归并dp

一开始有2^n 个队伍， 某个人是k只队伍的球迷。 首先一开始1和2打， 3和4打.... 输的进入lower group, 赢的进入upper group, 比如分别是1,3,5,8和2,4,6,7, 然后1,3打，5,8打，2,4打，6,7打，1,3的输者直接被淘汰，1,3的赢者打2,4的输者，赢者为lower group, 而2,4的赢者在upper group, 直到两个组各剩一个人，然后最后打一局，问球迷可以看到几场自己喜欢球队的比赛?

第一轮比赛是分组，之后每进行一轮人数少一半，第i轮是每2^i个人中只有一个胜者，一个只输了一场的人，当第i+1轮时，相连的2^i 个人的胜者对打，此局胜者保留，输者和两个败者的赢者打，赢的保留，因此想到归并的方式，当进行到第n轮时，前2^{n-1} 和后 2^{n-1} 各保留一个，然后第$n+1$轮巅峰之战。

于是可以通过归并dp的方式，每次幂次合并，可定义$dp[i][j][x][y]$, 其中i 表示从第i个人开始的2^j个人，最后赢的人是否为喜欢的(x的01表示), 输的人是否为喜欢的(y的01表示), 由于第一轮比较特殊，特殊考虑，最后一轮就一局也得单独考虑。

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
const int maxn = 1e7 + 20;
const int INF = 0x3f3f3f3f;

int mark[1<<17], dp[18][1<<17][2][2], n, k, tmp, ans;

int main()
{
    scanf("%d %d", &n, &k);
    for(int i = 1; i <= k; i++)
    {
        scanf("%d", &tmp);
        mark[tmp-1] = 1;
    }
    memset(dp, -INF, sizeof(dp));      //一开始要初始化，这样防止在不存在的情况下考虑

    for(int i = 1; i <= n; i++)
    {
        if(i == 1)
        {
            for(int j = 0; j < (1<<n); j += 2)   //存在的情况，有喜欢的为1，否则为0
            {
                dp[1][j][mark[j]][mark[j+1]] = mark[j] | mark[j+1];
                dp[1][j][mark[j+1]][mark[j]] = mark[j] | mark[j+1];
            }
            continue;
        }

        for(int j = 0; j < (1<<n); j += (1<<i))    //第2轮到第n轮
        {
            for(int x1 = 0; x1 < 2; x1++)        //前半部分赢者x1输者y1
                for(int y1 = 0; y1 < 2; y1++)
                    for(int x2 = 0; x2 < 2; x2++)  //后半部分赢者x2输者y2
                        for(int y2 = 0; y2 < 2; y2++)
                        {
                            int cost = dp[i-1][j][x1][y1] + dp[i-1][j+(1<<(i-1))][x2][y2];
                            if(x1 || x2) cost++;        //赢者x1与x2对打
                            if(y1 || y2) cost++;        //输者y1与y2对打
                        //枚举八种情况
                            dp[i][j][x1][x2] = max(dp[i][j][x1][x2], cost + (x2 | y1));
                            dp[i][j][x1][x2] = max(dp[i][j][x1][x2], cost + (x2 | y2));

                            dp[i][j][x1][y1] = max(dp[i][j][x1][y1], cost + (x2 | y1));
```

```
                            dp[i][j][x1][y2] = max(dp[i][j][x1][y2], cost + (x2 | y2));

                            dp[i][j][x2][x1] = max(dp[i][j][x2][x1], cost + (x1 | y1));
                            dp[i][j][x2][x1] = max(dp[i][j][x2][x1], cost + (x1 | y2));

                            dp[i][j][x2][y1] = max(dp[i][j][x2][y1], cost + (x1 | y1));
                            dp[i][j][x2][y2] = max(dp[i][j][x2][y2], cost + (x1 | y2));
                        }
        }
    }
    ans = max(ans, dp[n][0][0][0]);         //第n+1轮单独考虑
    ans = max(ans, dp[n][0][0][1] + 1);
    ans = max(ans, dp[n][0][1][0] + 1);
    ans = max(ans, dp[n][0][1][1] + 1);

    printf("%d\n", ans);
}
```

轮廓线dp

给定一个n×m的矩阵，用1×2的方块去填满(彼此不能覆盖)，问有几种方法?

分析：若是从上到下，从右到左进行dp，每次只规定方块能向左向上放，可以发现(i, j)的状态和(i-1,j)与(i,j-1)的状态有关，即和前m个相关，我记涂上为1，没涂为0，dp[1<<m]来枚举前m个的状态(更前面的全是1)。

```
int n, m, cur;
ll dp[2][1<<11];

void update(int pre, int crt)     //pre为之前m个的状态，crt为pre可以变成的状态(m+1个)
{
    if(crt & (1<<m)) dp[cur][crt^(1<<m)] += dp[1-cur][pre];   //crt的第一个状态必须为1
}
int main()
{
    while(~scanf("%d %d", &n, &m))
    {
        if(n == 0 && m == 0) break;
        if(m > n) swap(n, m);
        memset(dp, 0, sizeof(dp));
        dp[cur][(1<<m)-1] = 1;
        for(int i = 0; i < n; i++)
            for(int j  = 0; j < m; j++)
            {
                cur ^= 1;                   //每次操作都会让cur在0与1之间翻转
                memset(dp[cur], 0, sizeof(dp[cur]));      //1-cur为前m个的轮廓线
                for(int k = 0; k < (1<<m); k++)        //枚举1-cur轮廓线的状态
                {                                      //分析可以得到新的轮廓线的状态
                    update(k, k<<1);                       //不涂，等待右下方
                    if(i && !(k & 1<<(m-1))) update(k, ((k | 1<<(m-1))<<1) + 1);   //往上
                    if(j && !(k & 1)) update(k, ((k | 1)<<1) + 1);     //往左
                }
            }
        printf("%lld\n", dp[cur][(1<<m)-1]);
    }
}
```

离散化dp

一开始有属性a和b，这两个属性初始的时候均为0，每一天可以让a涨1点或b涨1点。我们共有 n 种奖励，第i种奖励有 x_i，y_i，z_i 三种属性，若 $a \geq x_i$ 且 $b \geq y_i$，则弱弱在接下来的每一天都可以得到 z_i 的分数。

问 m 天以后弱弱最多能得到多少分数，其中 $1 \leq n \leq 1000$，$1 \leq m \leq 2e9$，$x_i, y_i \leq 1e9$。

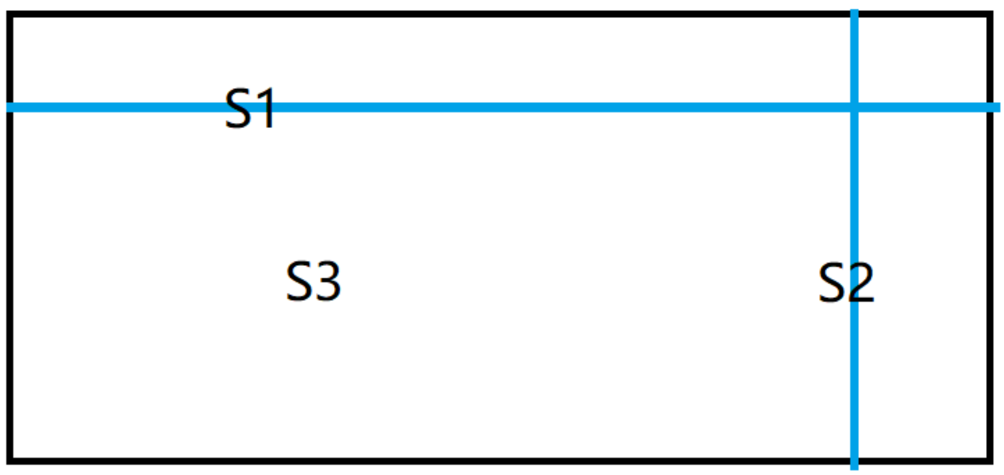
当 m 很小时，可以想到动态规划的方法。若 $dp[i][j]$ 表示 $i+j$ 天且 $a=i$，$b=j$ 时得到的最大奖励，那么这个值肯定和之前状态有关， 若用 $v[i][j]$ 表示 $a=i$，$b=j$ 时这一天可以得到的奖励，那么可以得到这样的状态转移方程：

$dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + v[i][j]$

而对于$v[i][j]$, 我们可以用前缀和的方法进行求解，若是初始时 $v[i][j]$ 表示 $x=i$，$b=j$ 的属性的 z 值，可以得到这样的状态转移方程：

$v[i][j] += v[i-1][j] + v[i][j-1] - v[i-1][j-1]$

这个可以类比一维的求前缀和得到，也可以简单地类比面积计算得到：



若 S 初始表示右上那一块小的，若要表示大长方形面积，对应 $v[i][j]$，S_1 对应左边两块，对应 $v[i-1][j]$，S_2 对应下边两块，对应 $v[i][j-1]$, S_3 对应左下那块，对应 $v[i-1][j-1]$, 则 $S+=S_1+S_2-S_3$，对应于上述求前缀和转移式。

而对于ans:

$ans = max(ans, dp[i][j] + v[i][j] * (m-i-j)) \quad (i+j \geq m)$

而这一题 m 的范围是很大的，通过直接枚举天数进行dp肯定会超时。所以这时候可以想到离散化。

比如当有3个奖励: $(100, 100, 1), (20, 30, 2), (30, 20, 3)$，对于 $dp[100][100]$, 只需要考虑 $dp[20][30], dp[30][20], dp[30, 30]$ 这些值怎么变化到 $dp[100][100]$ 就好了，而无需从 $dp[100][99], dp[99][100]$ 这些值得到，因此我们只需要考虑这些奖励的 x_i, y_i 所组成的结点就好了。为此我们需要将 x_i, y_i 离散化：

```
scanf("%d %d", &n, &m);
for(int i = 1; i <= n; i++)
{
    scanf("%d %d %d", &pro[i].x, &pro[i].y, &pro[i].z);
    X[i] = pro[i].x;
    Y[i] = pro[i].y;
}
 //先排序
sort(X + 1, X + 1 + n);              //先排序
sort(Y + 1, Y + 1 + n);
//再去重，得到去重后的元素个数
cnt1 = unique(X + 1, X + 1 + n) - (X + 1);
cnt2 = unique(Y + 1, Y + 1 + n) - (Y + 1);
for(int i = 1; i <= n; i++)
{
    //将值从小到大排序后，依次映射到1， 2....
    int x = lower_bound(X + 1, X + 1 + cnt1, pro[i].x) - X,
        y = lower_bound(Y + 1, Y + 1 + cnt2, pro[i].y) - Y;
    v[x][y] += pro[i].z;
```

```
    }
```

其中 $X[i]$ 中的值分别映射到 $1, 2...$

经过这样的映射之后，我们只对奖励里出现过的值进行遍历，比如上面的例子从 1 遍历到 100 可以变成 1 遍历到 3 (其中 1 对应 20， 2 对应 30， 3 对应 100)，那么原来 $dp[i][j]$ 与 $dp[i-1][j]$ 相差了 1 天， 现在就相差 $X[i] - X[i-1]$ 天。

求 $v[i][j]$ 的转移方程没有变，而求 $dp[i][j]$ 的转移方程为：
$dp[i][j] = max(dp[i-1][j] + v[i-1][j] * (X[i] - X[i-1] - 1), dp[i][j-1] + v[i][j-1] * (Y[j] - Y[j-1] - 1)) + v[i][j]$

对于每一个 $dp[i][j]$，若 $i + j \le m$， 那么：
$ans = max(ans, dp[i][j] + (m - X[i] - Y[j]) * v[i][j])$

这样得到的即为需要的最大值，其中离散化操作为 $O(nlogn)$，求前缀和，dp 为 $O(n^2)$

代码

```
#include <bits/stdc++.h>

using namespace std;
typedef unsigned long long ll;
const int maxn = 1005;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

struct node
{
    int x, y, z;
}pro[maxn];

int n, m, X[maxn], Y[maxn], cnt1, cnt2;
ll dp[maxn][maxn], v[maxn][maxn], ans;

int main()
{
    scanf("%d %d", &n, &m);            //进行离散化操作
    for(int i = 1; i <= n; i++)
    {
        scanf("%d %d %d", &pro[i].x, &pro[i].y, &pro[i].z);
        X[i] = pro[i].x;
        Y[i] = pro[i].y;
    }
    sort(X + 1, X + 1 + n);       //排序
    sort(Y + 1, Y + 1 + n);
    cnt1 = unique(X + 1, X + 1 + n) - (X + 1);     //去重
    cnt2 = unique(Y + 1, Y + 1 + n) - (Y + 1);

    for(int i = 1; i <= n; i++)
    {
        ///将值从小到大排序后，依次映射到1， 2....
        int x = lower_bound(X + 1, X + 1 + cnt1, pro[i].x) - X,
            y = lower_bound(Y + 1, Y + 1 + cnt2, pro[i].y) - Y;
        v[x][y] += pro[i].z;
    }
    for(int i = 1; i <= cnt1; i++)      //求前缀和
        for(int j = 1; j <= cnt2; j++)
            v[i][j] += v[i-1][j] + v[i][j-1] - v[i-1][j-1];

    for(int i = 1; i <= cnt1; i++)
    {
        for(int j = 1; j <= cnt2; j++)    //进行dp
```

```
        {
            dp[i][j] = max(dp[i-1][j] + v[i-1][j] * (X[i] - X[i-1] - 1), dp[i][j-1] + v[i]
[j-1] * (Y[j] - Y[j-1] - 1)) + v[i][j];
            if(X[i] + Y[j] <= m)        //这个if判断很关键!
                ans = max(ans, dp[i][j] + (m - X[i] - Y[j]) * v[i][j]);
        }
    }

    printf("%lld\n", ans);
}
```

区间dp

输入两个字符串A和B($|A|, |B| < 50$)，合并成一个串C，属于A和B的字符在C中顺序保持不变。如"abc"和"xyz"可以被组合成"axbycz"或"abxcyz"等。

我们定义字符串的价值为其最长回文子串的长度（回文串表示从正反两边看完全一致的字符串，如"aba"和"xyyx"）。需要求出所有可能的C中价值最大的字符串，输出这个最大价值。

若是这题暴力把所有合成的 C 串找出来，则共有 C_{100}^{50} 种可能，这种指数复杂度是不能承受的。而类比找一个字符串的回文串，若是只是单个字符串，我们可以通过dp进行 $O(n^2)$ 的算法:

```
//字符串从s[1]开始存储，长度为len
for(int L = 1; L <= len; L++)
{
    for(int i = 1; i + L  - 1 <= len; i++)
    {
        int j = i + L - 1;
        if(s[i] == s[j] && (L <= 2 || dp[i+1][j-1]))
        {
            dp[i][j] = 1;
            ans = max(ans, L);
        }
    }
}
```

若是要用动态规划，**区间dp的思想是**考虑从小规模的状态转移到大规模的状态，因此肯定是从长度较小的回文子串转移到长度较大的回文子串，即如果一个回文串首尾加上同样的字母，可以构成一个新的更长的回文串。

若是用 $dp[i][j]$ 表示区间 $[i, j]$ 构成的子串是不是回文串(01表示)，那么一个区间是回文串有以下三种可能:

①$i = j$，即单个字符是回文串;

②$i = j - 1$ 并且 $s[i] = s[j]$，即两个一样的字符是回文串;

③ $i < j - 1$ 并且 $s[i] = s[j]$，$dp[i+1][j-1] = 1$，即一个回文串首尾加上同样的字母，可以构成一个新的更长的回文串。

因此若是求单个字符串的回文子串，大循环是区间长度，小循环是区间起始位置进行双重循环就行，当然还有更优秀的马拉车算法($O(n)$)。

对于这一题，我们需要考虑的是两个字符串组合得到的回文串，则需要四维 $dp[i][k][j][l]$, 表示 $A[i][k], B[j][l]$ 是否可以组成一个回文串。记 $len_1 = k - i + 1$，$len_2 = l - j + 1$，同样地，我们需要考虑状态转移:

①如果 $len_1 + len_2 \leq 1$, 则 $dp[i][k][j][l]$ 为 1。等于 1 时两个字符串的截取一个为空，一个长度为1，自然是回文串；等于 0 时两个字符串都截取为空，其实是没有回文串的定义的，但是为了之后的转移方便(如从 $dp[0][0][0][0]$ 转移到 $dp[1][1][1][1]$), 定义为1。

②如果 $len_1 > 1$, 那么对于 $dp[i][k][j][l]$, 若 $A[i] = A[k]$ 且 $dp[i+1][k-1][j][l] = 1$，那么首尾加上 $A[i]$，$A[k]$ 还是回文串，则$dp[i][k][j][l] = 1$。同理对于$len_2 > 1$。

③如果$len_1 > 0$ 且 $len_2 > 0$, 若$A[i] = B[l]$ 且 $dp[i+1][k][j][l-1] = 1$，那么在首尾加上 $A[i], B[l]$ 还是回文串，则$dp[i][k][j][l] = 1$。同理对于$A[k] = B[j]$ 且 $dp[i][k-1][j+1][l] = 1$。

所以以上就得出了**从小回文串转移到大回文串的区间动态规划过程**。通过四重循环，前两重为A, B串截取的长度，后两重为 A, B串的区间起始位置，在计算时记录最大值就可以解决了。

代码

```
#include <bits/stdc++.h>

using namespace std;
typedef unsigned long long ll;
const int maxn = 1005;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

char a[52], b[52];
int f[52][52][52][52], len1, len2, ans;

int main()
{
    int t;
    scanf("%d", &t);
    while(t--)
    {
        ans = 1;
        scanf("%s %s", a + 1, b + 1);

        len1 = strlen(a + 1), len2 = strlen(b + 1);

        for(int k1 = 0; k1 <= len1; k1++)        //前两重循环区间长度
            for(int k2 = 0; k2 <= len2; k2++)
                for(int i = 1; i + k1 - 1 <= len1; i++)       //后两重循环区间起始位置
                    for(int j = 1; j + k2 - 1 <= len2; j++)
                    {
                        int k = i + k1 - 1, l = j + k2 - 1;
                        if(k1 + k2 <= 1)            //根据状态转移方程得出
                            f[i][k][j][l] = 1;
                        else
                        {
                            f[i][k][j][l] = 0;      //由于有多次询问，因此需要清零防止上次的dp数据干扰

                            if(k1 > 1) f[i][k][j][l] |= (f[i+1][k-1][j][l] && a[i] == a[k]);
                            if(k2 > 1) f[i][k][j][l] |= (f[i][k][j+1][l-1] && b[j] == b[l]);
                            if(k1 && k2) f[i][k][j][l] |= (f[i+1][k][j][l-1] && a[i] == b[l]);

                            if(k1 && k2) f[i][k][j][l] |= (f[i][k-1][j+1][l] && a[k] == b[j]);
                        }

                        if(f[i][k][j][l]) ans = max(ans, k1 + k2);
                    }

        printf("%d\n", ans);
    }
}
```

除了转移方程，还需要特别注意动态规划过程边界的处理。

树型dp

过程：一般先算子树然后进行合并，在实现上与二叉树的后序遍历类似，先遍历子树，遍历完之后把子树的值合并给父亲。

NC24953(树的最小支配集)：一个点被盖，它自己和与它相邻的点都算被覆盖。给你一棵无向树，问你最少用多

少个点可以覆盖掉所有其他的点?

若是边覆盖的话，一条边要么被某个结点自己覆盖，要么被其儿子覆盖。但是现在是覆盖点，一个点既可以被儿子覆盖，也可以被自己覆盖，也可以被父亲覆盖，所以要定义三个状态了。

$dp[i][0]$ 表示某个结点自身有覆盖，子树的最小点数， $dp[i][1]$ 表示某个结点自己没有覆盖，但是某个儿子覆盖， $dp[i][2]$ 表示某个结点自己没有覆盖，但是其父亲覆盖。

易知 $dp[i][0] = 1 + \sum min(dp[j][0], dp[j][1], dp[j][2])$ (j 为 i 的儿子)， $dp[i][2] = \sum min(dp[j][0], dp[j][1])$， $dp[i][1]$ 比较复杂一些，需要至少有一个儿子覆盖自身，则 $dp[i][1] = \sum min(dp[j][0], dp[j][1]) + inc$ $inc = max(0, min\ dp[j][0] - dp[j][1])$

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
const int maxn = 1e4 + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

int n, m;

struct edge                          //链式前向星
{
    int to, next;
}e[maxn*2];

int head[maxn], num;         //head为0表示搜索到了尽头

void add_edge(int u, int v)
{
    e[++num].to = v;
    e[num].next = head[u];
    head[u] = num;
}

int dp[maxn][3];

void dfs(int x, int fa)
{
    int flag = 0, tmp = INF;
    dp[x][0] = 1;
    for(int i = head[x]; i; i = e[i].next)
    {
        int to = e[i].to;
        if(to == fa)  continue;
        dfs(to, x);
        dp[x][0] += min(dp[to][0], min(dp[to][1], dp[to][2]));
        dp[x][2] += min(dp[to][0], dp[to][1]);
        if(dp[to][0] <= dp[to][1])
        {
            dp[x][1] += dp[to][0];
            flag = 1;
        }
        else
        {
            dp[x][1] += dp[to][1];
            tmp = min(tmp, dp[to][0] - dp[to][1]);
        }
    }
    if(flag == 0)  dp[x][1] += tmp;
}
```

```
int main()
{
    scanf("%d", &n);
    for(int i = 1; i < n; i++)
    {
        int x, y;
        scanf("%d %d", &x, &y);
        add_edge(x, y);  add_edge(y, x);
    }
    dfs(1, 0);
    printf("%d\n", min(dp[1][0], dp[1][1]));
}
```

NC24953(二分)：给定一棵 n 个结点的树，每条边有个边权，然后切去一些边，让根结点和除根结点外的叶子结点不连通，切去边的总值不能超过 m，问所有的方案中，切去最大边最小是多少?

可以知道最大边越大，越容易实现，所求的值是单调的，所以可以进行二分，进行树上的 dp；我的二分方法是让 L 一定不可行，让 R 一定可行，只要这两个值差距小于等于 1，答案就是 R。

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
const int maxn = 1010;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

int n;

struct edge                         //链式前向星
{
    int to, next, w;
}e[maxn*2];

int head[maxn], num;          //head为0表示搜索到了尽头

void add_edge(int u, int v, int w)
{
    e[++num].to = v;
    e[num].w = w;
    e[num].next = head[u];
    head[u] = num;
}

int mx, m, cur, d[maxn];

bool dfs(int x, int fa)           //x的子树断掉所有叶子结点是否可行，若可行，最小值记录在d[x]中
{
    if(e[head[x]].next == 0 && fa > 0)        //若 x 不是根结点且为叶子结点
        return false;

    d[x] = 0;
    for(int i = head[x]; i; i = e[i].next)
    {
        //cout<<x<<" "<<e[x].w<<" "<<cur<<endl;
        if(e[i].to == fa)  continue;
        if(!dfs(e[i].to, x))
        {
            if(e[i].w > cur)  return false;
```

```
            d[x] += e[i].w;

        }
        else if(e[i].w > cur)  d[x] += d[e[i].to];
        else d[x] += min(d[e[i].to], e[i].w);
    }
    if(d[x] > m)  return false;
    return true;
}

int main()
{
    scanf("%d %d", &n, &m);
    for(int i = 1; i < n; i++)
    {
        int x, y, z;
        scanf("%d %d %d", &x, &y, &z);
        mx = max(mx, z);
        add_edge(x, y, z);  add_edge(y, x, z);
    }

    cur = mx;
    if(!dfs(1, 0))                          //判断右端点是否可行
    {
        printf("%d\n", -1);
        return 0;
    }
    int l = 1, r = mx;
    while(r - l > 1)                          //二分
    {
        cur = (l + r) / 2;
        if(dfs(1, 0))  r = cur;
        else l = cur;
    }
    printf("%d\n", r);
}
```

状压dp

过程：一种直观而高效地表示复杂状态的手段

NC20240(按行考虑)：在N×N的棋盘里面放K个国王，使他们互不攻击，共有多少种摆放方案。国王能攻击到它上下左右，以及左上 左下右上右下八个方向上附近的各一个格子，共8个格子。

若是要单独考虑每一个国王的摆放，有一些困难(插头dp ?)，所以可以按照行来考虑，以一个二进制数 x 表示一行，首先这一行不能有相邻的 1， 即 $(x\&(x+1))! = 0$, 其次与上一行(记录状态为 y)也不能互相攻击，即 $(x\&(y+1))! = 0$, $(x\&(y-1))! = 0$, $(x\&y)! = 0$。所以我们可以得到转移方程： 第 i 行 摆了 j 个国王且状态为 k ： $f[i][j][k]+ = f[i-1][j-num[k]][p]$

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
const int maxn = 1010;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;
```

```
int n, K;
ll dp[10][82][600], ans;

int getNum(int x)
{
    int cnt = 0;
    while(x)
    {
        cnt += x & 1;
        x >>= 1;
    }
    return cnt;
}

int main()
{
    scanf("%d %d", &n, &K);
    dp[0][0][0] = 1;

    for(int i = 1; i <= n; i++)
    {
        for(int j = 0; j < (1 << n); j++)
        {
            if(j & (j<<1)) continue;
            int num = getNum(j);
            if(num > K) continue;
            for(int k = 0; k < (1 << n); k++)
            {
                if((k & (k<<1)) || (k & j) || (k & (j<<1)) || (k & (j>>1)))
                    continue;
                for(int h = 0; h + num <= K; h++)
                    dp[i][h+num][j] += dp[i-1][h][k];
            }
        }
    }


    for(int i = 0; i < (1 << n); i++)
        ans += dp[n][K][i];
    printf("%lld\n", ans);
}
```

NC16544(压缩初始状态)：题目很简单，给定一个无向图，求出长度大于2的简单环(顶点不重复出现)的数量，相应的取模。($n \leq 20$)

若是用搜索的话，当完全图时复杂度会到达 $O(n!)$。若是用状压 dp 的话，一维用01串表示去过的点的话，那必须还得记录这条路径的起点和终点，若是开成三维数组的话，内存较大，并且最后答案肯定还有去重，比较麻烦。其实01串不仅可以表示去过的点，还可以表示出路径的起始位置，**我们不妨用最低为 1 的位表示路径的起始位置，并规定这条路径只能走到标号比起始位置大的点**，这样的话我们更新只会将后面的 0 变成 1， 前面的 1 永远是 1(详细的转移方程见代码)，而由于一个环可以顺时针也可以逆时针，所以最后答案要除以2。

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
const int maxn = (1<<20) + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;
```

```
ll dp[maxn][20], ans[22], a[22];
int n, m, K, d[22][22];

int main()
{
    scanf("%d %d %d", &n, &m, &K);
    for(int i = 1; i <= m; i++)
    {
        int x, y;
        scanf("%d %d", &x, &y);
        d[x-1][y-1] = d[y-1][x-1] = 1;      //切记状压从第0位开始，所以点也是从0计数
    }

    for(int i = 0; i < n - 2; i++)         //以点 i 为起始点
        dp[1<<i][i] = 1;
    for(int i = 1; i < (1<<n); i++)
    {
        int s = 0;                          //s记录最低位1，即路径的起点
        for(;;s++)
            if(i & (1<<s))
                break;
        if(s >= n - 2)  continue;
        for(int j = s; j < n; j++)              //s前面不会有1
        {
            if(((1<<j) & i) == 0 || dp[i][j] == 0)  continue;  //找到要扩展出路径的点
            for(int k = s + 1; k < n; k++)                      //找到被扩展的点
            {
                if(((1<<k) & i) || d[j][k] == 0) continue;
                dp[i | (1<<k)][k] += dp[i][j];
                dp[i | (1<<k)][k] %= mod;
            }
            if(d[j][s])                              //如果可以形成一个环
            {
                int num = __builtin_popcount(i);         //统计1的个数
                if(num >= 3)
                    ans[num] += dp[i][j],  ans[num] %= mod;
            }
        }
    }
    for(int i = 3; i <= n; i++)
        a[i%K] = (a[i%K] + ans[i]) % mod;

    for(int i = 0; i < K; i++)
        printf("%lld\n", (a[i] * (mod + 1) / 2) % mod);
}
```

数位dp

过程：按位考虑。

NC15035：题意是说令人讨厌的数字定义成含有4或者38的数字，给定一个区间，问里面有多少个这样令人讨厌的数？

首先可以将各个位置上的数字拆分，$f[i][st]$表示 从高往低数前 $i-1$ 位的状态确定为 st，给第 i 位到第 1 位填数字有多少种填法。0 表示既没有4也没有 38， 1 表示既没有4也没有38并且第 $i-1$ 位为3， 2 表示前面已经有 4 和 38。

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;
```

```
typedef long long ll;
typedef pair<int, int> P;
const int maxn = 1e5 + 10;
const int INF = 0x3f3f3f3f;
const ll mod = 998244353;

int f[10][3], a[10];

int dp(int pos, int st, int flag)
{
//flag 表示是否能直接返回值，也就是前pos - 1位和原数是否一样
    if(pos == 0)  return st == 2;
    if(flag && f[pos][st] != -1)  return f[pos][st];
    int x = flag? 9: a[pos];
    int ans = 0;
    for(int i = 0 ; i <= x; i++)
    {
        if(i == 4 || st == 2 || (st == 1 && i == 8))
            ans += dp(pos - 1, 2, flag || i < a[pos]);
        else if(i == 3)  ans += dp(pos - 1, 1, flag || i < a[pos]);
        else ans += dp(pos - 1, 0, flag || i < a[pos]);
    }
    if(flag)  f[pos][st] = ans;
    return ans;
}

int calc(int x)
{
    if(x <= 0)  return 0;
    memset(a, 0, sizeof(a));
    int pos = 0;
    while(x)
    {
        a[++pos] = x % 10;
        x /= 10;
    }
    return dp(pos, 0, 0);
}

int main()
{
    int n, m;
    memset(f, -1, sizeof(f));
    while(~scanf("%d %d", &n, &m))
    {
        if(n == 0 && m == 0) break;
        printf("%d\n", calc(m) - calc(n - 1));
    }
}
```

NC20665：给定一个区间，问这个里面有多少数能被7整除且位数和为7。
在数位dp的时候，比较好维护的是数位和(%7结果) 和前面的数 %7 的结果，维护这两个即可。

```
#include <bits/stdc++.h>
#define pb push_back

using namespace std;

typedef long long ll;
typedef pair<int, int> P;
```

```
const int maxn = 1e5 + 10;
const int INF = 0x3f3f3f3f;
const int mod = 20020219;

ll l, r, f[30][7][7];
int n, a[25];

ll dp(int pos, int pre, int sum, int flag)
{
    if(pos == 0)  return (pre == 0 && sum == 0);
    if(flag && f[pos][pre][sum] != -1)  return f[pos][pre][sum];
    int maxi = flag? 9: a[pos], tmp = 10 * pre;
    ll ans = 0;
    for(int i = 0 ; i <= maxi; i++)
    {
        ans += dp(pos - 1, (tmp + i) % 7, (sum + i) % 7, flag || i < maxi);
    }
    if(flag)  f[pos][pre][sum] = ans;
    return ans;
}

ll calc(ll x)
{
    if(x < 0)  return 0;
    memset(a, 0, sizeof(a));
    int pos = 0;
    while(x)
    {
        a[++pos] = x % 10;
        x /= 10;
    }
    return dp(pos, 0, 0, 0);
}

int main()
{
    for(int i = 0; i < 25; i++)
        for(int j = 0; j < 7; j++)
            for(int k = 0; k < 7; k++)
                f[i][j][k] = -1;

    while(~scanf("%lld %lld", &l, &r))
    {
        if(r == 0 && l == 0) break;
        printf("%lld\n", calc(r) - calc(l - 1));
    }
}
```