目录

有向图博弈

例 1

/*
题意:
n 点 m 边有向图，不能走者负。
A 偏好:平局>赢>输
B 偏好:赢>输>平局
分两行，分别输出 n 个位置 A/B 先走时,先手按上述偏好最终的结果

做法:
1、按此通用模版，初始全是平局，从 deg 为 0 的开始转移。按上述偏好顺序转移即可。
2、A 能必平局一定平，B 能必赢一定赢。A 不能必定平局时，B 一定阻止其平，因为是其最坏结果。
B 不能必赢，A 一定阻止其赢，亦然。
故 A 不能必平局，则必赢。B 不能必赢，则必输。
拓扑排序标记点。
*/

```
vector<int>edg[MAX];
const int lim=2e7;
int n,m,top,tail;
int
dp[MAX][2],que[(int)2e7+5],deg[MAX],cnt[
MAX][2][3];
inline int mk_st(int lo,int id,int pre_st,int
now_st){
	return
now_st|(pre_st<<2)|(id<<4)|(lo<<5);
}
inline void ad(int lo,int id,int st){
	int tem=dp[lo][id];
	if(tem!=st){
		++top;
		if(top>lim)top=0;
		que[top]=mk_st(lo,id,tem,st);
		dp[lo][id]=st;
	}
}
int main(){
	freopen("game.in","r",stdin);
	freopen("game.out","w",stdout);
	read(n);read(m);
	for(int i=1;i<=m;i++){
		int u,v;read(u);read(v);
		++deg[u];edg[v].pb(u);
	}
	for(int i=1;i<=n;i++){
		dp[i][0]=dp[i][1]=1;
		cnt[i][0][1]=cnt[i][1][1]=deg[i];
	}
	for(int i=1;i<=n;i++)
		if(!deg[i])
			ad(i,0,2),ad(i,1,0);
	while(tail!=top){
		++tail;
		if(tail>lim)tail=0;
		int tem=que[tail];
		int
lo=tem>>5,id=(tem>>4)&1,pre_st=(tem>
>2)&3,now_st=tem&3;
		int pid=id^1;///前一个人
		for(int u:edg[lo]){
			--cnt[u][pid][pre_st];
			++cnt[u][pid][now_st];
			int val=0;
```

```
			if(pid==0){
				if(cnt[u][pid][0])val=0;
				else
val=(cnt[u][pid][2]?2:0);
			}
			else{
				if(cnt[u][pid][1])val=1;
				else
val=(cnt[u][pid][2]?2:0);
			}
			if(val!=dp[u][pid])ad(u,pid,val);
		}
	}
	for(int i=1;i<=n;i++)

printf("%c",dp[i][1]==2?'W':(dp[i][1]==0?'L':'D'));
	puts("");
	for(int i=1;i<=n;i++)

printf("%c",dp[i][0]==0?'W':(dp[i][0]==2?'L':'D'));
	return 0;
}
```

例 2

```
/*
题意：
两支队伍在一张 n 个点 m 条边的有向图上玩游戏，6 个人
轮流移动棋子行动一步，不能移动的人所属队伍输。
正常人希望赢，不能赢则希望平局；而演员希望输，不能输
则希望平局。
对于每个点计算棋子放在该点开始游戏时最后的游戏结果。

做法：
状态：fi;j 表示棋子位于 i，目前轮到第 j 个人行动时最终的
游戏结果。
定义-1 表示 A 胜，0 表示平局，1 表示 B 胜，则每个状
态的转移是 min 或者 max。
一开始假设所有状态的值都是 0，并维护出 cnt 数组表示每
个状态后继状态中有几个-1; 0; 1，用于 O(1)计算一个状态的值。
对于所有不能行动的状态，重新设定它的状态值，并加入队列。
用 SPFA 的方式依次松弛每个值发生改变的状态的前驱状态，并更新 cnt 数组。
每个状态的值只会在-1; 0; 1 之间切换常数次。
时间复杂度 O(n + m)。
（实现中上述-1,0,1 对应于 0,1,2）
*/
const double eps=1e-8;
const int lim=(1<<23);
int t,n,m,top,pos;
int
deg[MAX],who[MAX],que[(int)2e7+5],bel[MAX];
vector<int>edg[MAX];
char a[20];
int dp[MAX][8],cnt[MAX][8][4];
inline int mk_st(int lo,int id,int tem,int st){
	return st|(tem<<2)|(id<<4)|(lo<<7);
}
inline void ad(int lo,int id,int st){
	int tem=dp[lo][id];///之前转移的状态
	if(st!=tem){
		++top;
		if(top>lim)top=0;
		que[top]=mk_st(lo,id,tem,st);
		dp[lo][id]=st;
	}

}
int main(){
	read(t);
	while(t--){
		read(n);read(m);
		pos=top=0;
		for(int u,v,i=1;i<=m;i++){
			read(u);read(v);
			++deg[u];
```

```
			edg[v].pb(u);
		}
		scanf("%s",a);///sequence
		for(int i=0;i<6;i++)

who[i]=(a[i]=='A'),bel[i]=who[i];
		scanf("%s",a);///actor
		for(int i=0;i<6;i++)
			who[i]^=(a[i]=='1');
		for(int i=1;i<=n;i++){
			for(int j=0;j<6;j++){
				dp[i][j]=1;
				for(int
k=0;k<3;k++)cnt[i][j][k]=0;
				cnt[i][j][1]=deg[i];
			}
		}
		for(int i=1;i<=n;i++)
			if(!deg[i])
				for(int j=0;j<6;j++)
					ad(i,j,2*bel[j]);
		while(pos!=top){
			++pos;if(pos>lim)pos=0;
			int st=que[pos];
			int
lo=st>>7,id=(st>>4)&7,pre=(st>>2)&3,no
w=st&3;
			int pid=id-1;///前一个人
			if(pid<0)pid+=6;
			for(int u:edg[lo]){
				--cnt[u][pid][pre];
				++cnt[u][pid][now];
				int val=0;
				if(who[pid]){///是 A

if(cnt[u][pid][0])val=0;///能转移到 0 状态
					else
val=(cnt[u][pid][1]?1:2);
				}
				else{

if(cnt[u][pid][2])val=2;
					else
val=(cnt[u][pid][1]?1:0);
				}
				if(val!=dp[u][pid])
					ad(u,pid,val);
			}
		}
		for(int i=1;i<=n;i++){
			char
ans=(dp[i][0]==0?'A':(dp[i][0]==1?'D':'B'));
			printf("%c",ans);
		}
		puts("");
		for(int i=1;i<=n;i++){
			deg[i]=0;
			edg[i].clear();
		}
	}
	return 0;
}
```

Multi-SG

/*题意

给出 n 堆石子，每次可以选择将大于某个数 f 一堆平均分成多个堆，最后不能操作的失败。

*/

/*做法

考虑每堆是单独的子游戏，互不影响，所以我只要对每堆做完，然后把每堆的 SG 给 XOR 起来就可以了。

而每堆的结果也只跟堆的石子数量有关，那么我就可以对于每种石子数量做好了。

而对于一种特定的石子数量的 SG，我肯定是把他所有的后继状态的 SG 给 XOR 起来，那么枚举一下分成了多少堆，就可以直接算了。复杂度是 O(石子数^2)。

进一步观察，发现若干个数的分出来的较小的那个块是相同的，而且这一段最多最多也只能贡献 2 种不同的 SG 值，所以就可以每块丢到一起做，算算 2 种不同的 SG 值，就可以了。

数论分块 O(n^3/2)

```
*/
const int MAXN = 100011;
int n,F,SG[MAXN],ans,Tim,vis[MAXN];

inline int getint(){
    int w=0,q=0; char c=getchar();
while((c<'0'||c>'9') && c!='-') c=getchar();
    if(c=='-') q=1,c=getchar(); while
(c>='0'&&c<='9') w=w*10+c-
'0',c=getchar(); return q?-w:w;
}

inline int calc(int x){
    Tim++; int xiao,num,num2,now,nex;
    for(int i=2;i<=x;i=nex+1) {//枚举分成
的堆的数目
        xiao=x/i;//数量较小的堆有多少个
石子

        num2=x%i;//注意计算方法
        num=i-num2;
        now=0;
        //只与 i 奇偶性有关，那么只可能
贡献两种 SG 值
        if(num&1) now^=SG[xiao];
        if(num2&1) now^=SG[xiao+1];
        vis[now]=Tim;

        nex=min(x/xiao,x);
        if(i+1<=nex) {
            now=0;
            num2=x%(i+1);
            num=(i+1)-num2;
            if(num&1) now^=SG[xiao];
            if(num2&1)
now^=SG[xiao+1];
            vis[now]=Tim;
        }
    }
    int mex=0;
    for(;vis[mex]==Tim;mex++) ;
    return mex;
}
inline void init(){
    for(int i=0;i<F;i++) SG[i]=0;

    for(int i=F;i<=100000;i++)
        SG[i]=calc(i);
}

inline void work(){
    int T=getint(); F=getint(); int x;
    init();
    while(T--) {
        ans=0; n=getint();
        for(int i=1;i<=n;i++)
x=getint(),ans^=SG[x];
        if(T!=0) printf("%d ",ans==0?0:1);//
不能直接输出 SG 值啊...
        else printf("%d",ans==0?0:1);
    }
}

int main()
{
    work();
    return 0;
}
```

Hackenbush 博弈

/*
题意：
给定一棵 n 个节点的带权有根树，两人轮流操作，每次操作使某条边的边权减一，
若边权减至 0，则删去改边，并删去不含根的连通块，
判断先手是否必胜，并输出先手若要胜利，所有第一步可能的决策。

n,w <=1e6

做法:
考虑将题目转化为有根无向图删边游戏(Hackenbush),

每条树边的边权代表了无向图上两端点之间的边数量。
注意到对于边权大于 1 时会形成环, 按奇偶性缩环（无向图删边游戏的操作）即可。

在计算出 SG 值后, 每一步操作会改动一条到根路径上的所有 SG 值,
自根向下确定每个子树如果修改那么需要修改得到的 SG 值是多少, 即可检查某一种操作是否能使得先手必胜。

```
*/
vector<int>edg[MAX];
int fa[MAX],w[MAX],sg[MAX];
int T,n,cnt;
int an[MAX];
void dfs(int now){///算原图的 sg
	sg[now]=0;
	for(int v:edg[now]){
		dfs(v);
		if(w[v]==1)sg[now]^=sg[v]+1;
		else{
			sg[now]^=sg[v];
			if(w[v]&1)sg[now]^=1;
		}
	}
}
void dfs(int now,int target){
	if(target<0)return;
	for(int v:edg[now]){
		if(w[v]==1){///1->0

if((sg[now]^(sg[v]+1))==target)an[++cnt]=v;
		}
		else if(w[v]==2){///2->1
			if((sg[now]^(sg[v])^(sg[v]+1))==target)an[++cnt]=v;
		}
		else{/// ? ->   >1 的状态

if((sg[now]^1)==target)an[++cnt]=v;
		}
		/// 去掉其余子树的影响更新 target

if(w[v]==1)dfs(v,(target^(sg[now]^(sg[v]+1))) -1);
		else dfs(v,target^(sg[now]^sg[v]));
	}

}
int main(){
	read(T);
	while(T--){
		read(n);
		for(int i=1;i<=n;i++)edg[i].clear();
		for(int u,i=2;i<=n;i++){
			read(u);read(w[i]);
			edg[u].pb(i);
		}
		dfs(1);
		if(!sg[1])puts("0\n");
		else{
			cnt=0;
			dfs(1,0);
			sort(an+1,an+1+cnt);
			printf("%d\n",cnt);
			for(int i=1;i<=cnt;i++)printf("%d%c",an[i]," \n"[i==cnt]);
		}
	}
	return 0;
}
```

Anti-SG

antiSG 的定义是这样的:

1、决策集合为空的游戏者赢
2、其余规则与 SG 游戏相同

SJ 定理(亦即 csy 模版里的 Anti-SG Game)
建立在: 规定当局面中所有的单一游戏的 SG 值为 0 时，游戏结束。

（也可以替换成：当局面中所有的单一游戏的 SG 值为 0 时，存在一个单一游戏它的 SG 函数能通过一次操作变为 1）
“因为笔者发现这样可以出题，我们可以将题目模型设成这样：游戏中存在一个按钮，游戏双方都可以触动按钮，当其中一个人触动按钮时，触动按钮的人每次必须移动对方上次移动的棋子。如果触动按钮的人能保证他能够使得对方无路可走，那么他同样获胜！”

Every-SG

定义：

Every-SG 游戏规定，对于还没有结束的单一游戏，游戏者必须对该游戏进行一步决策；
Every-SG 游戏的其他规则与普通 SG 游戏相同

在通过拓扑关系计算某一个状态点的 SG 函数时（事实上，我们只需要计算该状态点的 SG 值是否为 0），对于 SG 值为 0 的点，我们需要知道最快几步能将游戏带入终止状态，对于 SG 值不为 0 的点，我们需要知道最慢几步游戏会被带入终止状态，我们用函数来表示这个值。

$$step(v)=\begin{cases}0 & v\text{为终止状态}\\ \max\left(step(u)\right)+1 & SG(v)>0\wedge u\text{为}v\text{的后继状态}\wedge SG(u)=0\\ \min\left(step(u)\right)+1 & SG(v)=0\wedge u\text{为}v\text{的后继状态}\end{cases}$$

[定理] 对于 Every-SG 游戏先手必胜当且仅当单一游戏中最大的为奇数。

斐波那契博弈

有一堆石子，两个顶尖聪明的人玩游戏，先取者可以取走任意多个，但不能全取完，以后每人取的石子数不能超过上个人的两倍
结论：先手必败，当且仅当石子数为斐波那契数

Nim 积

我们对于一些二维 Nim 游戏（好像更高维也行），可以拆分成两维单独的 Nim 然后求 Nim 积。

定义为

$$x\otimes y=\mathrm{mex}\{(a\otimes b)\oplus(a\otimes y)\oplus(x\otimes b),0\le a<x,0\le b<y\}$$

其中 $\otimes$ 定义为 NimNim 积，$\oplus$ 定义为异或。

运算性质：

$$x\otimes 0=0\otimes x=0$$

$$x\otimes 1=1\otimes x=x$$

$$x\otimes y=y\otimes x$$

即 0 与所有数做 Nim 积仍然为 0， 1 仍然是单位元，并且满足交换律。

不会证明的两个结论：

$$x\otimes (y\otimes z) =(x\otimes y)\otimes z$$

$$x\otimes (y\oplus z) =(x\otimes y)\oplus (x\otimes z)$$

就是说满足乘法交换律，和乘法分配率（把 $\otimes$看作 × 以及 $\oplus$ 看做 + ）。

2 费马数的一些运算性质

经过数学家的艰苦卓绝的努力，我们有两个十分强大的运算法则。

定义 Fermat 2-power 为 2^{2^n} ，其中 $n \in \mathbb{N}$ ，设其为 a 。

1. 一个 Fermat 2-power 与任意小于它的数的 Nim 积为一般意义下乘法的积，即 $a \otimes x = a \times x\ (x < a)$ 。
2. 一个 Fermat 2-power 与自己的 Nim 积为自己的 $\frac{3}{2}$ 倍，即 $a \otimes a = \frac{3}{2}a = a \oplus \frac{a}{2}$ 。

3 算法解决

注意暴力求 Nim 积是 $\mathcal{O}((xy)^2)$ 的，我们可以利用一些性质在 $\mathcal{O}(\log x \log y)$ 的时间内解决。

对于任意 x, y 的解法

我们设 $f(x,y) = x \otimes y$ ，我们特判 x or $y = 0, 1$ 的情况后，可以考虑拆出 x, y 的每个二进制位单独算。

就是设 $g(x,y) = 2^x \otimes 2^y$ ，那么 $f(x,y) = \oplus_{x' \in x, y' \in y} g(x', y')$ 。

对于 $2^x \otimes 2^y$ 的解法

> 这一段是 zhou888 教我的，太恐怖啦 %%%

那么我们问题就转化为求 $g(x,y)$ 了。

我们考虑把 x, y 的二进制位拆出来，变成一个个费马数，然后利用性质处理。

$$2^x \otimes 2^y = (\otimes_{x' \in x} 2^{2^{x'}}) \otimes (\otimes_{y' \in y} 2^{2^{y'}})$$

考虑 **从高到低** 依次考虑 x, y 的每一位，如果这位都为 0 我们显然可以忽略。

x and y 的情况

假设全都为 1 那么对于这一位 2^u 我们设 $M = 2^{2^u}, A = 2^{x-2^u}, B = 2^{y-2^u}$ ，那么有 $A, B < M$ 。

那么我们的答案其实就是 $ans = (M \otimes A) \otimes (M \otimes B)$ （注意费马数的 $\times$ 和 $\otimes$ 是一样的）即 $(M \otimes M) \otimes (A \otimes B)$ ，化简一下答案其实就是 $\frac{3}{2} M \otimes (A \otimes B)$ 。

那么此时我们把 $2^x, 2^y$ 都去掉最高的一位 u 变成 A, B ，继续向低位递归。

x xor y 的情况

假设一个为 1 一个为 0 ，同样我们设这位为 2^u ，假设 x 此位为 1 ，那么有 $M = 2^{2^u}, A = 2^{x-2^u}, B = 2^y$ 。

那么答案的形式为 $ans = (M \otimes A) \otimes B$ 也就是 $M \otimes (A \otimes B)$ 。类似的，我们去掉最高位，然后不断向下推。

讨论完上面两种情况，我们可以写一下表达式。

我们显然可以利用交换律把 x xor y 和 x and y 的情况分开。

$$
\begin{aligned}
2^x \otimes 2^y &= (\otimes_{i \in \{x \text{ xor } y\}} 2^{2^i}) \oplus (\otimes_{i \in \{x \text{ and } y\}} \frac{3}{2} 2^{2^i}) \\
&= (\prod_{i \in \{x \text{ xor } y\}} 2^{2^i}) \otimes (\otimes_{i \in \{x \text{ and } y\}} \frac{3}{2} 2^{2^i})
\end{aligned}
$$

那么对于前者可以直接算，后面利用 f 递归算就行了。

复杂度不难发现只会遍历两个所有二进制位，也就是单次为 $\mathcal{O}(\log^2 x)$ 。

```
/*
在一个二维平面中，有 n 个灯亮着并告诉你坐标，每回合需要找到一个矩形，
这个矩形 (x,y) 坐标最大的那个角落的点必须是亮着的灯，然后我们把四个角落的灯状态反转，不能操作为败。

Turning Corners 是裸的二维 Nim 问题，直接上模板就好了。

复杂度是 O(Tnlogxlogy) 的。
*/

#define Resolve(i, x) for (int u = (x), i = 0; (1ll << i) <= u; ++ i) if (u >> i & 1)

ll f(ll x, ll y);

ll g(int x, int y) {
	if (!x || !y) return 1ll << (x | y);
	if (~ tab[x][y]) return tab[x][y];
	ll res = 1;
	Resolve(i, x ^ y) res <<= (1 << i);
	Resolve(i, x & y) res = f(res, 3ll << ((1 << i) - 1));
	return tab[x][y] = res;
}

ll f(ll x, ll y) {
	if (!x || !y) return x | y;
	if (x == 1 || y == 1) return max(x, y);
	ll res = 0;
	Resolve(i, x) Resolve(j, y) res ^= g(i, j);
	return res;
}
```