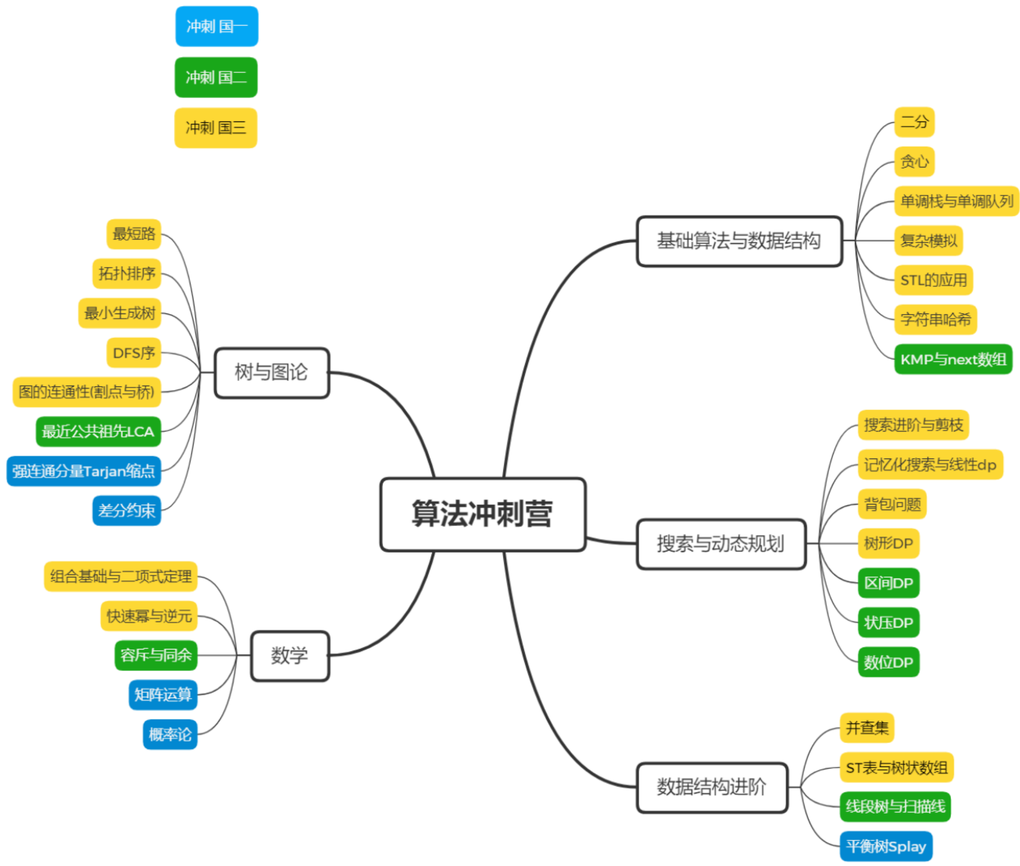
蓝桥杯30天算法冲刺集训



Day-11 状态压缩DP

状态压缩

首先让我们来看一个简单的例子：现在有一个含 8 个数的集合 $\{7,6,5,4,3,2,1\}$。你现在要从中选择个子集。如果我用 0 表示这个数选了，1 表示这个数没选，那么每一种选择子集的方法都对应着一个八位二进制数。比如：

如果我选择了 $\{3,1,0\}$，那么对应的二进制数就是 $00001011_{(2)}$。如果我选择了 $\{7,6\}$，那对应的二进制数就是 $11000000_{(2)}$。

这种用一个二进制数来表示集合的选取状态的方法，被我们叫做状态压缩。

如果我们在设计 DP 状态的时候，其中一维用一个二进制数来表示集合的选取情况，那就被叫做状态压缩动态规划，简称状压 DP。

在我们用一个二进制数来表示集合之后，我们还需要学会使用位运算来表示集合的常见操作。

常见的就是，两个集合的 **并**，就对应着两个二进制数的 **按位或(|)**；比如上面举出的例子 $\{3,1,0\}\cup\{7,6\}=\{7,6,3,1,0\}$，那就相当于 $00001011_{(2)}|11000000_{(2)}=11001011_{(2)}$。

两个集合的 **交**，就对应着两个二进制数的 **按位与**。比如 $\{3,1,0\}\cap\{3,2,1\}=\{3,1\}$，就相当于 $00001011_{(2)}\&00001110_{(2)}=00001010_{(2)}$。

一个集合 A 是另一个集合 B 的 **子集**，就相当于是 A 对应的二进制数按位或上 B 对应的二进制数，得到的结果是 B 对应的二进制数。

一个集合的 **补集**，就相当于是这个集合的全集 2^n-1 减去这个集合的二进制数。

当然也有不常见的，比如说像枚举子集的技巧，这个我们会在后面的题目中看到。

状态压缩中的常见位运算

下文中的 a 表示十进制下的整数。

1、获得第 i 位的数字： `(a>>i)&1` 或者 `a&(1<<i)`

很好理解，我们知道可以用 `&1` 来提取最后一位的数，那么我们现在要提取第 i 位数，就直接把第 i 位数变成最后一位即可（直接右移）。或者，我们可以直接 `&(1<<i` ，也能达到我们的目的。

2、设置第 i 位为 1： `a|=(1<<i)`

我们知道强制赋值用 `|` 运算，所以就直接强制 `|` 上第 i 位即可。

3、设置第 i 位为 0： `a&=~(1<<i)`

这里比较难以理解。其实很简单，我们知道非 `~` 运算是按位取反，`(1<<i)` 非一下就变成了第 i 为是 0，其它全是 1 的二进制串。这样再一与原数进行 `&` 运算，原数的第 i 位无论是什么都会变成 0，而其他位不会改变（实在不明白的可以用纸笔进行推演）。

4、把第 i 位取反： `a=a^(1<<i)`

1 左移 i 位之后再进行异或，我们就会发现，如果原数第 i 位是 0，一异或就变成 1，否则变成 0。

5、取出一个数的最后一个1： `a&(-a)`

学过树状数组的同学会发现，这就是树状数组的 $lowbit$。事实上，这和树状数组的原理是一样的。

互不侵犯

因为每个国王都只会影响当前行，上一行和下一行，所以设计 DP 状态的时候可以以行号作为一个维度。同时题目对放多少个国王有要求，所以还要设一维表示放了多少个国王。接着我们从第 n 行开始考虑。

首先，如果只考虑第一行，那么

如果我们在第 n 行找到了一些摆法，那么受到第 $n-1$ 行的影响，第 $n-1$ 行就会有一些位置不能放棋。如果我们再在 $n-1$ 行可以放棋的位置放上了棋子，那又会导致 $n-2$ 行的一些位置不能放棋。

如果能够把当前行哪些位置可以放棋，哪些位置不能放棋设计进状态里，那就可以实现转移了。那怎么把哪些位置可以放棋，哪些位置不能放棋设计进状态里呢？可以用二进制！用一个二进制数，如果对应的位为 1，就表示这个位置可以放棋，如果对应的位为 0 ，就表示这个位置不能放棋。

所以现在我们设 $f[i][S][k]$ 表示已经处理完了前 i 行，且第 i 行能够放的位置状态为二进制数 S，放置了 k 个国王的方案数。进行状态转移的时候，我们需要枚举第 $i-1$ 行放棋子的情况。

假设我们枚举出来的第 $i-1$ 行放棋子的情况用二进制数表示是 T。那么只有当 $T \subseteq S$ 的时候，枚举的 T 才是合法的。

对于一个枚举出来的 T，不应该有相邻两个位置同时放棋子，也就是不应该有相邻的两个位同时为 1。否则这两个棋子可以相互攻击。

对于一个合法的 T，上一位不能被选择的位置就应该是 $((T >> 1) + (T <<) + T) \bmod 2^n$，将它取反就可以得到上一行可以被选择的位置。

因此可以得到

$$f[i][S][k] = \sum_{T\text{是合法的}} f[i-1][2^n - ((T >> 1) + (T <<) + T) \bmod 2^n][k - |T|]$$

这样时间复杂度就是 $O(nk4^n)$，相比原来有了很大优化。而且可以注意到 k 如果过大那么一定是无解的，分析可得 k 应该不会超过 25，否则答案一定是 0。同时又可以发现有很多无用的 T，如果提前预处理出所有合法的状态，复杂度是远小于这个值的。

```
#include <bits/stdc++.h>
using namespace std;

int n, K;
int st[100], ks[100], cnt; // st[i]: 第 i 种合法状态
long long f[10][100][100];
long long ans;
int popcount(int x) { // 统计 x 状态中 1 的个数
```

```
    int num = 0;
    while(x) {
        x &= x-1;
        num++;
    }
    return num;
}

int main(){
    cin >> n >> K;
    // 根据放置规则，预处理出所有的合法状态
    for(int i = 0; i <= (1<<n)-1; i++) {
        if((i&(i<<1)) || (i&(i>>1))) // 没有相邻的国王
            continue;
        st[++cnt] = i;
        ks[cnt] = popcount(i);

    }
    for(int i = 1; i <= cnt; i++)
        f[1][i][ks[i]] = 1;  // 第一层放了 ks[i] 个国王，且国王位置状态为 i

    // DP
    for(int i = 2; i <= n; i++)
        for(int j = 1; j <= cnt; j++) // 枚举当前行的状态
            for(int k = 1; k <= cnt; k++) {  // 枚举上一行的状态
                if((st[j]&st[k]) ||             // 有同一列都有国王
                   ((st[j]<<1)&st[k]) ||        // 右下一列有国王
                   ((st[j]>>1)&st[k]))          // 左下一列有国王
                    continue;
                for(int p = K; p >= ks[j]; p--)
                    f[i][j][p] += f[i-1][k][p-ks[j]];
            }
    for(int i = 1; i <= cnt; i++) {
        ans += f[n][i][K];
    }
    cout <<ans;
    return 0;
```

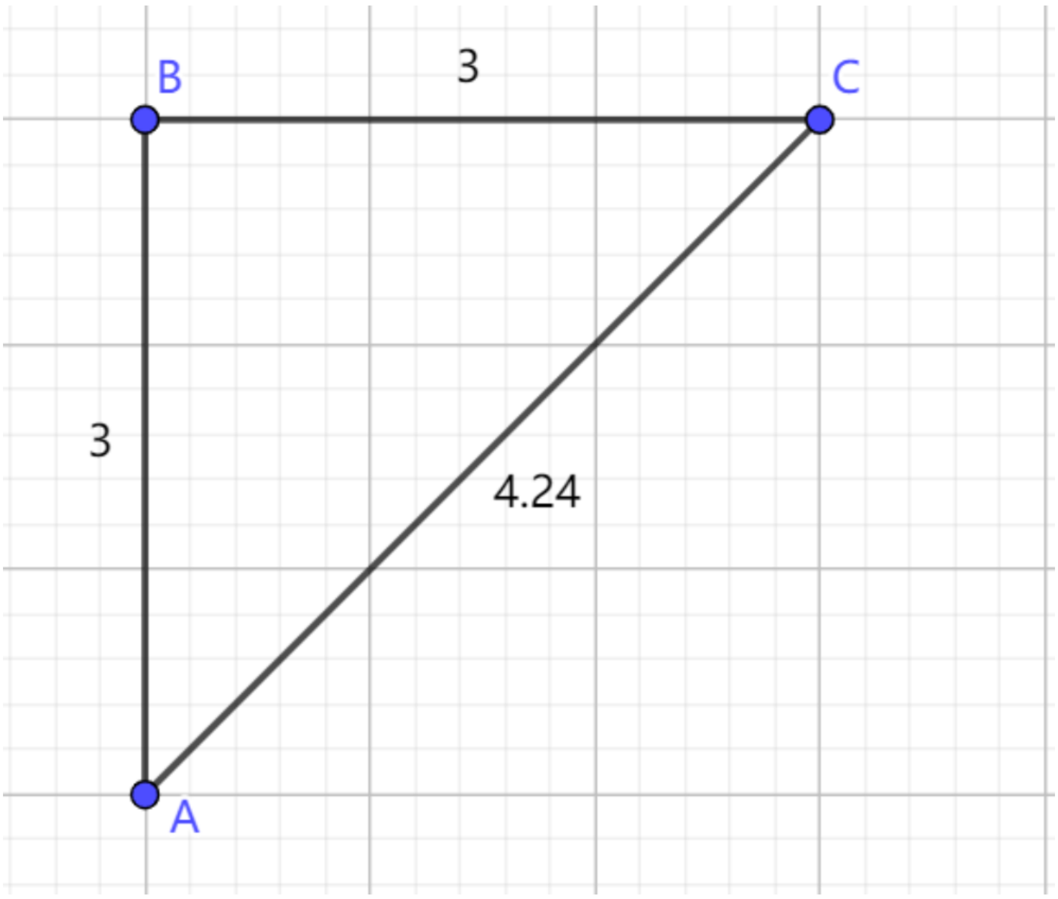
```
}
```

蓝桥杯真题

补给（蓝桥杯Java2020B组国赛）

状态压缩不错的板子题。

首先题目要求我们每次飞行最长的长度需要小于等于D，那么我们就有可能没法直飞，只能通过中转飞行，也就是如下图所示：



假设我们现在$D=4$那么我们从A到C的话距离可以算出来是$3\sqrt{2}$因为我们限制了一次飞行的最远距离不能超过D，所以只能先从A到B，再从B到C，最终需要至少飞行$3+3=6$的距离。

那么我们怎么去处理这种两点之间的距离超过D的情况呢？实际上我们根据两点之间的距离公式算出来两点之间的距离之后，如果两点之间的距离超过了D那么就设置一个极大值类似 `INT_MAX` 即可，因为$1 \le x_i, y_i \le 10^4, 1 \le D \le 10^5$。

那么我们怎么去表示A到C在当前最远距离小于等于D限制下的最近距离为6呢?

实际上直接使用最短路就可以了，因为$1 \leq n \leq 20$，所以直接用Floyd最短路即可。

然后我们就可以状压DP了，我们可以设置$f[i][j]$**代表当前状态为i，最后的终点是j的使用的最短距离**

那么我们的最终答案是不是就是:

$$ans = min_{1 \leq i \leq n}\{f[2^n - 1][i] + d[i][1]\}$$

其中$d[i][j]$代表从i到j所需要的最小花费。

那么我们中间的状态如何去转移?

我们需要枚举二进制来表示现在的已经去的机场，然后再枚举两个结束的端点j和k，具体代码如下:

```
for (int i = 1; i < 1<<n; ++i)//阶段
        for (int j = 0; j < n; ++j) if (i>>j&1)
            for (int k = 0; k < n; ++k) if ((i^1<<j)>>k&1)//状
态
                f[i][j] = min(f[i][j], f[i^1<<j][k]+dis[k]
[j]);//转移
```

首先我们看到第一个判断

```
if (i>>j&1)
```

判断出来是不是当前这个状态能以j结尾，如果可以的再进行第三个for循环再去判断

```
if ((i^1<<j)>>k&1)
```

上面这一行的意思是去除j这个结尾之后，还有没有以k为结尾的状态，如果有的话那么就可以执行下面的转移了。因为我们已经判断了j这一位置二进制已经有1了，所以我们 `(i^1<<j)` 是用来把这一位置上的1变换成0

```
f[i][j] = min(f[i][j], f[i^1<<j][k]+dis[k][j])
```

然后因为求min，我们可以最开始先把$f[i][j]$都设置成一个极大值类似 `INT_MAX` 。然后把设置$f[1][1]=0$做为初始状态，表示起点为1终点为1所需要的最短路程为0

最终时间复杂度为$O(n^3+2^n n^2)$

> **Caution**
>
> 上面的式子都是按照题意下标为1的情况，实际上我们下标从$0\sim n-1$会更容易我们实现整个代码，所以上面的给出的代码是从$0\sim n-1$的。
>
> 下面给出的示例代码也是按照下标从$0\sim n-1$来进行实现的，所以我们要设置$f[1][0]=0$

示例代码:

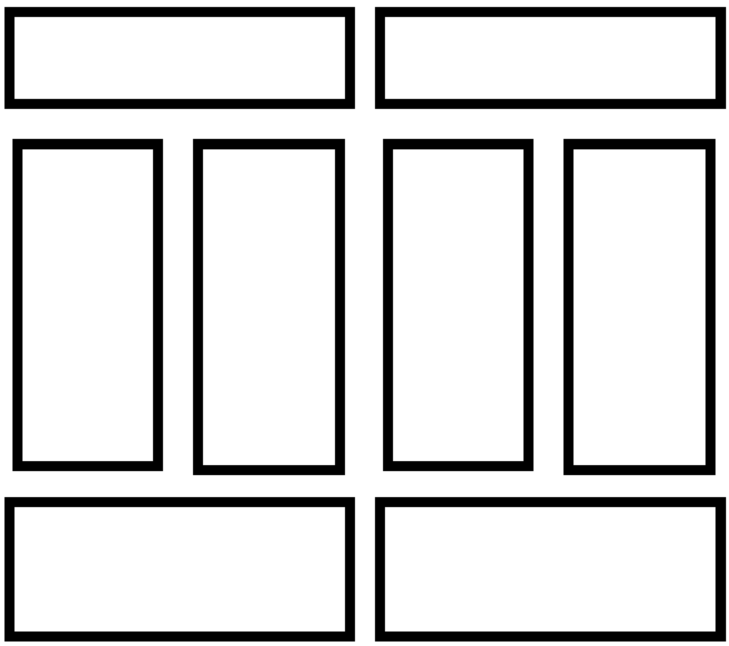
```
#include <bits/stdc++.h>
using namespace std;
const int N = 20, inf = 1e9;
int n, d;
double ans = inf, x[N], y[N], f[1<<N][N], dis[N][N];
int main() {
    cin >> n >> d;
    for (int i = 0; i < n; ++i) cin >> x[i] >> y[i];
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            double dist = sqrt((x[i]-x[j])*(x[i]-x[j])+(y[i]-
y[j])*(y[i]-y[j]));//欧几里得距离
            if (dist > d) dis[i][j] = inf;//不让该边过去
            else dis[i][j] = dist;
        }
    for (int k = 0; k < n; ++k)//Floyd 最短路
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                dis[i][j] = min(dis[i][j], dis[i][k]+dis[k]
[j]);
    memset(f, 0x7f, sizeof(f)), f[1][0] = 0;//初始化
    for (int i = 1; i < 1<<n; ++i)//阶段
        for (int j = 0; j < n; ++j) if (i>>j&1)
```

```
            for (int k = 0; k < n; ++k) if ((i^1<<j)>>k&1)//状态
                f[i][j] = min(f[i][j], f[i^1<<j][k]+dis[k][j]);//转移
    for (int i = 0; i < n; ++i) ans = min(ans, f[(1<<n)-1][i]+dis[i][0]);//注意题意要求回到第一个点
    printf("%.2lf", ans);
    return 0;
}
```

覆盖（蓝桥杯C/C++2021A组国赛）

这题状压，状态表示稍微复杂一点，不过也是练习状态压缩的好题。

首先我们需要知道一点就是纸片不只是只能按照$2 \times m$中这种方式排列的，他其实还可以类似下面这样竖着摆放：



如上图所示，这也是4×4一种的摆放方式，于是我们考虑使用状态压缩来解决，如果没有这种竖着的摆放方式，能够求出一横行的类似2×8的个数$S_{2\times8}$那么最终的答案$S_{8\times8} = S_{2\times8}^4$，不过这题并不是这样的。

那么我们可以考虑对于8行，每行的8列的状态进行一下枚举，然后看看相邻的行之间，这种竖着摆放的情况是否是合法的情况。

于是我们令$f[i][j]$代表$1\sim i$**行状态为**j**的方案总数**

其中状态j，可以表示如下：

我们可以用数字1来表示竖着摆放方块上面的部分，0来表示其他的部分。

那么我们可以发现假设当前行的状态是j，上一行的状态是k，那么我们要满足上一行当前二进制是1的，这一行必须是0，因为必须要先把这个竖着的方块放置好。然后上一行当前二进制是0的，这一行可以任意，也就是说上下两行肯定没有两个对应二进制都是1的情况。那么我们直接用$j\&k==0$就能判断两行之间是否是有冲突的竖着的方块了。但是横着的方块我们还得保证它每一个连续的部分都是偶数，也就是不能有横着的方块是$2\times odd$的情况，其中odd代表一个奇数。因为我们上一行已经用1来表示竖着的方块的上面的部分，0来表示其他的，就是横着的。我们下面这一行用0来表示其他的，我们可以先做一个运算令$tmp=j|k$，然后tmp里面的1就都是竖着的方块，0就都代表着我们当前行的横着的方块了，接着我们去判断0到底是不是连续的部分都是偶数个的即可。

这样我们就一行一行的计算答案，最终答案就是$f[8][0]$，为什么是$f[8][0]$呢？显然是第8行，然后0代表的是这一行都是横着的方块，肯定不可能再放竖着的了。

那么总体复杂度就是$O(2^8\times 2^8\times 8\times 8)=O(2^{22})$，也就是差不多$4e6$级别的运算次数。

最后别忘了再设置一个初始值$f[0][0]=1$即可

```
#include <bits/stdc++.h>
using namespace std;
int f[10][(1<<8)];
int check(int x)
{
    int tmp=0;
    for(int i=0;i<8;i++)
    {
        int y = (x>>i)&1;
        if(y==0) tmp++;
        else
        {
            if(tmp%2) return 0;
            tmp=0;
        }
```

```
    }
    return tmp%2==0;
}
int main()
{
    f[0][0]=1;
    for(int i=1;i<=8;i++)
    {
        for(int j=0;j<(1<<8);j++)
        for(int k=0;k<(1<<8);k++)
        if((j&k)==0 && check(j|k))
        {
            f[i][j] += f[i-1][k];
        }
    }
    cout<<f[8][0];
}
```

习题

序列操作

首先两个操作是可以独立操作的，我们可以先对序列 A 进行操作 2

然后对操作完的序列 A 进行操作 1 使得 $A=B$

假设进行完操作 2 后的序列 $A=\{a_{p_1},a_{p_2},\cdots,a_{p_n}\}$，那么那么操作次数就是 $p_1,p_2,\cdots,p_n$ 的逆序对个数 cnt，因为操作 2 可以相当于冒泡排序，而冒泡排序的次数就是序列的逆序对个数。因此，操作 2 的贡献为 $ans=cnt*Y$

然后进行操作 1，对答案贡献就是 $X*\sum_{i=1}^{n}|a_{p_i}-b_i|$

则总共贡献为 $ans=cnt*Y+X*\sum_{i=1}^{n}|a_{p_i}-b_i|$

那么如何求出最小 ans 呢?

首先 $n\leq 18$ 很小，可以考虑状态压缩。

设 dp_i，记 i 的二进制位为 1 的个数为 tot。

dp_i 为序列 A 的前 tot 个数已经确定了，并且前 tot 个数所对应的下标的 2 进制表示为 i 的最小花费，也就是选择序列 A 中 tot 个数（与其他数无关），对这 tot 个数进行操作使得这 tot 个数与序列 B 的前 tot 个数相同的最小花费。

状态转移:

当 $i\&(1 << j)$ 为真时:

$$dp_i = min(dp_i, dp_{i-(1<<j)} + abs(a_{j+1} - b[tot]) * X + a_j\text{的逆序对个数} * Y)$$

```
#include <iostream>
#include <vector>
#include <algorithm>
#define inf 1e18
using namespace std;
typedef long long ll;

ll N, X, Y;
ll A[20], B[20];
ll dp[1<<18];

int f(int S, int x)
{
  int ret = 0;
  for(int p = 1; p <= N; p++){
    if((S & (1<<(p-1))) == 0 && p < x) ret++;
  }
  return ret;
}

int main(void)
{
  cin >> N >> X >> Y;
  for(int i = 1; i <= N; i++) cin >> A[i];
  for(int i = 1; i <= N; i++) cin >> B[i];

  dp[0] = 0;
  for(int S = 1; S < (1<<N); S++) dp[S] = inf;
```

```
    for(int S = 0; S < (1<<N); S++){
        int sizeS = 0;
        for(int i = 1; i <= N; i++) if(S & (1<<(i-1))) sizeS++;
        for(int x = 1; x <= N; x++){
            if(S & (1<<(x-1))) continue;
            dp[S|(1<<(x-1))] = min(dp[S|(1<<(x-1))], dp[S] + abs(A[x] - B[sizeS+1]) * X + f(S, x) * Y);
        }
    }
    cout << dp[(1<<N)-1] << endl;

    return 0;
}
```

简单的题目

如果我们贸然设状态，那么一个大小为 k 的环就会被重复计算 k 次，所以需要想办法消除重复计数。

为了防止重复计数，首先我们不妨规定环上数字最小的那个顶点是环的起点。这样每个环只会被计算一次。

那我就可以设 $f[S][i]$ 表示当前这条路径走过了集合 S 中的点，且终点位于 i 的简单路径数量。因为我们规定了环上数字最小的那个顶点是环的起点，所以 $f[S][i]$ 这个状态的起点就是 S 集合中最小的那个数字。转移的时候：

- 如果从 i 扩展到的那个点比 S 中最小的那个点还小，那肯定会在其他的状态被计入答案，跳过即可；
- 如果从扩展到的那个点等于中最小的那个点，那就说明 **成环** 了，直接累加进答案中；
- 如果 i 扩展到的那个点大于 S 中最小的点，那就是一个合法的转移。

关于如何求出点集 S 中最小的点，很简单，用我们学过的 $lowbit(s) = s\& - s$ 即可。

最后，我们发现因为图是无向图，所以如果你重复经过了一条边，那答案会计多，所以答案需要减去 m。又因为图是无向图，所以顺时针经过一个环和逆时针经过一个环会被计算两次，答案要除以 2。

时间复杂度是 $O(n^2 2^n)$，因为转移的时候有很多转移不会进行，所以实际的运行时间要比这个小。

```cpp
#include <bits/stdc++.h>
using namespace std;

int n, m, g[20][20];
// 为了防止重复计数，首先我们不妨规定环上数字最小的那个顶点是环的起点。
// f[s][i]: 当前点为 i，前面经过的点的状态为 s，
// 且经过的的第一个点是经过的所有点中编号最小的一个点（避免重复计数）的简单路径的条数。
long long f[1<<20][20], ans;

int main(){
    cin >> n >> m;
    for(int i = 1, a, b; i <= m; i++) {
        cin >> a >> b;
        a--, b--;   // 结点编号从 0 开始，方便状态压缩
        g[a][b] = g[b][a] = 1;
    }
    for(int i = 0; i < n; i++)
        f[1<<i][i] = 1;
    for(int s = 0; s < (1<<n); s++) // 枚举集合 s
        for(int i = 0; i < n; i++) {  // 枚举当前终点 i
            if(!f[s][i]) continue;    // i 不在 s 中
            for(int j = 0; j < n; j++) {
                if(!g[i][j]) continue;  // 无边
                if((s&-s)>(1<<j)) continue;  // j 比 s 中起点还小
                if(1<<j == (s&-s)) {    // j 在 s 中，且是起点，成环
                    ans += f[s][i];
                }
                if(!((1<<j)&s)) {  // j 不在 s 中，且比起点大，可转移
                    f[s|(1<<j)][j] += f[s][i];
                }
            }
        }
```

```
        }
    cout << (ans-m)/2;
    return 0;
}
```