2 SAT

2-SAT is a special case of boolean satisfiability. Boolean satisfiability or just *SAT* determines whether we can give values (**TRUE** or **FALSE** only) to each boolean variable in such a way that the value of the formula become **TRUE** or not. If we can do so, we call formula *satisfiable*, otherwise we call it *unsatisfiable*. Look at the example below:

$f = A \wedge \neg B$, is *satisfiable*, cause A = **TRUE** and B = **FALSE** makes it **TRUE**.

but $g = A \wedge \neg A$, is *unsatisfiable*, look at this table:

A	$\neg A$	$A \wedge \neg A$
TRUE	**FALSE**	**FALSE**
FALSE	**TRUE**	**FALSE**

As you can see g is *unsatisfiable* cause whatever values of its boolean variables are, g is **FALSE**.

Note: $\neg$ in $\neg X$ is boolean *not* operation. $\wedge$ in $X \wedge Y$ is boolean *and* operation and finally $\vee$ in $X \vee Y$ is boolean *or* operation.

SAT is a NP-Complete problem, though we can solve *1-SAT* and *2-SAT* problems in a polynomial time.

1-SAT

Note: This doesn't really exist, I define it cause it help understanding *2-SAT*.

Consider $f = x_1 \wedge x_2 \wedge \ldots \wedge x_n$.

Problem: Is f *satisfiable*?

Solution: Well *1-SAT* is an easy problem, if there aren't both of x_i and $\neg x_i$ in f, then f is *satisfiable*, otherwise it's not.

2-SAT

Consider $f = (x_1 \vee y_1) \wedge (x_2 \vee y_2) \wedge \ldots \wedge (x_n \vee y_n)$.

Problem: Is f *satisfiable*?

But how to solve this problem? $x_i \vee y_i$ and $\neg x_i \Rightarrow y_i$ and $\neg y_i \Rightarrow x_i$ are all equivalent. So we convert each of $(x_i \vee y_i)$ s into those two statements.

Now consider a graph with $2n$ vertices; For each of $(x_i \vee y_i)$ s

we add two directed edges

1. From $\neg x_i$ to y_i
2. From $\neg y_i$ to x_i

f is not *satisfiable* if both $\neg x_i$ and x_i are in the same SCC (Strongly Connected Component) (Why?) Checking this can be done with a simple Kosaraju's Algorithm.

Assume that f is *satisfiable*. Now we want to give values to each variable in order to satisfy f. It can be done with a topological sort of vertices of the graph we made. If $\neg x_i$ is after x_i in topological sort, x_i should be **FALSE**. It should be **TRUE** otherwise.

Some problems:

- SPOJ — BUGLIFE
- SPOJ — TORNJEVI
- UVa — Manhattan
- UVa — Wedding
- CF — The Road to Berland is Paved With Good Intentions
- CF — Ring Road 2
- CF — TROY Query
- CEOI — Birthday party — Solution

CODE: //Solution of SPOJ BUGLIFE

```
#define lim          2005  //number of
nodes(yes/no nodes)
//0 based
vector<int> adj[2*lim]; //2*lim for true and false
argument(only adj should be cleared)
int col[2*lim],low[2*lim],tim[2*lim],timer;
int group_id[2*lim],components;//components=number
of components, group_id = which node belongs to
which node
bool ans[lim]; //boolean assignment ans
stack<int>S;

void scc(int u) {
    int i,v,tem;
        col[u]=1;
    low[u]=tim[u]=timer++;
    S.push(u);
    for(int i=0; i<adj[u].size(); i++) {
        v=adj[u][i];
        if(col[v]==1)
            low[u]=min(low[u],tim[v]);
        else if(col[v]==0) {
            scc(v);
            low[u]=min(low[u],low[v]);
        }
    }

    //SCC checking...
    if(low[u]==tim[u]) {
        do {
            tem=S.top();
            S.pop();
            group_id[tem]=components;
            col[tem]=2; //Completed...
        } while(tem!=u);
        components++;
    }
}

int TarjanSCC(int n) { //n=nodes (some change may
be required here)
    int i;
    timer=components=0;
    clr(col,0);
    while(!S.empty()) S.pop();
    for(int i=0; i<n; i++)
        if(col[i]==0) scc(i);
    return components;
}

//double nodes needed normally
bool TwoSAT(int n) { //n=nodes (some change may be
required here)
    TarjanSCC(n);
    int i;
    for(i=0; i<n; i+=2) {
        if(group_id[i]==group_id[i+1])
            return false;
        if(group_id[i]<group_id[i+1]) //Checking
who is lower in Topological sort
            ans[i>>1]=true;
        else ans[i>>1]=false;
    }
    return true;
}

void add(int ina,int inb) {
    adj[ina].pb(inb);
}

int complement(int n) {
    if(n%2) return n-1;
    return n+1;
}

void initialize(int n) {
    for(int i=0; i<n; i++)
        adj[i].clear();
}

int main() {
    int T;
    scanf("%d", &T);
    for (int caseNo = 0; caseNo < T; caseNo++) {
        int N,M;
        scanf("%d %d", &N, &M);
        N <<= 1;
        initialize(N+5);
        while (M--) {
            int b1, b2;
            scanf("%d %d", &b1, &b2);
            b1--, b2--;
            b1<<=1; //! b1
            b2<<=1; //! b2
            adj[b1].push_back(b2^1); //!b1 v b2
            adj[b2].push_back(b1^1); //!b2 v b1
            /// As, its undirected graph
            adj[b1^1].push_back(b2);
            adj[b2^1].push_back(b1);
        }
        printf("Scenario #%d:\n", caseNo + 1);
        if (!TwoSAT(N))
            printf("Suspicious bugs found!\n");
        else
            printf("No suspicious bugs found!\n");
    }
    return 0;
}
```