D. A Simple Task

time limit per test: 2 seconds
memory limit per test: 256 megabytes

Given a simple graph, output the number of simple cycles in it. A simple cycle is a cycle with no repeated vertices or edges.

Input

The first line of input contains two integers n and m ($1 \le n \le 19$, $0 \le m$) – respectively the number of vertices and edges of the graph. Each of the subsequent m lines contains two integers a and b, ($1 \le a, b \le n$, $a \ne b$) indicating that vertices a and b are connected by an undirected edge. There is no more than one edge connecting any pair of vertices.

Output

Output the number of cycles in the given graph.

Examples

input Copy

```
4 6
1 2
1 3
1 4
2 3
2 4
3 4
```

output Copy

```
7
```

Note

The example graph is a clique and contains four cycles of length 3 and three cycles of length 4.

→ **Problem tags**

bitmasks dp graphs *2200

No tag edit access

Problem Breakdown

Lets understand the problem first

Count the number of simple cycles in an undirected graph with:

- n ≤ 19 nodes
- m edges (simple graph: no self-loops, no multi-edges)

A simple cycle:

- must have length ≥ 3
- must have no repeated vertices
- is counted once, regardless of direction
  (i.e., cycle 1→2→3→1 = 2→3→1→2 = same)

Bitmask DP Over Subsets of Vertices

A simple cycle is defined by a set of vertices, and a Hamiltonian path within that set that loops back.

BUT we cannot brute-force all permutations (19! too large).

We instead do:

For every subset S of vertices where |S| ≥ 2,
choose the smallest vertex in S as the starting point, call it start,
DP to count paths inside S from start to some other vertex u,
and check if edge(u, start) exists.

This prevents overcounting because:

- each cycle is assigned to exactly one canonical subset S
- and exactly one canonical start = min(S)

Only consider masked paths where start is the smallest vertex, and other nodes in S are larger → avoids duplicates.

DP State Definition

Let:

- Nodes are 0 ... n-1.
- Let start be the smallest vertex in subset S.

Define:

dp[mask][v] = number of simple paths
that start at start,
visit exactly the vertices in mask,
and end at vertex v.

Where:

- mask is a subset of vertices including start
- v is the current endpoint
- Note: v ≠ start unless mask has exactly 1 element

Transitions

To move from an existing path ending at u to v, we require:

1. v not in mask
2. there is an edge (u, v)
3. v > start (to maintain canonical ordering / avoid duplicate cycles)

Transition:

```
dp[mask ∪ {v}][v] += dp[mask][u]
```

Cycle Completion

A cycle is completed when:

- The subset has size ≥ 3
- There is an edge (v, start)

So:

```
if (subset_size ≥ 3 AND edge(v, start)):
    cycles += dp[mask][v]
```

Code

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

long long dp[1 << 19][19];
bool G[19][19];

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);

    int n, m;
    cin >> n >> m;

    for (int i = 0; i < m; i++) {
        int a, b;
        cin >> a >> b;
        --a; --b;
        G[a][b] = G[b][a] = true;
    }

    long long ans = 0;

    // Iterate starting points
    for (int start = 0; start < n; start++) {

        // Clear dp
        int LIMIT = 1 << n;
        for (int mask = 0; mask < LIMIT; mask++)
            for (int i = 0; i < n; i++)
                dp[mask][i] = 0;

        dp[1 << start][start] = 1;

        // Iterate all masks that include 'start'
        for (int mask = 1 << start; mask < LIMIT; mask++) {
            if (!(mask & (1 << start))) continue;

            for (int u = 0; u < n; u++) {
                if (!(mask & (1 << u))) continue;       // u must be in mask
                long long ways = dp[mask][u];
                if (!ways) continue;

                // Try extending to a new vertex v
                for (int v = start + 1; v < n; v++) {   // ensure v > start
                    if (mask & (1 << v)) continue;      // already visited
                    if (!G[u][v]) continue;             // must be adjacent

                    int newMask = mask | (1 << v);
                    dp[newMask][v] += ways;
                }

                // Try closing a cycle (only if size >= 3)
                if (__builtin_popcount(mask) >= 3 && G[u][start]) {
                    ans += ways;
                }
            }
        }
    }

    // Each cycle counted twice (once per direction)
    cout << ans / 2 << "\n";
}
```

Even with canonical constraints, each cycle is counted:

- once in clockwise DP
- once in counter-clockwise DP

So final answer = total / 2.