2183 Deadlock Detection

A *deadlock* is defined as a set of processes entering a state where each process is waiting for response (or computing resources) from other processes. For example, in OS text books, there are deadlock detection algorithms to detect if a set of processes is *deadlocked* during runtime.

The deadlock detction problem here is to prevent deadlocks from happening before a program is executed (or even before the program is implemented). The problem is to find if there are *potential deadlocks* among a set of processes without executing the program.

Assume that processes communicate via *send* and *receive* with buffer of zero lenght; that is, when a process invokes *send*, it must block (or wait) until the message is received by the designated process via *receive*. When a process invokes a *receive* command and there is no message available, the process must block (or wait) until a message arrives. Note that it is possible for two processes to send to the same process P and then P receive the two in sequence. For example, there are three processes $P1$,$P2$, and $P3$ as follows:

```
P1:                     P2:                     P3:
do {                    do {                    do {
   msg1 = 1;              msg2 = 2;                 receive(&msg);
   send(P3,msg1);         send(P3, msg2);           if (msg == 1)
   .....                  .....                        send(P1,''ACK'');
   receive(&msg);         receive(&msg);            else
} while (1);            } while (1);                   send(P2,''ACK'');
                                                } while(1);
```

A sender invoking a *send* command must specify the receiving process. When $P1$ invokes *send*, it will wait until the message is retrieved by $P3$. The *receive* command, on the other hand, does not need to specify the process from wich the message is sent, so there is no process ID in the *receive* command. The synchronization among the processes constitutes a synchronization structure. If there is no deadlock in a system, we say the system is well-synchronized.

Researchers discovered that we can abstract every process is abstracted into a finite-state machine which is expressed using a directed graph with labels on the vertices and edges. The following game on the abstracted graphs can be used to solve a simplified version of the deadlock detection problem. You are given n, $1 \leq n \leq 6$, directed graphs $G_1, G_2, \ldots, G_n$ abstracted from processes. Let V_i and E_i be the vertices and edges of G_i, respectively. You may assume each graph has less than 20 nodes. The nodes of V_i are numbered $0, 1, \ldots, |V_i| - 1$. Each edge $e_{i,j} \in E_i$ is denoted as $s_{i,j} \to t_{i,j}$ indicatin and edge from the node $s_{i,j}$ to the node $t_{i,j}$. Each edge is labeled with an integer $label(e_{i,j})$ that can be either positive or negative, but not 0. Initially, the game starts by picking a node s_i in each graph G_i and then place a stone in it. In each of following steps, we move exactly two stones s_i and s_j using the following rules:

1. A stone can only be moved forward one edge in distance along an edge in the graph where the stone is in as a process can only advance to its next state.

2. Assume that s_i moves to s'_i, and s_j moves to s'_j. Then $label(s_i \to s'_i) + label(s_j \to s'_j) = 0$. Note that a positive edge means the sending of a message and a negative edges denotes a receiving command. In this simplified version, a send command does not specify the destination.

Hence we can describe each step of the game using an n-tuple vector $S_i = < c_{i,1}, c_{i,2}, \ldots, c_{i,n} >$, where $c_{i,j}$ is a node in the graph G_j. The game ends when it is not possible to make any more moves. The steps that cannot be further advanced are called *deadlocked* steps.

An example is given below for a set of games without deadlocked steps and a set of games with a deadlocked step. In Figure 1, there are three graphs.In each graph, we initially put a stone in the nodes as shown in the figure. Following the rules described above, for example, we camn move the stones of G1 and G3 to their next nodes by following the edges labeled with '`111`' and '`-111`', respectively. Continuously, you can allways find two stones to move without any deadlock. We can verify that this example has no deadlocks states. Figure 2(a) is an example that has deadlocks. Suppose we initially put the stones as shown in Figure 2(a). Suppose we move the stone of G1 and G3 to become a state in (b). In (b), we can not make any further movements, thus is deadlocked. It can be veritied that the number of all possible deadlock steps in this example is 1.

Given n, the set of graphs and their starting nodes, yopur task is to find the number of all the possible deadlocked steps from the starting step.

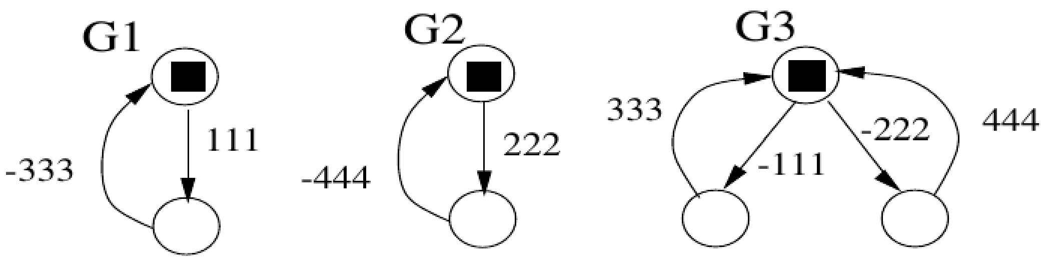

Figure 1: The game has no deadlock state.

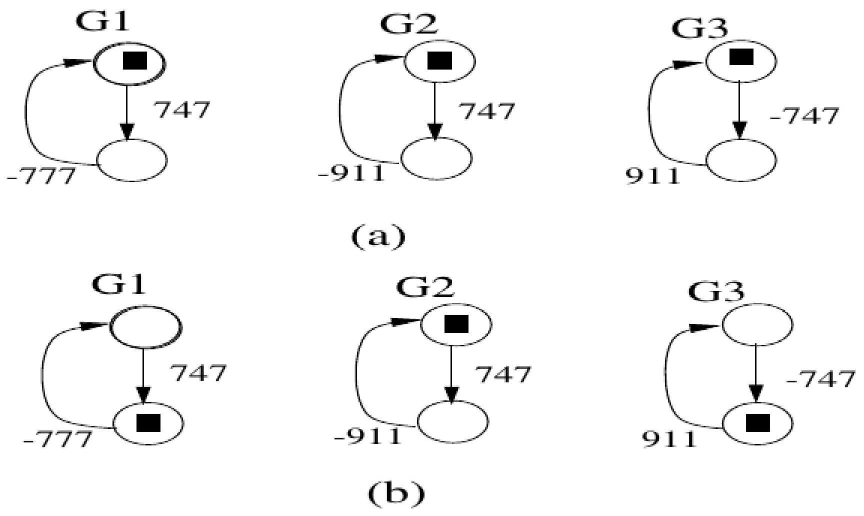

Figure 2: The game has 1 deadlock state.

Input

The input contains multiple yest cases. The first line contains the number of graphs n and the followed by the data of the n graphs. Each graph starts with three integers 's t i', where s is the number of nodes, t is the number of transactions, and i is the starting node. Following the line of the three integers s, t and i is t lines of directed edges in the form of 'a b c', wich represents and edge with label b from node a to node c. A line with three '`0`'s means the end of a test case. When the number of graphs is '`0` ($n = 0$), it is the end of all test cases.

Output

For each test case, output the total number of deadlock steps in a line.

Sample Input

```
3
2 2 0
0 111 1
1 -333 0
2 2 0
0 222 1
1 -444 0
3 4 0
0 -111 1
0 -222 2
1 333 0
2 444 0
0 0 0
3
2 2 0
0 747 1
1 -777 0
2 2 0
0 747 1
1 -911 0
2 2 0
0 -747 1
1 911 0
0 0 0
0
```

Sample Output

```
0
1
```