near[i]; //BestDis用于剪枝，表示所有的点都走到它的最短边。
    for(i=1;i<=n;i++)//枚举从每个点出发，开始传球
    {
         BestDis-=near[i];
         dfs(0,n-1,i);
         BestDis+=near[i];
    }
    printf("%d\n",Ans);
```

例1：传球游戏 搜素 参考代码2

```
void dfs(int Sum,int Left,int Now)//Sum记录之前传过球的人的代价总和
{
    if(Left==0)          //剩余人数为0，表示找到一种方案
    {
        if(Sum<Ans)Ans=Sum;
        return;
    }
    if(Sum>=Ans)return;                  //可行性剪枝
    if(Sum+BestDis>=Ans)return;    //最优性剪枝
    visit[Now]=true;
    for(int i=1;i<=n;i++)
        if( i!=Now && visit[i]==false )
        {
            BestDis-=near[i];
            dfs(Sum+cost[Now][i],Left-1,i);   //把球从Now传给i
            BestDis+=near[i];
        }
    visit[Now]=false;
}
```

例2：校长的烦恼 NKOJ3700

例2：校长的烦恼 NKOJ3700

某中学开设有s门课程，现有教师m个。今天有n个求职者来应聘新教师。

已知每个人工资和能教授的课程。在职教师不能辞退，校长想知道，最少支付多少工资就能使得每门课都有至少两名教师能教。

1<=s<=8

1<=m<=20

1<=n<=100

10000<=每个人的工资<=50000

1<=每个人教的课的编号<=s

例2：校长的烦恼 问题分析

本题课程数量规模较小，考虑集合类型的动规，设定状态：

f[i][j]：表示达到i和j状态所需最小工资。

其中i表示课程集合(二进制),i集合中的课程都至少有一个教师能教。

若i==10011表示第0,1,4门课至少有一个老师能教

其中j表示课程集合,j集合中的课程都至少有两个教师能教。

决策：对于每个求职者，要么选，要么不选，就是01背包问题。

p[k]表示第k号求职者能上的课程集合。 若k号人能上0,3,4这三门课，则p[k]=11001

对于当前的i和j，可以根据当前枚举到的求职者课程即可，可推出下一个状态：

S1 = p[k] | i //选了第k个人后，更新至少有一个老师教的课程集合

S2 = (p[k] & i) | j //选了第k个人后，更新至少有两个老师教的课程集合

转移方程：

f[S1][S2] = min(f[S1][S2], f [i][j] + p[k])

例2：校长的烦恼 参考代码1

```
//输入和初始化//////////////////////////////////////////
maxState = (1<<courseNum) - 1;
for(i=1; i<=m+n; i++)
{
        字符串读入转成数字num……
        p[i]=p[i]|(1<<(num-1));        //p[i]表示i能教的课程集合
        if(i<=m)Cnt[num-1]++;          //记录下第num-1门课能上的人数
}
if(i <= m)
{
      Sum+=Cost[i];                    //记录在职教师的总工资
      S1=S1|p[i];                      //S1集合表示： 在职教师能教的课中，至少有1个人能上的课
}

for(i=0;i<courseNum;i++)
   if(Cnt[i]>1) S2=S2|(1<<i);    //S2集合表示：在职教师能教的课中，至少有两个人能上的课
```

例2：校长的烦恼 参考代码2

```
//DP//////////////////////////////////////////////////////
  设置f[ ][ ]数组为无穷大
  f[S1][S2]=Sum;                          //Sum为在职教师总工资
  for(k=m+1;k<=n+m;k++)                   //依次枚举讨论每个求职者
     for(i=maxState;i>=0;i--)              //反向for循环，使得每个物品只能使用一次。
        for(j=maxState;j>=0;j--)
        {
            if(f[i][j]>=INF) continue;
            S1=(p[k]|i);
            S2=(p[k]&i)|j;
            f[S1][S2]=min(f[S1][S2],f[i][j]+Cost[k]);
         }
  printf("%d\n",f[maxState][maxState]);
```

例3：玉米地 NKOJ 1902

例3：玉米地 nkoj 1902

Farmer John买了一片土地，可以表示为一片由方块组成的网格，长度为M，宽度为N（1 <= M, N <= 12）。现在FJ要在地里种上一些玉米让他的奶牛们直接在地里食用。FJ知道他的这片土地有些地方很贫瘠，没有办法种玉米；并且奶牛们不喜欢挨在一起吃玉米，所以不能在相邻的两块地上种玉米。请帮助FJ计算一下所有可能的种玉米的方案数。注意，结果输出对于100,000,000的余数，一棵玉米也不中也算是一种方案。

输入格式：

第一行 两个整数M和N

接下来是一个M*N的矩阵，1表示该方块可以种玉米，0表示不行

输出格式：

一个整数，表示对应的结果

样例输入：

2 3

1 1 1

0 1 0

样例输出：

9

样例说明：

把每块能中玉米的地编上号：

1 2 3

0 4 0

只种一块地，方案有(1, 2, 3, 4)；种两块地方案有 (13, 14, 34)；种三块地 (134), 一块也不种也算一种方案，总共4+3+1+1=9

分析1：

简单的说就是在0和1构成的矩阵上放置牛，1可以放，0不可放，牛不能相邻，问总方案数。
此题若采用搜索，状态数会达到2^(m*n)，显然是不可行的。 怎么办？

此题由数据0,1构成且每行最大为12，并且当前行只受上一行的影响。
那么考虑用**二进制**数来表示每行的放置牛的状态。

以样例为例：

2 3
1 1 1
0 1 0

讨论记录土地的情况：

把每行最初的情况看成二进制形式存储到Land[i]中（低位在右边）
Land[1]=二进制(111)=7 Land[2]=二进制(010)=2

下面讨论记录放置牛的情况：

1.把每一行的放置牛的情况看成一个二进制数作为一个状态，如果这一行的某一列放置了一头牛，对应的二进制位就是1，如果没有放置，那么这个二进制位就是0。

2.如果一种放置情况中出现了两个以上连续的1，那是不可行的（牛不能相邻），排除掉。

3.若一种放置情况在1的位置对应Land[ ]中的数为0，表示将牛放在了不能放牛的位置，也不可行，排除掉。

分析2：

动规状态：f[i][j]表示从1到第i行，且第i行状态为j时最多有多少中可能的放置方案。

样例：
2 3
1 1 1
0 1 0

讨论第1行：

在样例数据中，对于第一行中三位有0到7这八种状态(2^3),但是并不是每种状态都合法，
如3（011），6（110），7（111）都是不合法的(有相邻的1)，直接去掉。
所以对于第一行来说有5中情况合法：000，100，010，001，101
于是有f[1][0]=1(f[1][000]) f[1][1]=1(f[1][001]) f[1][2]=1(f[1][010]) f[1][3]=0(f[1][011])
f[1][4]=1(f[1][100]) f[1][5]=1(f[1][101]) f[1][6]=0(f[1][110]) f[1][7]=0(f[1][111])

讨论第2行：

f[2][0]和上一行的所有都不会发生冲突，因为是0，所以f[2][0]的值为上一行的所有f[1][i]的和=5;
f[2][1],1=(001)将牛放在第1个位置，而Land[2]=010对应位置为0，表示不能放牛，冲突，故f[2][1]=0
f[2][2],2=(010)和上一行的000,100,101,001 四个状态不发生冲突，故f[2][2]=f[1][0]+f[1][1]+f[1][4]+f[1][5]=4
f[2][3],3=(011)与Land[2]冲突且有连续1,f[2][3]=0
f[2][4],4=(100)与Land[2]冲突 f[2][4]=0
f[2][5],5=(101)与Land[2]冲突 f[2][5]=0
f[2][6],6=(110)与Land[2]冲突且有连续1,f[2][6]=0
f[2][7],7=(111)与Land[2]冲突且有连续1,f[2][7]=0
所以：到第2行总方案数=f[2][0]+f[2][1]+..+f[2][7]=9

动规方程：

$$f[i][j]=\begin{cases}\text{Sum}(f[i-1][k]) & j\text{与}k\text{不冲突 且}j\text{与Land}[i]\text{不冲突 且}j\text{中无连续1}\\ 0 & j\text{出现冲突}\end{cases}$$

分析3：

判断冲突：

判断一种状态j是否与前一行的状态k冲突：　**(j & k) ==0 表示不冲突**

判断一种状态j是否与Land[i]冲突：　**(j &(~Land[i])) ==0 表示不冲突**
(j | Land[i])==Land[i]

判断是否有连续的“1”：

(((j >> 1) & j==0)&&((j <<1) & j ==0))==true　表示j中没有连续1

动规方程：

$$f[i][j]=\begin{cases} \text{Sum}(f[i-1][k]) & (j\&k)==0 \text{ 并且 } (j\ \&(\sim Land[i]))==0 \\ & \text{并且 } (((j >> 1)\ \&\ j==0)\&\&((j <<1)\ \&\ j ==0))==true \\ 0 & j\text{出现冲突} \end{cases}$$

$0<=k,j<=Max=1+2+4...+2^{n-1}$
$1<=i<=m$

代码实现：

样例：
2 3
1 1 1
0 1 0

```
//输入：
 scanf("%d%d",&m,&n);
 for(i=1;i<=m;i++)
   for(j=1;j<=n;j++)
   {
      scanf("%d",&x);
      Land[i]=Land[i]+x*(1<<(n-j)); //作用是？
   }
```

把输入的数据当成二进制，
转换成十进制存储

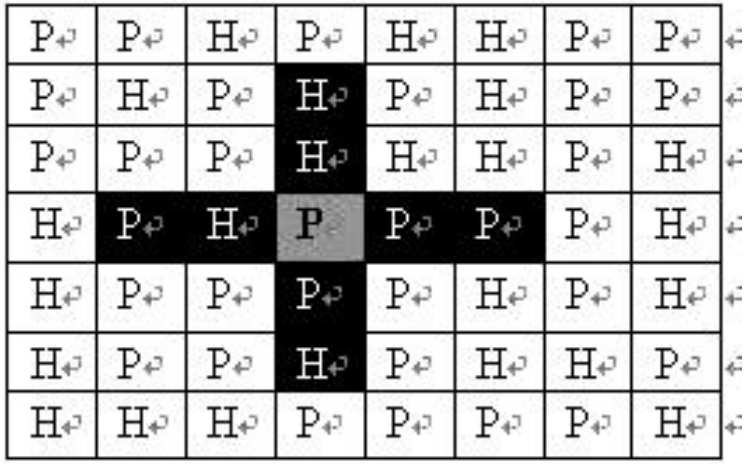
```
bool ok(int jj,int ii)   //判断在第ii行按jj的情况放置牛是否可行，可行返回true
{
  if (((jj & (~Land[ii]))==0)&&(((jj >> 1)&jj)==0)&&(((jj << 1)& jj)==0))return true;
  return false;
}
```

具体说就是判断jj状态与所在行ii的土地情况有无冲突
同时判断jj中是否包含连续的1

例4：炮兵阵地 NKOJ1184

例4：炮兵阵地 NKOJ1184

将军们打算在N*M(N≤100；M≤10)的网格地图上部署他们的炮兵部队。地图的每一格可能是山地（用“H” 表示），也可能是平原（用“P”表示），在每一格平原地形上最多可以布置一支炮兵部队（山地上不能够部署）； 一支炮兵部队在地图上的攻击范围如图中黑色区域所示：



在地图中的灰色所标识的平原上部署一支炮兵部队，则图中的黑色的网格表示它能够攻击到的区域：沿横向左右各两格，沿纵向上下各两格。从图上可见炮兵的攻击范围不受地形的影响。

现在，将军们规划如何部署炮兵部队，在防止误伤的前提下（任何一支炮兵部队都不在其他支炮兵部队的攻击范围内），在整个地图区域内最多能够摆放多少我军的炮兵部队。

分析：

采用类似前例的方法，我们可以得到状态：f[i][x][y]

表示当前讨论到第i行，其中x（对应的二进制）表示第i行的炮兵摆放状态，y（对应的二进制）表示第i-1行的摆放状态

状态转移的方程：

f[i][x][y]=max{ f[i-1][y][z] }+Cnt[x] //Cnt[x]为x状态中的炮兵的个数

要求x，y, z表示的状态互不冲突

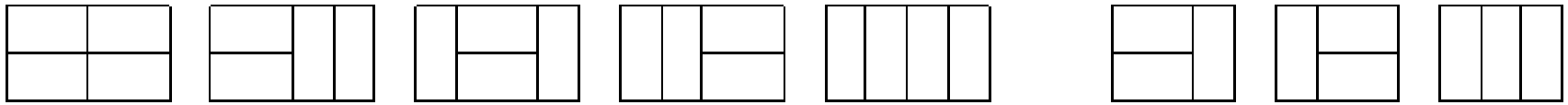
找出最后两行状态的f的最大值，这个值即为所求结果。

注意：f[ ][ ][ ]只跟前两行的状态发生关系，可用滚动数组来优化空间。

例5：广场铺砖问题 NKOJ1466

例5：广场铺砖问题 nkoj1466

给出一个W行H列的广场，用1*2小砖铺盖，小砖之间互相不能重叠，问有多少种不同的铺法？（1<=W，H<=11）



图例是在不同大小棋盘上的一些铺放方案
左边4*2的棋盘有5种方案，右边3*2的棋盘有3种方案

分析1

性质1：如果w和h都是奇数，则无解，否则有解。

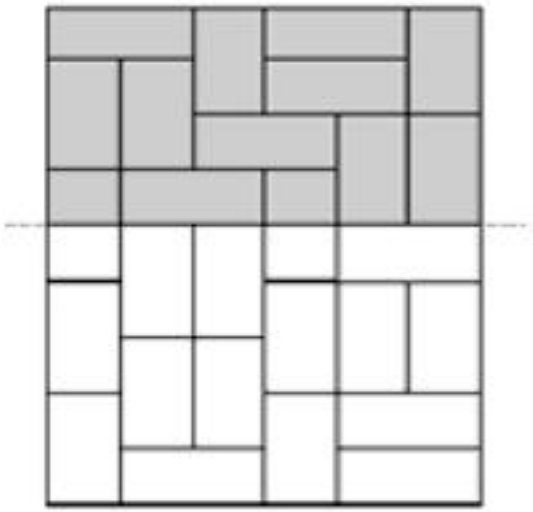
证明：w,h都是奇数，则w*h也是奇数，由于1×2的砖有2块，因此无论铺多少块都是偶数，因此不能覆盖所有的地板。如果地板的面积S是偶数，肯定能被2整除，因此可以用S/2块砖铺满整个地板。

性质2：对于每铺一次地板，只会影响所铺的上下两行。

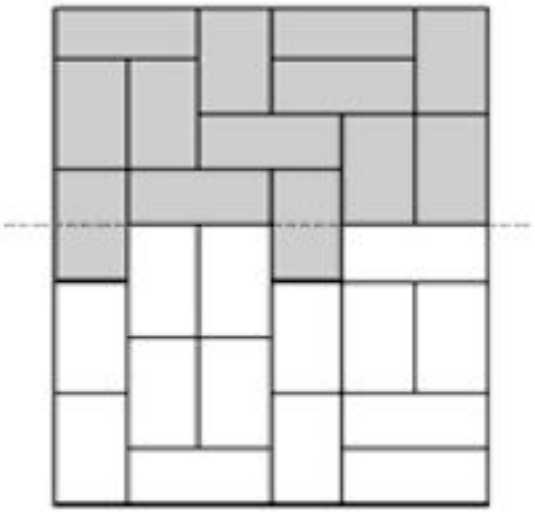
性质3：如果按行铺地板，每一行的铺法都类似。

分析2：

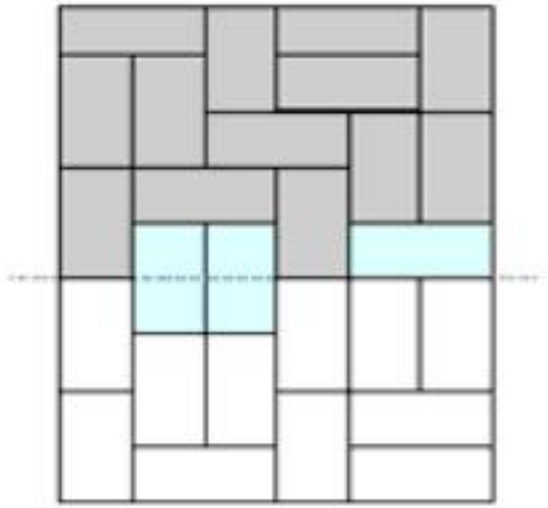
一个6×9的面积铺法如下图:



图一



图二



图三

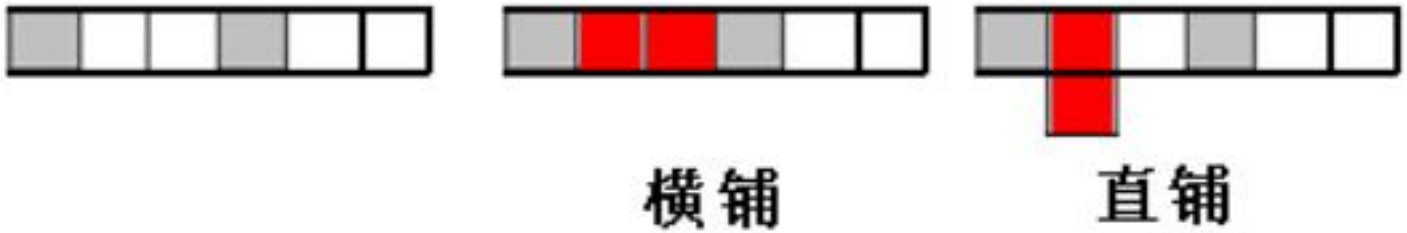
可以看出，在按行铺的过程中，某些砖会分成两半，如图2，但最多也是向下突出一格，在图3中，我们将图2的空隙填满，则又转移到了下一种状态。

状态的表示

如果我们把行进行动态规划，则第i行的各种情况即表示第i个阶段的的各种状态。

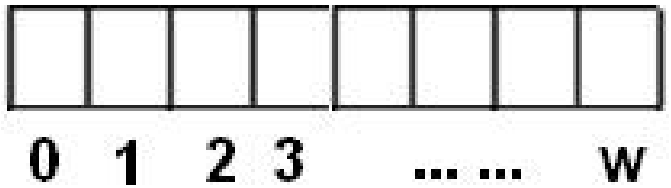
若某格被铺了砖，则用1表示，没有铺砖，则用0表示，那么行的状态就是一个w个格子的0,1串，即w位的二进制数。如下图状态为：100100

下面就是由某个二进制数能转化到另一个二进制数的问题了。如下图,状态由100100 → 111100 和110100 ：



动态规划

显然，对于每一行,宽度为w,每格可放0和1，因此有2^w 种状态，如下图：



设状态f(i,s)表示铺到第i行，前i-1行已铺满,第i行状态为s的方案数，则

$$f(i,s)=\sum f(i-1,s'),\text{其中}s'\text{的状态能变到}s$$

初始值f(1,0)=1

Ans=f[h+1][0]

时间复杂度为$O(h*2^w)$

位运算操作

若当前状态为s,对s 有下列操作：

1.判断第i位是否为0 ： **(S & 1 << i) == 0**，意思是将1左移i位与s进行与运算后，看结果是否为0.

2.将第i位置1 ： **S | 1 << i**，意思是将1左移动i位与s进行或运算.

3.将第i位置0 ： **S & ~(1 << i)**，意思是s与第i位为0，其余位为1的数进行与运算。

例如：

s=1010101，i=5

(S & 1 << i): 1010101 & 0100000 = 0

S | 1 << i: 1010101 | 0100000 = 1110101

s=1110101，i=5

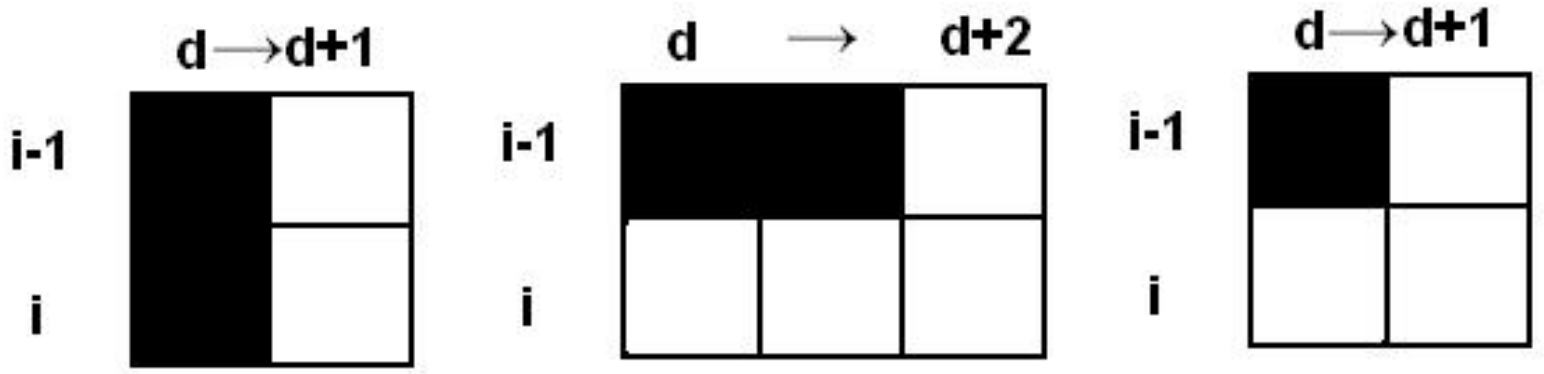
S & ~(1 << i): 1110101 & 1011111 = 1010101

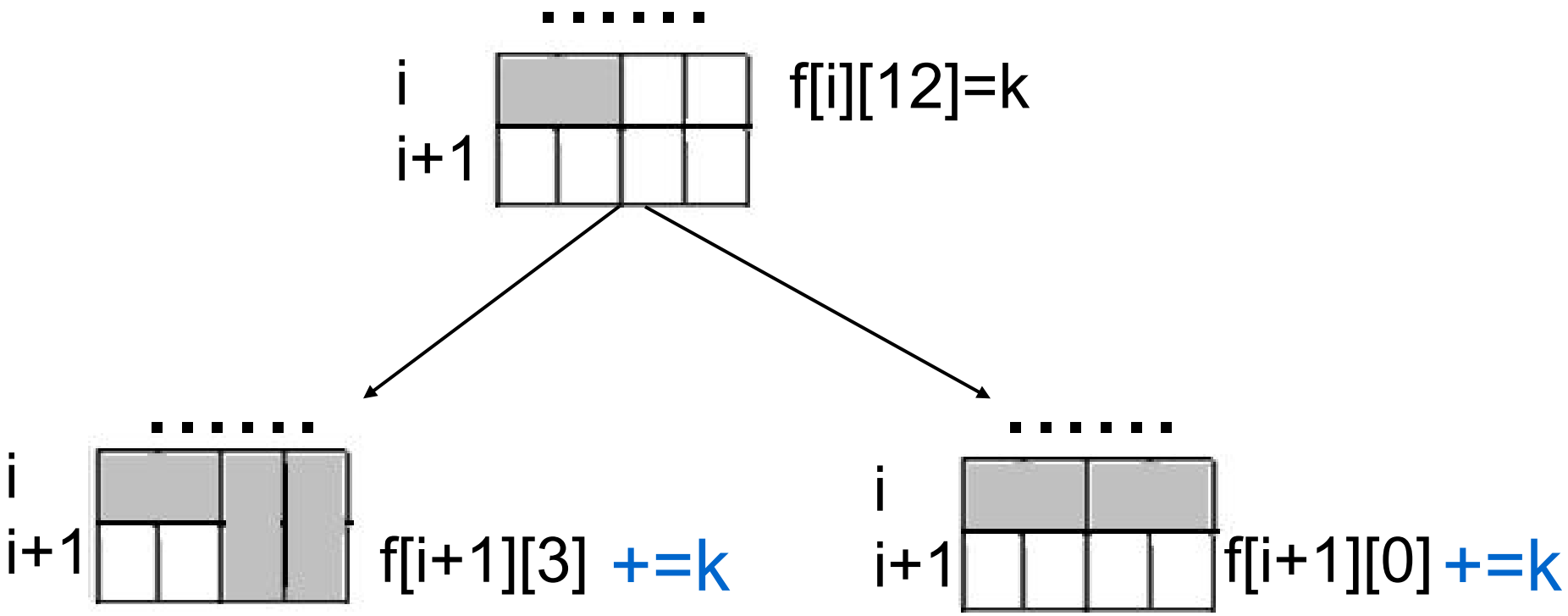
放置操作

对于每一行有w个位置，所以每一行都有0～2^w-1种状态。

对于当前行的状态s，它可推出下一行的状态s'，显然，对于该行某个位置j：

1. 如果当前该位置为0,那么该位置可以竖放 ，s'对应位置为 1
2. 如果当前连续两个位置为0,那么这两个连续位置可以横放 即s'对应位置为00
3. 如果当前行该位置为1,显然该位置不能再放,于是应该把s'该位置设置为0




i
i+1
f[i][12]=k
i
i+1
f[i+1][3] +=k
i
i+1
f[i+1][0] +=k

伪代码：

```
//当前讨论第i行的第d位，第i行初始状态为s,当前状态为s1,i+1状态为s2
void dfs(int i, int s,int s1, int s2, int d)
{
    ......
    if (s1 == MAX)   //如第i行已经铺完，则累加 Max=2^h-1
        f[i + 1][s2] += f[i][s];
    else
        if ((s1 & (1 << d)) == 0) //第d位为0
        {
            s1=s1  | 1 << d;
            dfs(i,s,s1, s2| (1 << d), d + 1); //竖放一块，将第d位变为1，并右移1位
             //如果第d+1位也为0，表示可以横放一块，则直接搜索d+2位
            if (d < h-1 && (s1 & (1 << (d + 1))) == 0)
            {
                s1=s1|1<<(d+1);
                dfs(i,s,s1, s2, d + 2);
            }
        }
        else //将s1的第d位不能放，把s2的第d位变为0,并右移1位
            dfs(i,s,s1,s2 & ~(1 << d), d + 1);
    ......
}
```

分析

按上述分析可以比较容易地写出dfs过程。

在dfs的边界，我们要同时记录当前行状态s1，和前一行状态s2。s1是由s2转移过来的。可能会有多个s2转移到同一个s1,那么s1的方案数就应该等于各个s2方案数之和。明白了这一点，逐行递推就比较容易了。

在dfs时，把所有s1,s2状态对保存在数组中。然后再逐行递推。本题的总的状态数不算多，可以采用这种方法。

也可以不保存状态，而是边递推边dfs。

例6：硬木地板 NKOJ2310

例6：硬木地板 nkoj2310

- 有一M×N 的矩形地板
- 可以使用的地板砖形状有两种：

  1) 2×1的矩形砖

  2) 2×2中去掉一个1×1的角形砖
- 你需要计算用这些砖铺满地板共有多少种不同的方案。
- 必须盖满，地板砖数量足够多，不能存在同时被多个板砖覆盖的部分。
- 1<=M<=9, 1<=N<=9

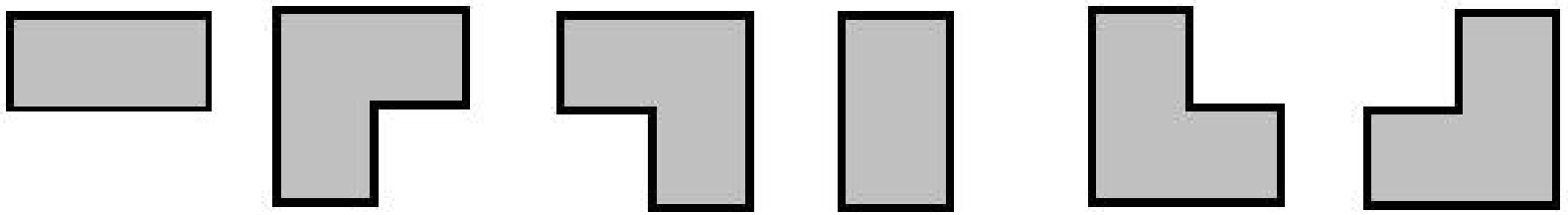
分析

2 × 3 的地板所有铺法共 5 种，如下图。

可以看出这题与动归 4 的“广场铺砖问题”完全类似。只不过这里多了一种砖而已。

分析

- 将矩阵的１行看成一种状态，则某一行矩阵的铺砖情况可以用一个０１串表示：０表示未铺砖，１表示已铺了砖。
- 该题有两种类型的砖，因此对应６种铺法：



（1）　（2）　（3）　（4）　（5）　（6）

- 对于上下两行：要能用某种类型的砖铺，必须该砖所覆盖的区域为空。

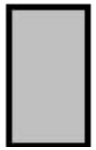
```
#define bin(i) (1 << i)              /*1左移i位*/
#define emp(a,i) (!(a & bin(i)))  /*判断a的i位是否位0*/
```

下图a,b表示相邻两行的状态，b为a的下一行的状态

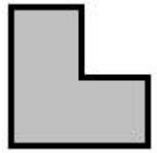
a= a | bin(i) | bin(i + 1); b=b

a =a | bin(i) | bin(i + 1); b= b | bin(i)

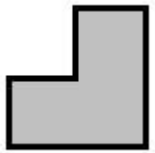
a =a | bin(i) | bin(i + 1); b= b | bin(i+1)



a= a | bin(i) ; b= b | bin(i)



a= a | bin(i) ; b= b | bin(i) | bin(i - 1)



a= a | bin(i) ; b= b | bin(i) | bin(i + 1)

动态规划

- 设f(i,s)表示第i行的状态为s时可达的方案数，则：

$$f(i,s)=\sum f(i-1,s'),\text{其中}s'\text{的状态能变到}s$$

- 初始值f(1,0)=1
- Ans=f[h+1][0]
- 时间复杂度为O(h*2^w)

状态压缩的动态规划：

从状态压缩的特点来看，这个算法适用的题目符合以下的条件：

1.解法需要保存一定的状态数据（表示一种状态的一个数据值），每个状态数据通常情况下是可以通过2进制来表示的。这就要求状态数据的每个单元只有两种状态，比如说棋盘上的格子，放棋子或者不放，或者是硬币的正反两面。这样用0或者1来表示状态数据的每个单元，而整个状态数据就是一个一串0和1组成的二进制数。

2.解法需要将状态数据实现为一个基本数据类型，比如int，long等等，即所谓的状态压缩。状态压缩的目的一方面是缩小了数据存储的空间，另一方面是在状态对比和状态整体处理时能够提高效率。这样就要求状态数据中的单元个数不能太大，比如用int来表示一个状态的时候，状态的单元个数不能超过32。

例7 商品促销 NKOJ1903

例7 商品促销 nkoj1903

在一个商店里每种商品都有一个价格。比如：一束鲜花的价格是2元，一个花瓶的价格是5元。为了吸引更多的顾客，该商店推出了一些优惠促销活动。 一种优惠是一次购买多个商品可以获得减价。比如，3束鲜花只要5元而不是6元，两个花瓶和一束鲜花只要10元而不是12元。

请你写一个程序帮助顾客计算享受优惠后，他需要支付的费用，顾客们总是选择最佳的优惠，所以，应该使费用尽可能的低。但是你不能私自给顾客增加商品，尽管有时这么做了可以使顾客支付的费用更低。 比如根据上述的优惠活动，最低的价格购买3束花和2个花瓶的价格是14元：两个花瓶和一束鲜花享受优惠价格10元。另外两束鲜花采用普通价格购买花费4元，总费用14元。

输入格式：

第一行，一个整数b，表示需要购买b种不同的商品 (0 <= b <= 5)
接下来b行，每行3个整数c,k和p，c(1 <= c <= 999)表示该种商品的编号，k(1 <= k <= 5)需要购买该商品的数量，p (1 <= p <= 999)表示该商品的价格。
接下来一行，一个整数s，表示有s(0 <= s <= 99)种优惠活动
接下来s行，每行描述一种优惠活动的详细情况：第一个整数n(1 <= n <= 5)表示，该活动中商品的种数。接下来有n对数字（c,t），表示编号为c的商品需要购买t(1 <= t <= 5)个才能享受优惠。最后一个数字v(1 <= v <= 9999) ,表示享受该优惠活动需要支付的费用。

输出格式：

一个整数，表示最小要支付的费用

输入样例：

```
2
7 3 2
8 2 5
2
1 7 3 5
2 7 1 8 2 10
```

输出样例：

```
14
```

分析1：

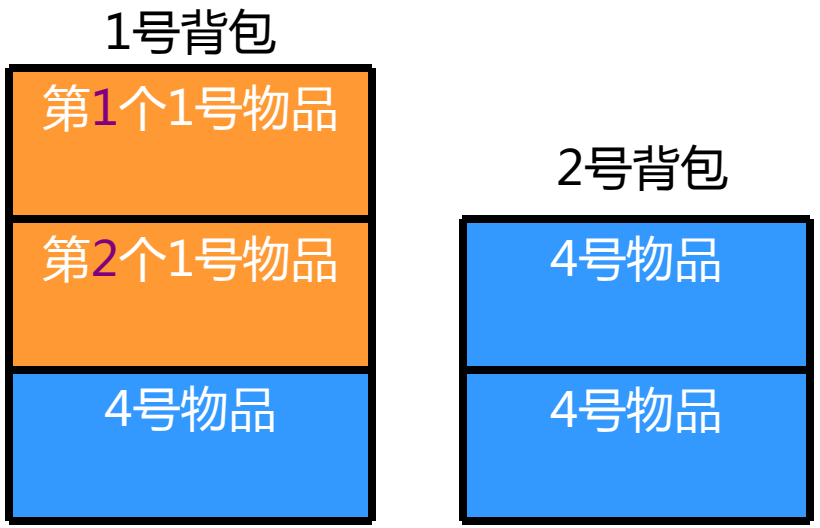
输入样例：
2
7 3 2
8 2 5
2
1 7 3 5
2 7 1 8 2 10

看起来像一个背包问题！

以样例数据为例，可看成有两个背包：
1号背包容积为3，且只能装7号物品
2号背包容积为2，且只能装8号物品

有四种物品：
1号.单独的一个7号物品，在1号背包中占用1单位体积，价格为2
2号.单独的一个8号物品，在2号背包中占用1单位体积，价格为5
3号.由3个7号物品构成的物品，在1号背包中占用3个单位体积，价格为5
4号.由1个7号物品和2个8号物品构成的物品，在1号背包中占用1个单位体积，在2号背包中占用2个单位体积，价格为10

最终求解填满所有背包所需最小费用！



分析2：

最多5种物品，每种物品最多买5个。

可用一个**五位的六进制数**来表示物品购买的情况！

输入样例：

```
2
7 3 2
8 2 5
2
1 7 3 5
2 7 1 8 2 10
```

以样例数据为例，先对物品**重新编号**：
7号物品编号为0， 8号物品编号为1
目标状态(背包)可表示为六进制数$(00023)_6$ == 十进制数$(15)_{10}$

有四种物品：
0号.单独的一个7号物品，**编号为0**，六进制表示为$(1)_6$ == $(1)_{10}$
1号.单独的一个8号物品，**编号为1**，六进制表示为$(10)_6$ ==$(6)_{10}$
2号.由3个7号物品构成的物品，**编号为2**，六进制表示为$(3)_6$ ==$(18)_{10}$
3号.由1个7号物品和2个8号物品构成的物品，**编号为3**，六进制表示为$(21)_6$ == $(13)_{10}$

$(15)_{10}==(13)_{10}+(1)_{10}+(1)_{10}$

$(23)_6 ==(21)_6 +(1)_6 +(1)_6$

2个1号物品,3个0号物品 == 1个3号物品 +两个0号物品

分析3：

f[x]表示填满空间状态为x的背包所需最小费用

f[x]= min{ f[x-offer[y].stat]+offer[y].price }　　x要能放得下y

怎样**判断**空间状态为**x**的背包能否放得下空间状态为**y**的物品？
比如状态为$(23)_6$的背包 能否放得下状态为$(21)_6$的物品？

从右往左逐位比较x和y对应位置的数字

```
bool Check(int x,int y)//判断可用空间为x的背包能否放下体积为y的物品
{
    for(int i=0;i<b;i++)
    {
        if(x%6<y%6)return false;  //放不下
        x/=6; y/=6;
    }
    return true;                  //放得下
}
```

问题解决了！

状态压缩简单的说就是将复杂的多个数据表示的状态转换成某种进制的单个数字来表示，在实际处理的时候其实还是采用的十进制。