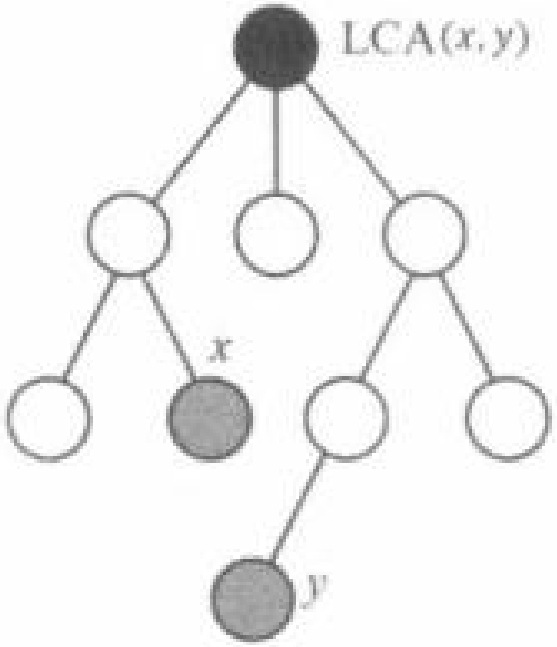
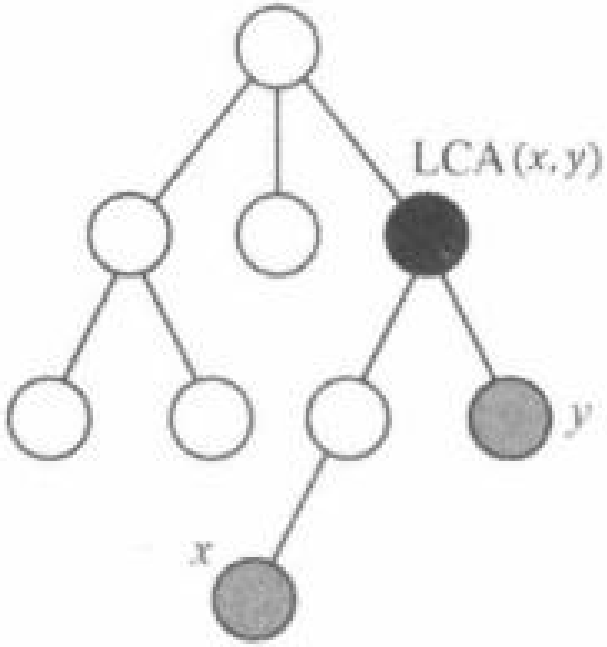
21级实验室暑假第四讲

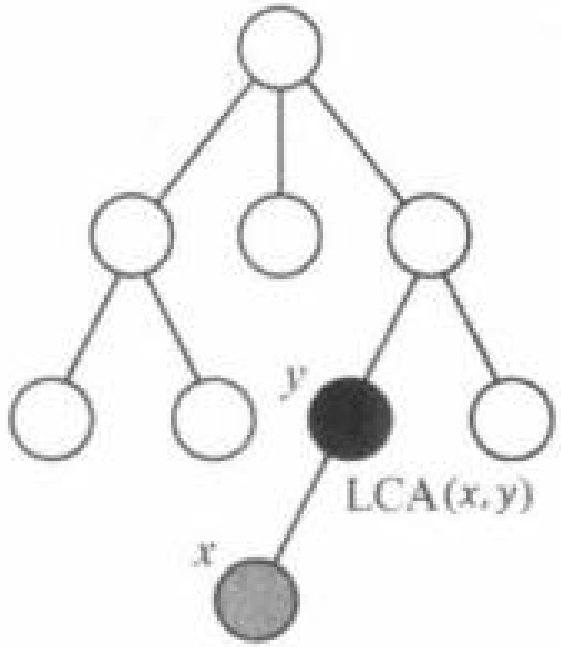
目录

LCA（最近公共祖先）

- 给定一棵有根树，若结点z既是结点x的祖先，也是结点y的祖先，则称为z是x的公共祖先。在x,y的所有公共祖先中，深度最大的一个称为x,y的**最近公共祖先(Lowest Common Ancestors)**，即为LCA(x,y)







- LCA（x,y）是x到根的路径与y到根的路径的**交汇点**。它也是x与y之间的路径深度最小的结点。

LCA（最近公共祖先）

求最近公共祖先的方法通常有三种：

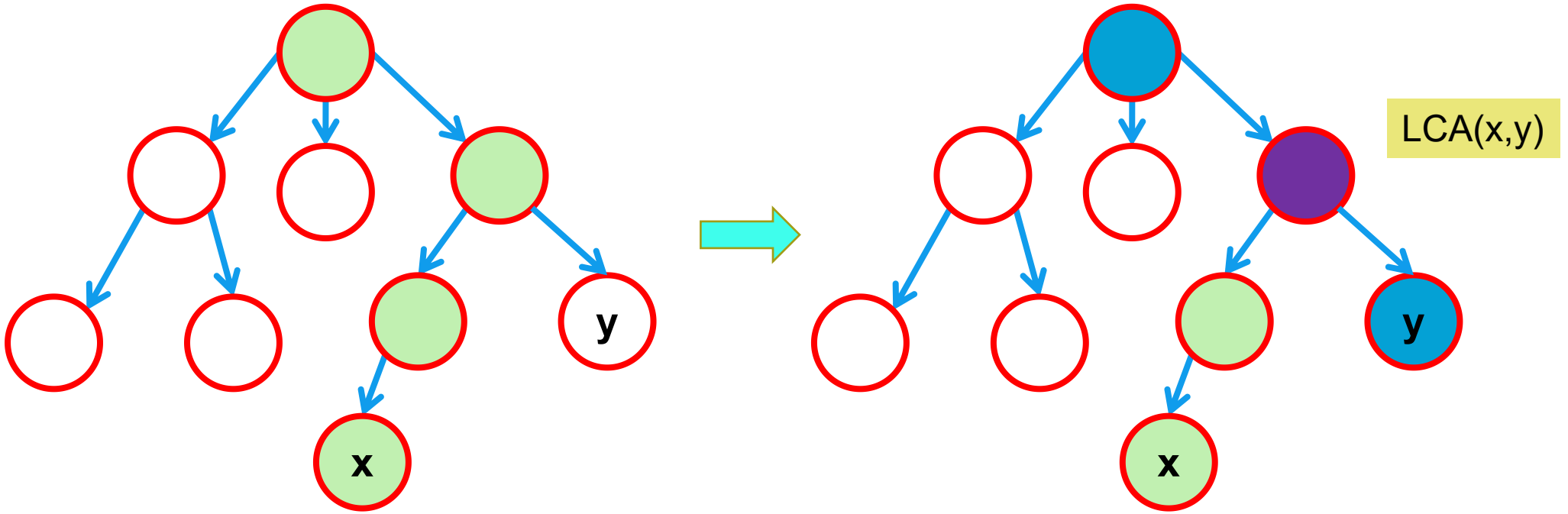
✓向上标记法
✓树上倍增法
✓LCA的Tarjan算法（与强连通分量的Tarjan算法不同）
注: 前两种基于**在线算法**，而第三种基于**离线算法**

在计算机科学中，一个**在线**算法是指它可以以**序列化的方式一个个的处理输入**，也就是说在开始时并不需要已经知道所有的输入。相对的，对于一个**离线**算法，在开始时就需要知道**问题的所有输入数据**，而且**在解决一个问题后就要立即输出结果**。

LCA（最近公共祖先）

✓向上标记法

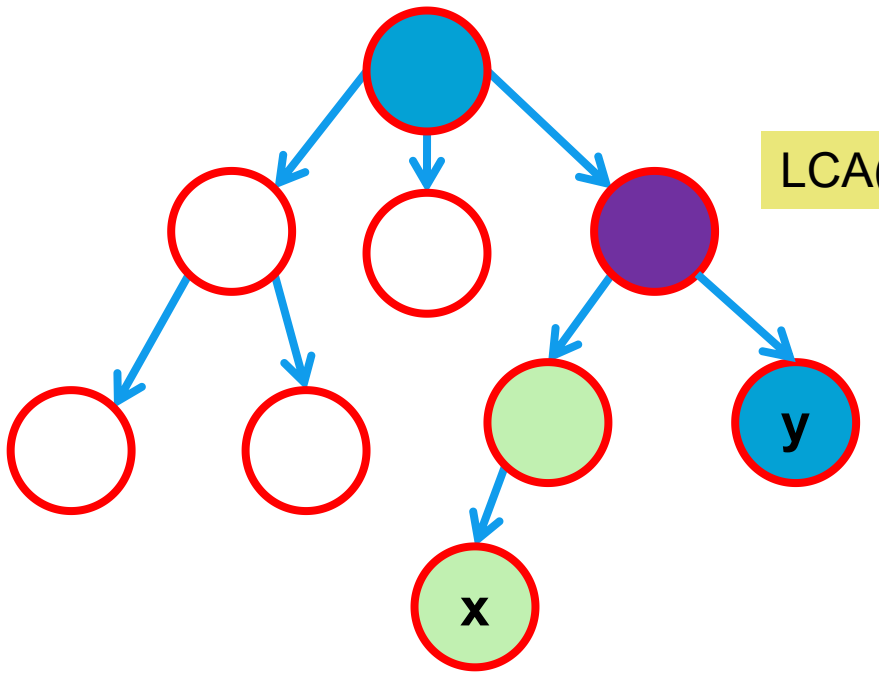
◆首先从x向上走到根结点，并标记所有经过的结点(包括结点x)
◆然后再从y向上走到根结点，当第一次遇到已经标记过的结点时，就找到LCA(x,y)



LCA（最近公共祖先）

✓ 树上倍增法

给出 n 个点的一棵树，多次询问两点之间的最短距离。



已知是一棵树，那么任意两点之间的路径是唯一的，因为最短路径就是这条路径的距离。

设dist数组表示每个点到根结点的距离，故任意两点的距离公式为:

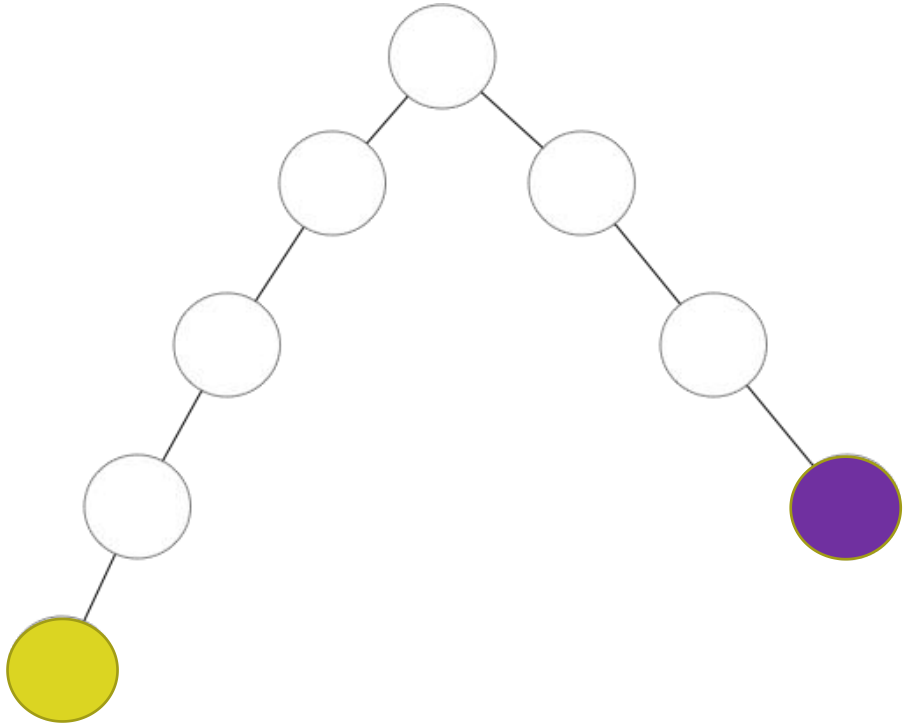
$$dist[x]+dist[y]-2\times dist[lca(x,y)]$$

LCA（最近公共祖先）

✓向上标记法

◆对于每一次询问，向上标记法的时间复杂度最坏是O(N)

当树的形状是下面这样的时候，需要遍历所有树上的结点，时间复杂度为O(N)



LCA（最近公共祖先）

✓树上倍增法

树上倍增法是一种很重要的算法。除了求 LCA 外，它在很多问题中都有广泛的应用。

设**fa[x,k]表示x的2^k辈祖先($0\le k\le \log_2 N$,N为结点个数)**，即从x向根结点走2^k步所到达的结点。
特别地，如果该结点不存在，则令fa[x,k]=0，其中fa[x,0]就是x的父结点。
除此之外，**fa[x,k]=fa[fa[x,k-1],k-1]**

此处为什么fa[x,k]=fa[fa[x,k-1],k-1]呢?

$2^k = 2^{k-1} + 2^{k-1}$(即先走2^{k-1}步，再走2^{k-1}步就能到达2^k步)

LCA（最近公共祖先）

✓ 树上倍增法

这类似于一个动态规划的过程，“阶段”就是**结点的深度**。因此，我们可以对树进行广度优先遍历(bfs)，按照层次顺序，在结点入队之前，计算它在fa数组中对应的值。

fa[x,k]=fa[fa[x,k-1],k-1]

以上部分是预处理，时间复杂度为O(nlogn)，之后可以多次对不同的x，y计算LCA，每次询问的时间复杂度为O(logn)。

LCA（最近公共祖先）

✓树上倍增法

关于**结点的深度的问题**

已知，结点数为N的**完全二叉树**的深度 k 为 ： $k = \log_2 N + 1$

N	1	2~3	4~7	8~15	16~31
$\log_2 N+1$	1	2	3	4	5

其中头文件math.h中只有底数为e或者10的对数函数，无底数为2的对数函数
double log (double); //以e为底的对数
double log10 (double);//以10为底的对数

对数换底公式：$\log_a b = \dfrac{\log_c b}{\log_c a}$　　故：$k = \log_2 N + 1 = \dfrac{\ln N}{\ln 2} + 1$

```
int k=(int)(log(N)/log(2))+1;
```

LCA(最近公共祖先)

✓树上倍增法

```
int head[MAXN],ed[2*MAXN],nex[2*MAXN],val[2*MAXN],idx;

int depth[MAXN],fa[MAXN][50],dist[MAXN];
//dist[i]表示i到root的距离

void init(int n){//初始化
    idx=0;
    for(int i=1;i<=n;i++)
        head[i]=-1;
}

void add(int a,int b,int c){//链式前向星add操作
    ed[idx]=b;
    val[idx]=c;
    nex[idx]=head[a];
    head[a]=idx++;
}
```

LCA(最近公共祖先)

✓树上倍增法

```
void bfs(int root){
    queue<int> q;
    memset(depth,0x3f,sizeof(depth));
    q.push(root);
    dist[root]=0;
    depth[0]=0,depth[root]=1;
    //令0为哨兵,root的父结点是0,0的深度为0,root的深度为1
    //当从某个点跳出根结点root,那么祖宗点就是0
    //跳出去以后深度为0,便不会执行跳的操作
    while(q.size()>0){
        int temp=q.front();
        q.pop();
        for(int i=head[temp];i!=-1;i=nex[i]){
            int j=ed[i];
            if(depth[j]>depth[temp]+1){//未更新
                depth[j]=depth[temp]+1;
                dist[j]=dist[temp]+val[i];
                q.push(j);
                fa[j][0]=temp;
                for(int l=1;l<=k;l++)
                    fa[j][l]=fa[fa[j][l-1]][l-1];
            }
        }
    }
}
```

LCA（最近公共祖先）

✓树上倍增法

基于fa数组计算LCA(x,y)分为以下几步：

1. 设depth[x]表示x的深度，不妨设depth[x]≥depth[y],否则可交换x,y。
2. 用**二进制拆分思想**，把x向上调整到与y同一深度。

2^0	2^1	2^2	2^3	2^4
1	2	4	8	16

假设拼凑出11$(1011)_2$，从大到小依次枚举2^k，如果2^k>当前数，跳过，否则更新**当前数=当前数-2^k**，直到所有的k遍历完

LCA（最近公共祖先）

✓树上倍增法

2. 用**二进制拆分思想**，把x向上调整到与y同一深度。

2^0	2^1	2^2	2^3	2^4
1	2	4	8	16

假设拼凑出11$(1011)_2$，从大到小依次枚举2^k，如果2^k>当前数，跳过，否则更新**当前数=当前数-2^k**，直到所有的k遍历完

1. 选取2^4=16，此时16>11，跳过
2. 选取2^3=8，此时8≤11，更新11-8=3
3. 选取2^2=4，此时4>3，跳过
4. 选取2^1=2，此时2≤3，更新3-2=1
5. 选取2^0=1，此时1≤1，更新1-1=0

故根据步骤中是否更新(更新为1，不更新为0)，11可转换为二进制$(01011)_2$

LCA（最近公共祖先）

✓树上倍增法

2. 用**二进制拆分思想**，把x向上调整到与y同一深度。

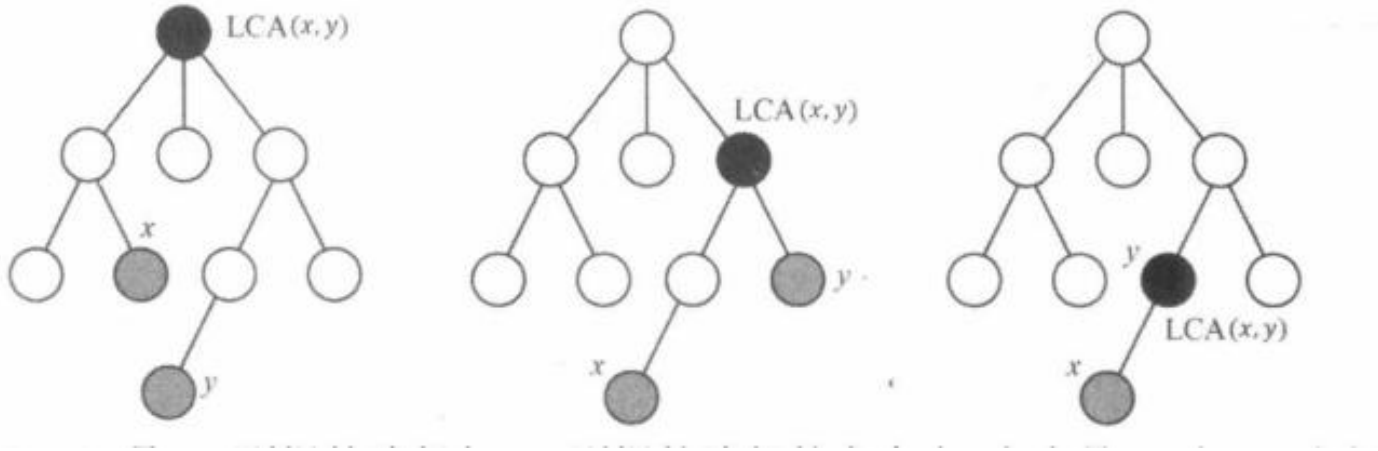
具体来说，就是依次尝试从x向上走$k=2^{\log n},\cdots 2^{1},2^{0}$步，检查到达的结点是否比y深。在每次检查中，若是，则令x=fa[x,k]

```
for(int i=k;i>=0;i--)
    if(depth[fa[x][i]]>=depth[y])
        x=fa[x][i];
```

LCA（最近公共祖先）

✓树上倍增法

3. 若此时x=y，说明已经找到了LCA,LCA就是y(或x)。
这就是该图中的**第三种情况**。




```
if(x==y)
    return x;
    //return y;
```

LCA（最近公共祖先）

✓树上倍增法

4. 用**二进制拆分思想**，把x,y同时向上调整，并保持深度一致且不会相会。

具体来说，就是依次尝试把x,y同时向上走 $k=2^{\log n},\cdots 2^1,2^0$ 步，在每次尝试中，若fa[x,k]≠fa[y,k](即仍未相会)，则令x=fa[x,k],y=fa[y,k]

```
for(int i=k;i>=0;i--){
    if(fa[x][i]!=fa[y][i])
        x=fa[x][i],y=fa[y][i];
}
```

LCA（最近公共祖先）

✓ 树上倍增法

5. 此时x,y必定只差一步就会相会了，它们的父结点fa[x,0]就是LCA。

```
return fa[x][0];
```

LCA（最近公共祖先）

✓树上倍增法

计算LCA(x,y)步骤:

1. 设depth[x]表示x的深度，不妨设depth[x]≥depth[y],否则可交换x,y。

2. 用**二进制拆分思想**，把x向上调整到与y同一深度。

3. 若此时x=y，说明已经找到了LCA,LCA就是y(或x)。

4. 用**二进制拆分思想**，把x,y同时向上调整，并保持深度一致且不会相会。

5. 此时x,y必定只差一步就会相会了，它们的父结点fa[x,0]就是LCA。

LCA(最近公共祖先)

完整树上倍增法LCA代码链接

✓树上倍增法

```
int lca(int x,int y){
    if(depth[x]<depth[y])//步骤1
        swap(x,y);
    for(int i=k;i>=0;i--)//步骤2
        if(depth[fa[x][i]]>=depth[y])
            x=fa[x][i];

    if(x==y)//步骤3
        return x;
        //return y;
    for(int i=k;i>=0;i--){//步骤4
        if(fa[x][i]!=fa[y][i])
            x=fa[x][i],y=fa[y][i];
    }
    return fa[x][0];//步骤5
}
```

LCA（最近公共祖先）

✓ LCA的Tarjan算法

- Tarjan算法本质上使用**并查集对于“向上标记法”**的优化。
- 它是一个**离线算法**，需要把 m 个操作一次性读入，统一计算，最后统一输出。它的时间复杂度是O(m+n)

LCA（最近公共祖先）

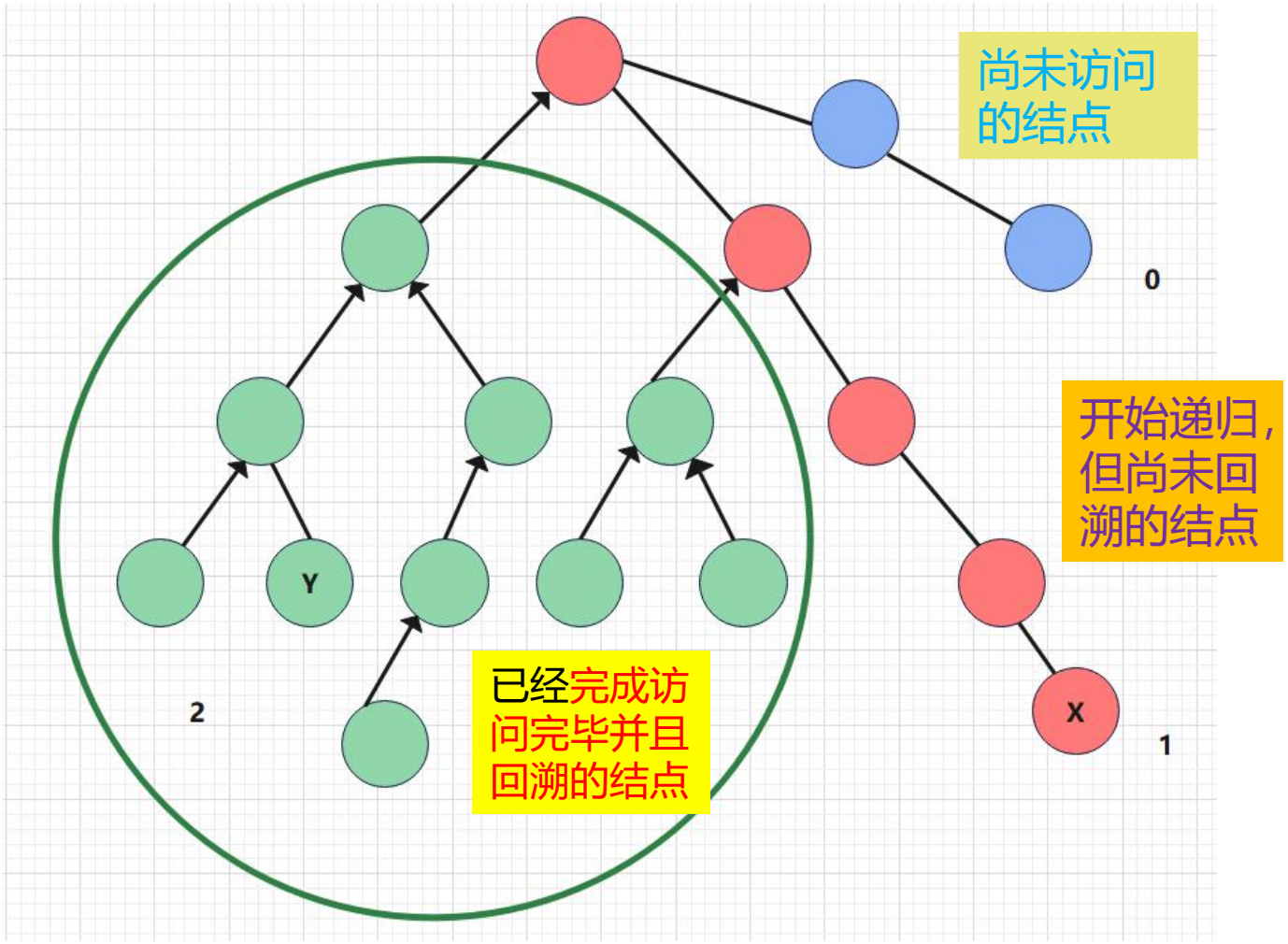
✓ LCA的Tarjan算法

在深度优先遍历的任意时刻，树中结点分为三类：

- ✓ 已经完成访问完毕并且回溯的结点。在这些结点上标记一个整数2
- ✓ 已经开始递归，但尚未回溯的结点。这些结点就是当前正在访问的结点 x以及 x 的祖先。在这些结点上标记一个整数1。
- ✓ 尚未访问的结点，这些结点没有标记(默认标记为0)

LCA（最近公共祖先）

✓ LCA的Tarjan算法

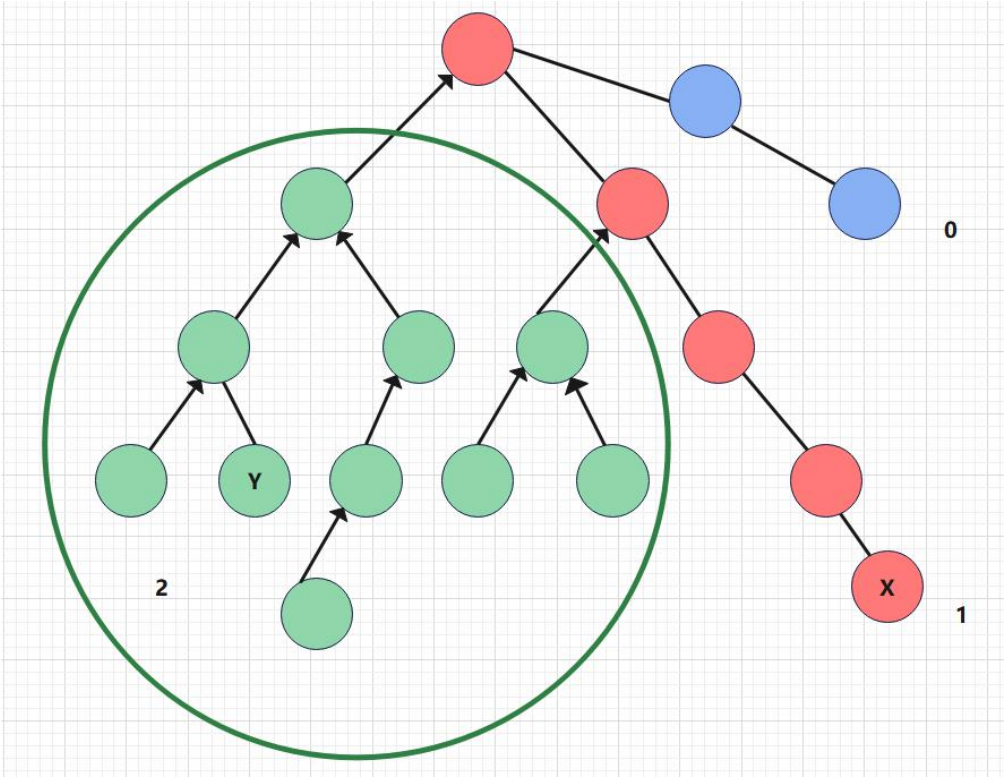


LCA（最近公共祖先）

✓ LCA的Tarjan算法

对于正在访问的结点 x， 它到根结点的路径已经标记为 1。若 y 已经是访问完毕并且回溯的结点，则 LCA(x,y) 就是从 y 向上走到根， 第一个遇到的标记为 1 的点。

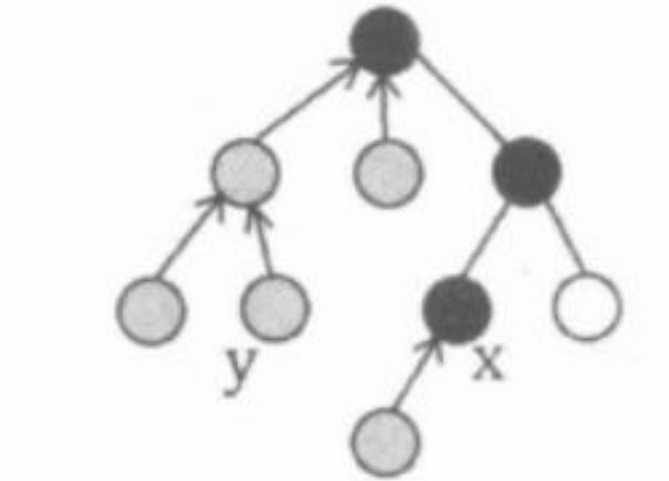
可以利用**并查集**进行优化，当一个结点获得整数 2 的标记时，把它所在的集合合并到它的父结点所在的集合中（合并时它的父结点一定为 1，并且单独构成一个集合）。
这相当于每个完成回溯的结点都有一个指针指向它的父结点，只需要查询 y 所在的集合的代表元素（并查集的 find 操作），等价于从 y 向上一直走到一个开始递归但尚未回溯的结点（具有标记 1），即 LCA(x,y)



LCA（最近公共祖先）

✓ LCA的Tarjan算法

- 在x回溯之前，标记情况与合并情况如下图所示。黑色表示标记为1，灰色表示标记为2，白色表示没有标记，箭头表示执行了合并操作。



- 此时扫描与x相关的所有询问，若询问当中的另一个点y的标记为2，就知道了该询问的回答应该是y在并查集中的代表元素（并查集中 find(y)函数的结果)。

LCA(最近公共祖先)

✓LCA的Tarjan算法

```
typedef pair<int,int> PII;
int t,n,m;

int head[MAXN],ed[2*MAXN],nex[2*MAXN],val[2*MAXN],idx;//链式前向星
int dist[MAXN];//保存每个点到根结点的距离

int p[MAXN],vis[MAXN];//此处用p数组作并查集操作,vis数组用来Tarjan标记数组
int res[MAXN];//存储最终答案
vector<PII> query[MAXN];
//其中query[i]中pair的first表示查询与i相连的另一条边,pair的second存编号
```

LCA（最近公共祖先）

✓LCA的Tarjan算法

先看主函数，对比与在线算法的差别

```
int main(){
    scanf("%d",&t);
    while(t--){
        scanf("%d %d",&n,&m);
        init();
        for(int i=1;i<=n-1;i++){
            int a,b,c;
            scanf("%d %d %d",&a,&b,&c);
            add(a,b,c);
            add(b,a,c);
        }
        for(int i=1;i<=m;i++){
            int a,b;
            scanf("%d %d",&a,&b);
            if(a!=b){
                query[a].push_back({b,i});
                query[b].push_back({a,i});
            }
        }
        dfs(1,-1);
        tarjan(1);
        for(int i=1;i<=m;i++)
            printf("%d\n",res[i]);
    }
    return 0;
}
```

LCA（最近公共祖先）

✓ LCA的Tarjan算法

```
void init(){//初始化
    memset(head,-1,sizeof(head));
    memset(vis,0,sizeof(vis));
    idx=0;
    for(int i=1;i<=n;i++)
        p[i]=i,query[i].clear();
}

void add(int a,int b,int c){//链式前向星的add操作
    ed[idx]=b;
    val[idx]=c;
    nex[idx]=head[a];
    head[a]=idx++;
}
```

LCA（最近公共祖先）

✓ LCA的Tarjan算法

```
void dfs(int root,int fa){//root是当前结点,fa是它的父结点
    for(int i=head[root];i!=-1;i=nex[i]){//遍历root结点的所有边
        int j=ed[i];
        if(j==fa)//当前结点等于父结点不作处理
            continue;
        dist[j]=dist[root]+val[i];//更新j到根结点的距离
        dfs(j,root);//遍历以j为当前结点,root为它的父结点
    }
}

int find(int x){//并查集find函数
    if(p[x]!=x)
        p[x]=find(p[x]);
    return p[x];
}
```

LCA（最近公共祖先）

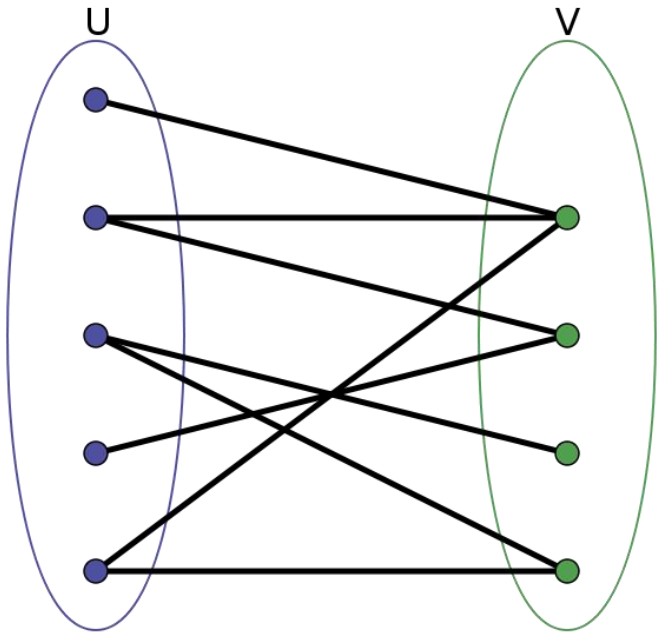
LCA的Tarjan算法完整代码链接

✓LCA的Tarjan算法

```
void tarjan(int u){
    vis[u]=1;//当前结点u标记为1
    for(int i=head[u];i!=-1;i=nex[i]){//遍历u的所有边
        int j=ed[i];
        if(vis[j]==0){//将未遍历的进行Tarjan
            tarjan(j);
            p[j]=u;//同时回溯时将j与u进行join操作
        }
    }
    for(int i=0;i<query[u].size();i++)//对当前u的查询进行更新{
        int  y=query[u][i].first,id=query[u][i].second;
        if(vis[y]==2){//标记为2的找它的父结点
            int anc=find(y);
            res[id]=dist[u]+dist[y]-2*dist[anc];
       //保存结果,任意两点x,y的距离为dist[x]+dist[y]-2*dist[anc]
        }
    }
    vis[u]=2;//回溯完标记为2
}
```

二分图

如果一张**无向图**的N个结点(N≥2)可以分成A，B两个非空集合，其中A∩B=∅，并且在同一集合内的点之间没有边相连，那么称这张无向图为一张二分图。A，B分别称为二分图的左部和右部。

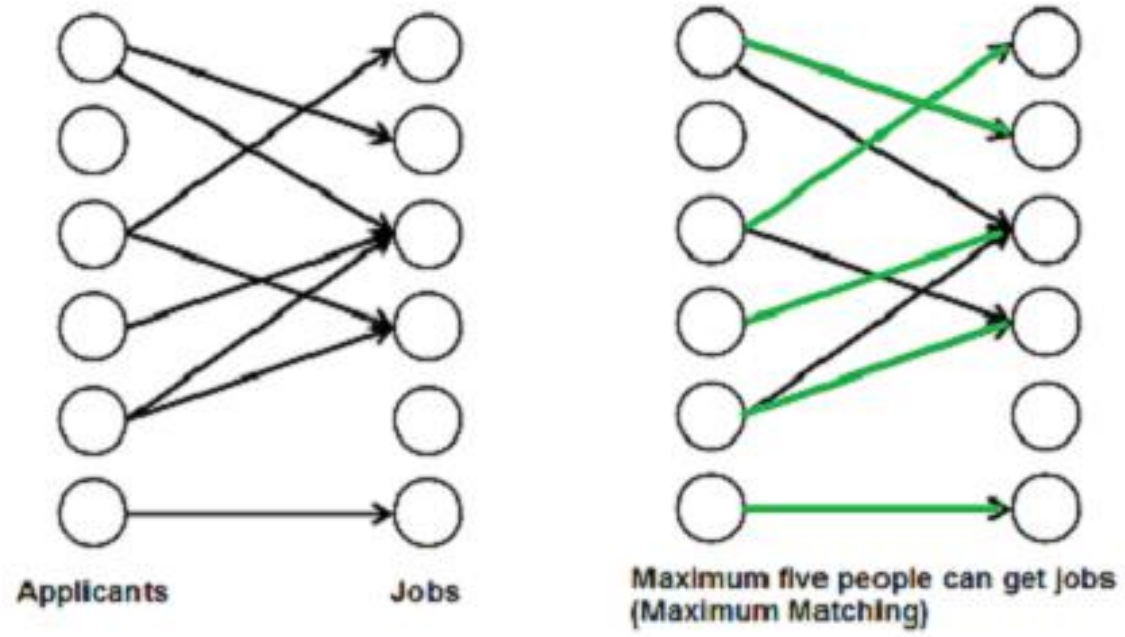


Hungary算法（匈牙利算法）

二分图

解决问题:

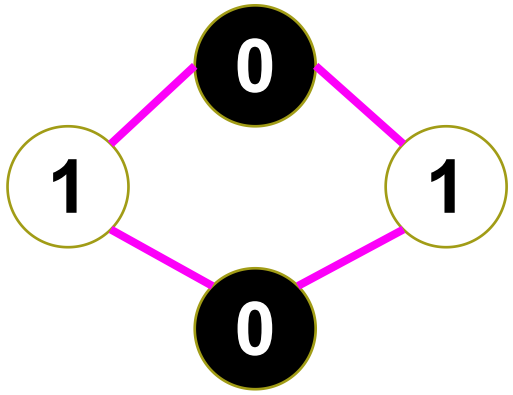
- 有m个职位申请者和n个职位。每个申请人都有他/她感兴趣的工作子集。每个职位空缺只能接受一个应聘者，一个职位应聘者只能被指定一个职位。找一份**工作分配**给申请者，以便尽可能多的申请者得到工作。
- 月老配对问题等等

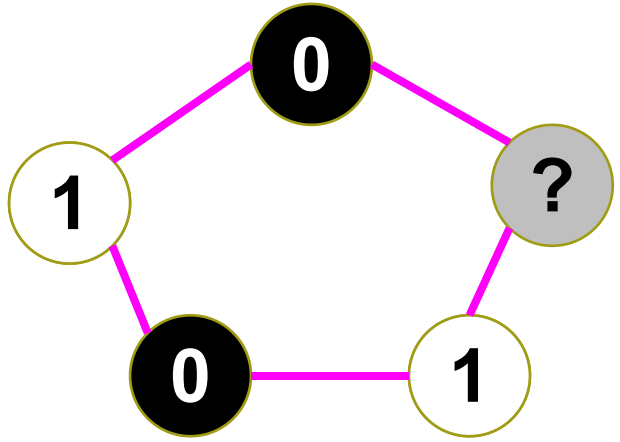


二分图的判定

对于给定的一个无向图，首先需要判断它是否为二分图才能进行相关操作，对于一个无向图来说，当且仅当**不存在有奇数条边构成的环**时，它才是二分图，这是充要条件，举个例子简单证明

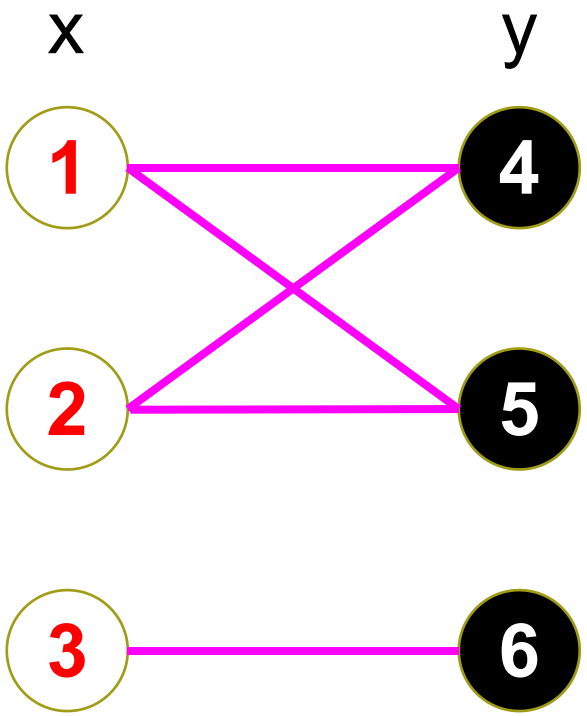
如图，当为奇数环时，根据二分图的定义，可以看到 ？ 处无法被赋值

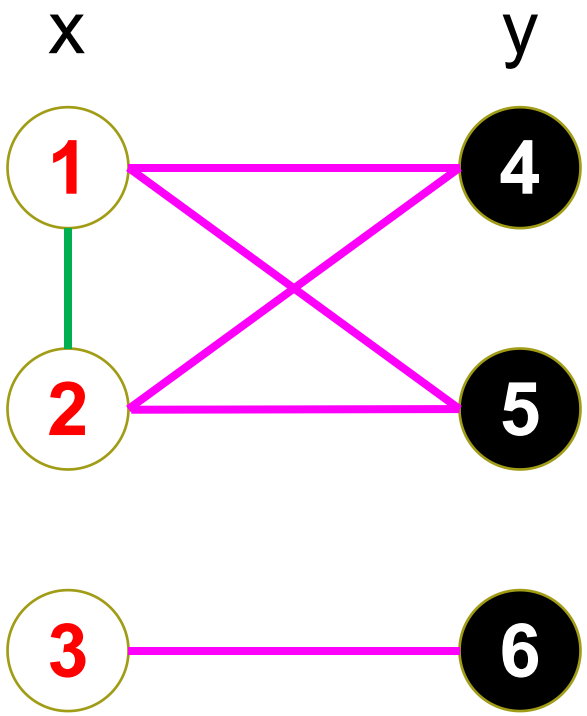




二分图的判定

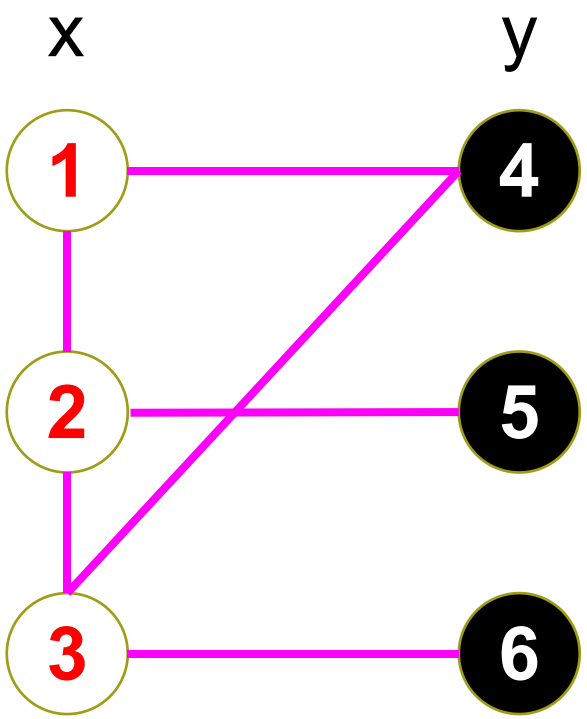
如左图所示，其存在回路。如：1--4--2--5--1，长度为4，偶数。任意一种都为偶数。
如果在1和2之前添一条边，那就不是二分图了，如右图所示
添了1--2的边后，回路就存在了1--4--2--1，长度为3，奇数，所以右图就不是二分图。

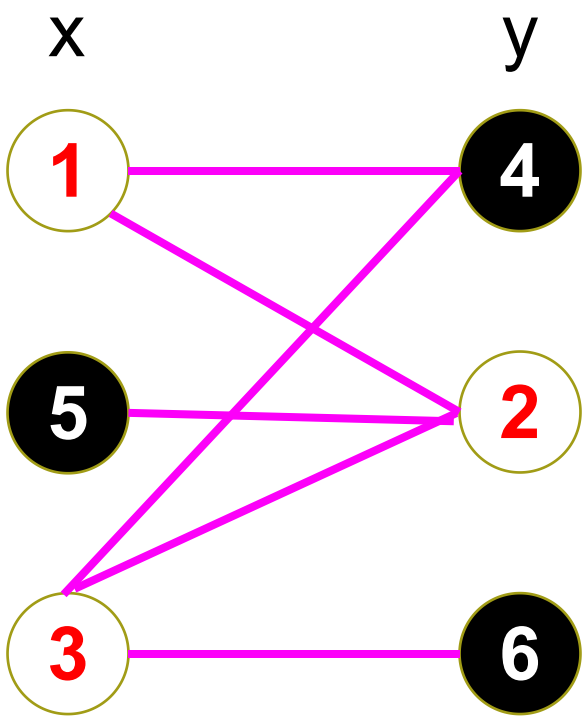




二分图的判定

先看下左图，会发现它存在回路，且任意一种都为偶数，但看上去不像是二分图。不用管顶点的颜色(并未规定123一定是一个集合)，将2和5换个位置，就可以得到右图，如右图所示，将1、5、3分1个集合，4、2、6为1个集合，就是一个二分图。





二分图的判定

定理：一张无向图是二分图，当且仅当图中不存在奇环(长度为奇数的环)

根据二分图的性质可以知道，所有的点被分成了两个集合，那么**相连的两个点必然不属于同一集合**

根据该定理，可以用**染色法**进行二分图的判定。大致思路是:尝试用黑白两种颜色标记图中的结点，当一个结点被标记后，它的所有相邻结点应该被标记与它相反的颜色。若标记过程中产生冲突，则说明图中存在奇环。二分图染色一般基于深度优先遍历实现，时间复杂度为O(N+M)(其中N为点数，M为边数)，其流程如下:

二分图的判定

```
void dfs(int x, int color)
    赋值 v[x] ← color
    对于与 x 相连的每条无向边(x,y)
        if v[y] = 0 then
            dfs(y, 3-color)
        else if v[y] ≠ color then
            判定无向图不是二分图，算法结束
在主函数中
    for i ← 1 to N
        if v[i] = 0 then dfs(i, 1)
    判定无向图是二分图
```

二分图的判定

```
int n,m;
int head[MAXN],ed[MAXM],nex[MAXM],idx;
int col[MAXN];//其中0表示未标记,1表示标记为黑色,2标记为白色

void add(int a,int b){
    ed[idx]=b;
    nex[idx]=head[a];
    head[a]=idx++;
}
```

二分图的判定

```
int dfs(int x,int color){
    col[x]=color;
    for(int i=head[x];i!=-1;i=nex[i]){
        int j=ed[i];
        if(col[j]==0){
            if(dfs(j,3-color)==0)
                return 0;
        }
        else if(col[j]==color)
            return 0;
    }
    return 1;
}
```

Hungary算法（匈牙利算法）

二分图的判定

```
int main(){
    scanf("%d %d",&n,&m);
    memset(head,-1,sizeof(head));
    for(int i=1;i<=m;i++){
        int a,b;
        scanf("%d %d",&a,&b);
        add(a,b),add(b,a);
    }
    /*主函数操作部分*/
    int flag=1;
    for(int i=1;i<=n;i++)
        if(col[i]==0)
            if(dfs(i,1)==0)
            {
                flag=0;
                break;
            }
    /*主函数操作部分*/
    if(flag==1)
        printf("Yes\n");
    else printf("No\n");
    return 0;
}
```

染色法判定二分图
完整代码链接

二分图的判定

关押罪犯 此题可用**二分图的判定**解决，也可以用种类并查集

题目描述

复制Markdown 收起

S 城现有两座监狱，一共关押着 N 名罪犯，编号分别为 $1-N$。他们之间的关系自然也极不和谐。很多罪犯之间甚至积怨已久，如果客观条件具备则随时可能爆发冲突。我们用“怨气值”（一个正整数值）来表示某两名罪犯之间的仇恨程度，怨气值越大，则这两名罪犯之间的积怨越多。如果两名怨气值为 c 的罪犯被关押在同一监狱，他们俩之间会发生摩擦，并造成影响力为 c 的冲突事件。

每年年末，警察局会将本年内监狱中的所有冲突事件按影响力从大到小排成一个列表，然后上报到 S 城 Z 市长那里。公务繁忙的 Z 市长只会去看列表中的第一个事件的影响力，如果影响很坏，他就会考虑撤换警察局长。

在详细考察了 N 名罪犯间的矛盾关系后，警察局长觉得压力巨大。他准备将罪犯们在两座监狱内重新分配，以求产生的冲突事件影响力都较小，从而保住自己的乌纱帽。假设只要处于同一监狱内的某两个罪犯间有仇恨，那么他们一定会在每年的某个时候发生摩擦。

那么，应如何分配罪犯，才能使 Z 市长看到的那个冲突事件的影响力最小？这个最小值是多少？

输入格式

每行中两个数之间用一个空格隔开。第一行为两个正整数 N, M，分别表示罪犯的数目以及存在仇恨的罪犯对数。接下来的 M 行每行为三个正整数 a_j, b_j, c_j，表示 a_j 号和 b_j 号罪犯之间存在仇恨，其怨气值为 c_j。数据保证 $1 < a_j \leq b_j \leq N, 0 < c_j \leq 10^9$，且每对罪犯组合只出现一次。

输出格式

共 1 行，为 Z 市长看到的那个冲突事件的影响力。如果本年内监狱中未发生任何冲突事件，请输出 `0`。

二分图的判定

输入 #1 复制

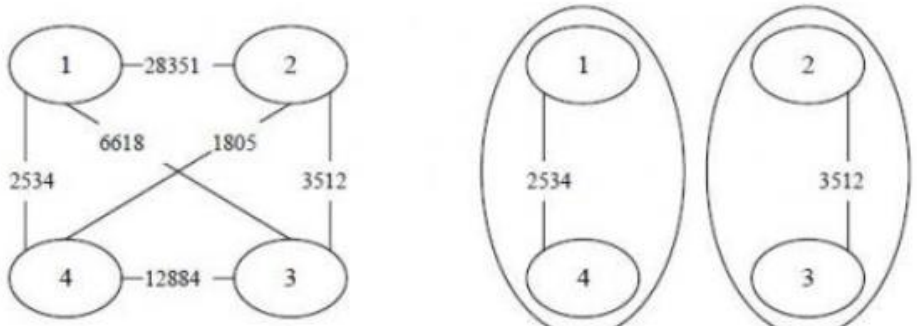
```
4 6
1 4 2534
2 3 3512
1 2 28351
1 3 6618
2 4 1805
3 4 12884
```

输出 #1 复制

```
3512
```

说明/提示

【输入输出样例说明】罪犯之间的怨气值如下面左图所示，右图所示为罪犯的分配方法，市长看到的冲突事件影响力是 3512（由 2 号和 3 号罪犯引发）。其他任何分法都不会比这个分法更优。



二分图的判定

- 将罪犯当做点，罪犯之间的仇恨关系当做点与点之间的无向边，边的权重是罪犯之间的仇恨值。
- 那么原问题变成：将所有点分成两组，使得各组内边的权重的最大值尽可能小。
- 我们在 $[0,10^9]$之间枚举最大边权limit，当limit 固定之后，剩下的问题就是：
- 判断能否将所有点分成两组，使得所有权值大于limit 的边都在组间，而不在组内。也就是判断由所有点以及所有权值大于limit 的边构成的新图是否是二分图。
- 判断二分图可以用**染色法**，时间复杂度是 O(N+M)，其中 N 是点数，M 是边数

二分图的判定

为了加速算法，我们来考虑是否可以用**二分**枚举 limit， 假定最终最大边权的最小值是 Ans:

- ✓ 那么当 limit∈[ans,10^9]时，所有边权大于 limit 的边，必然是所有边权大于Ans 的边的子集，因此由此构成的新图也是二分图。
- ✓ 当limit∈[0,ans−1] 时，由于ans 是新图可以构成二分图的最小值，因此由大于 limit 的边构成的新图一定不是二分图。
- ✓ 所以整个区间具有二段性，可以二分出分界点ans 的值。

时间复杂度分析
总共二分logC 次，其中 C 是边权的最大值，每次二分使用染色法判断二分图，时间复杂度是O(N+M)，其中 N 是点数，M 是边数。因此总时间复杂度是O((N+M)logC)。

二分图的判定

关押罪犯的完整染色法+二分代码链接

```
int dfs(int x,int color,int limit){
    col[x]=color;
    for(int i=head[x];i!=-1;i=nex[i]){
        if(val[i]<=limit)
            continue;
        int j=ed[i];
        if(col[j]==0){
            if(dfs(j,3-color,limit)==0)
                return 0;
        }
        else if(col[j]==color)
            return 0;
    }
    return 1;
}
```

```
int check(int limit){
    memset(col,0,sizeof(col));
    for(int i=1;i<=n;i++)
        if(col[i]==0)
            if(dfs(i,1,limit)==0)
                return 0;
    return 1;
}
```

```
    int l=0,r=1e9;
    while(l<r)
    {
        int mid=(l+r)/2;
        if(check(mid))
            r=mid;
        else l=mid+1;
    }
```

二分图的最大匹配

匹配：在图论中，一个「匹配」（matching）是一个边的集合，其中任意两条边都没有公共顶点。图2、图 3 中红色的边就是图 1 的匹配。

最大匹配：一个图所有匹配中，所含匹配边数最多的匹配，称为这个图的最大匹配。图 3 是一个最大匹配，它包含 4 条匹配边。

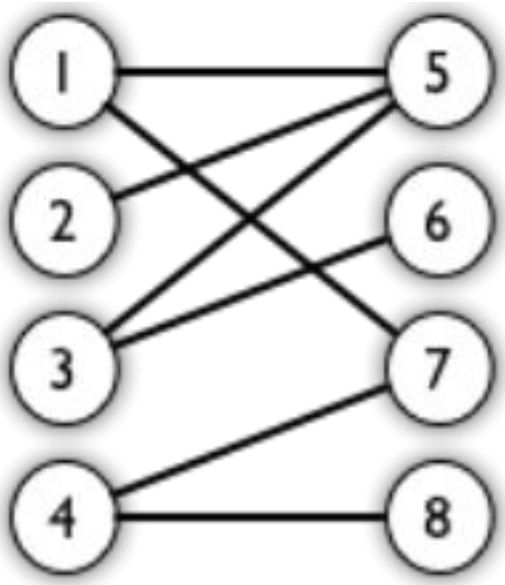


图1

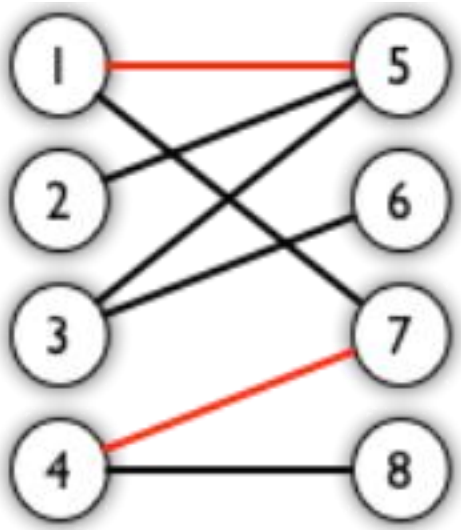


图2

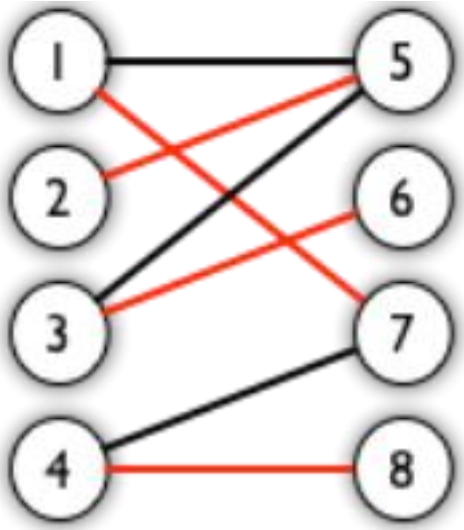
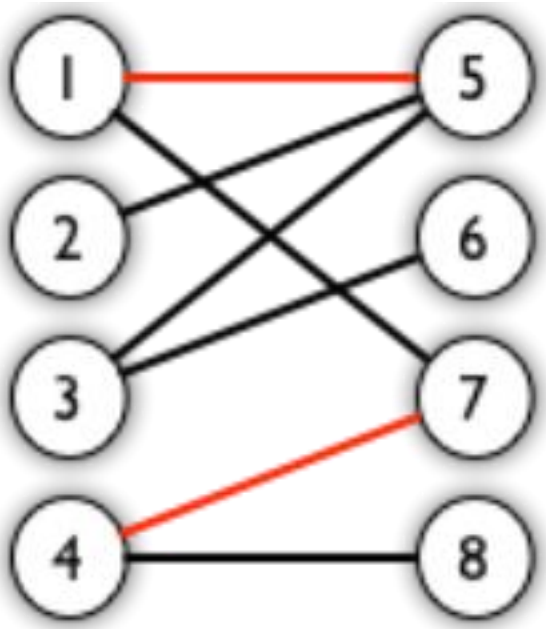


图3

二分图的最大匹配

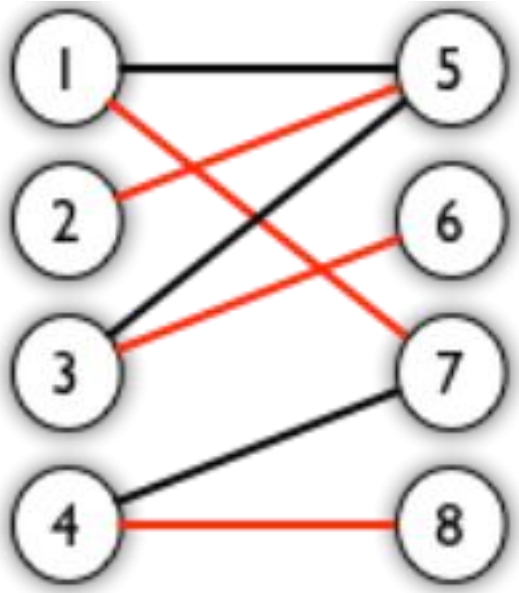
匹配点、匹配边、非匹配点、非匹配边：对于任意一组匹配S（S是一个边集），属于S的边被称为“匹配边”，不属于S的边称为“非匹配边”。匹配边的端点被称为“匹配点”，其他结点被称为”非匹配点”

例如该图 中 1、4、5、7 为匹配点，其他顶点为非匹配点；1-5、4-7为匹配边，其他边为非匹配边。



二分图的最大匹配

完美匹配：如果一个图的某个匹配中，**所有的顶点都是匹配点**，那么它就是一个完美匹配。该图 是一个完美匹配。显然，完美匹配一定是最大匹配（完美匹配的任何一个点都已经匹配，添加一条新的匹配边一定会与已有的匹配边冲突）。**但并非每个图都存在完美匹配**。

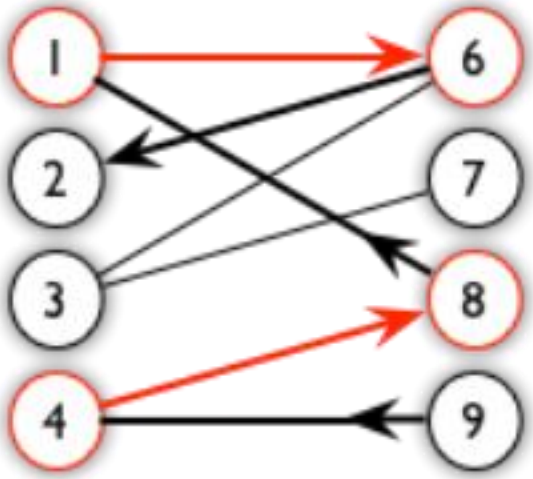


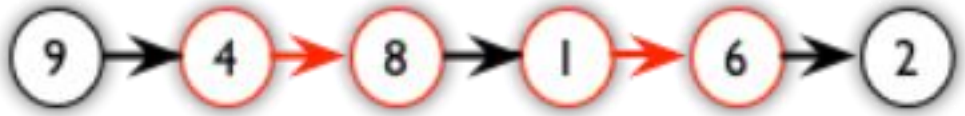
二分图的最大匹配

交替路：从一个未匹配点出发，依次经过非匹配边、匹配边、非匹配边...形成的路径叫交替路。

增广路：如果二分图中存在一条连接两个非匹配点的路径path,使得非匹配边与匹配边在path上交替出现，那么称为path是匹配S的增广路。

左图中的一条增广路如右图所示（图中的匹配点均用红色标出）





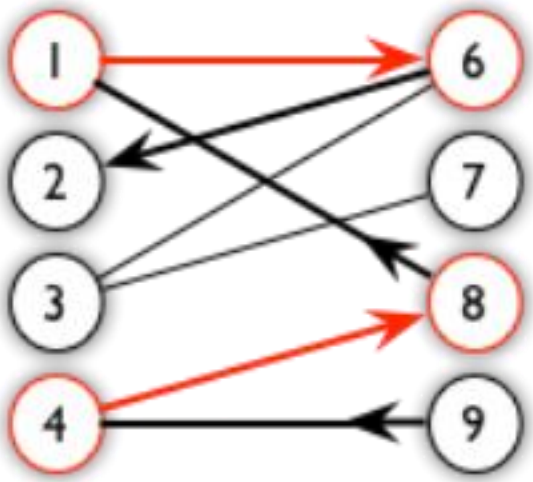
二分图的最大匹配

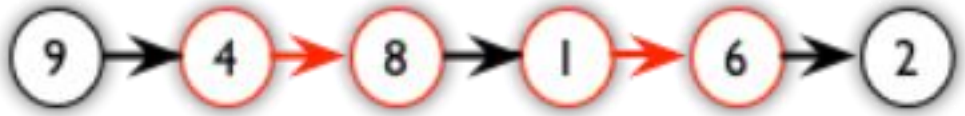
增广路的性质:

- 长度len是奇数;
- 路径中第1,3,5,...,len条边是非匹配边，第2,4,6,...len-1条边是匹配边。

正因如此，如果把路径上的所有边的状态取反，原来的匹配边变成非匹配的，原来的不匹配的变成匹配的，那么得到新的边集S'仍然是一组匹配，并且匹配边数增加了1.

故: **二分图的一组匹配S是最大匹配，当且仅当图中不存在S的增广路。**





二分图的最大匹配

匈牙利算法（增广路算法）

Hungary 算法（音译为“匈牙利算法”），又称为增广路算法，是用来解决二分图匹配问题的利器。
它的主要过程如下:

- 1. 设S=∅ ,即所有边都是非匹配边;
- 2. 寻找增广路path,把路径上所有边的匹配状态取反，得到一个更大的匹配S';
- 3. 重复第二步，直至图中不存在增广路。

该算法关于如何找到一条增广路。匈牙利算法依次尝试给每一个左部结点x寻找一个匹配的右部结点y。
右部结点y能与左部结点x匹配，需满足以下两个条件之一:

- 1. y本身就是非匹配点。

此时无向边(x,y)本身就是非匹配边，自己构成一条长度为1的增广路。

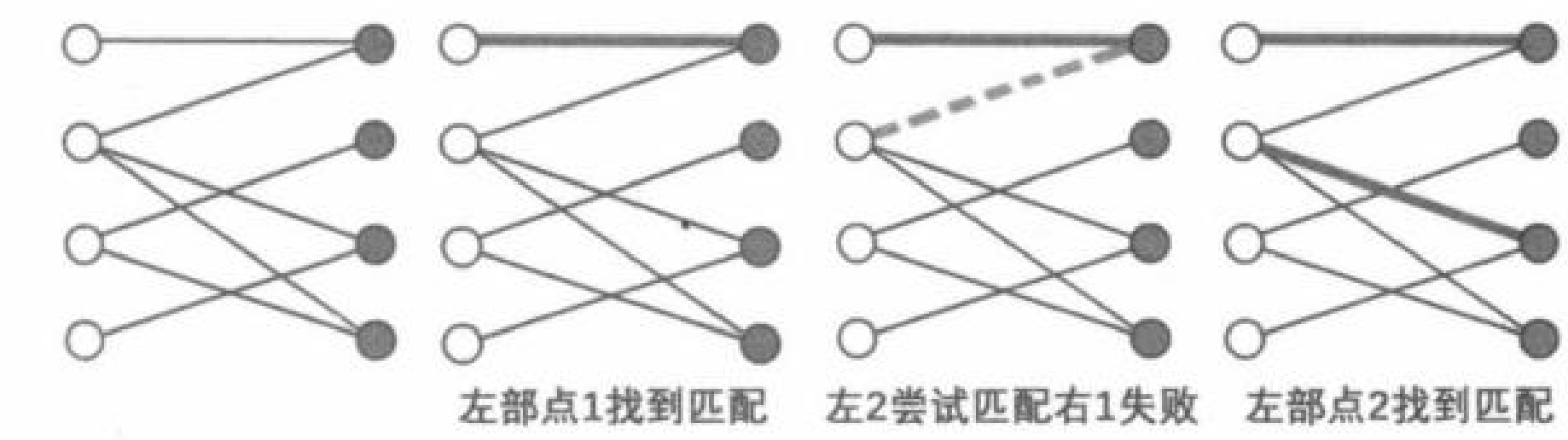
- 2. y已经与左部结点x'匹配，但从x'出发能找到另一个右部结点y'与之匹配。

此时x-->y-->x'-->y'为一条增广路。

二分图的最大匹配

匈牙利算法（增广路算法）

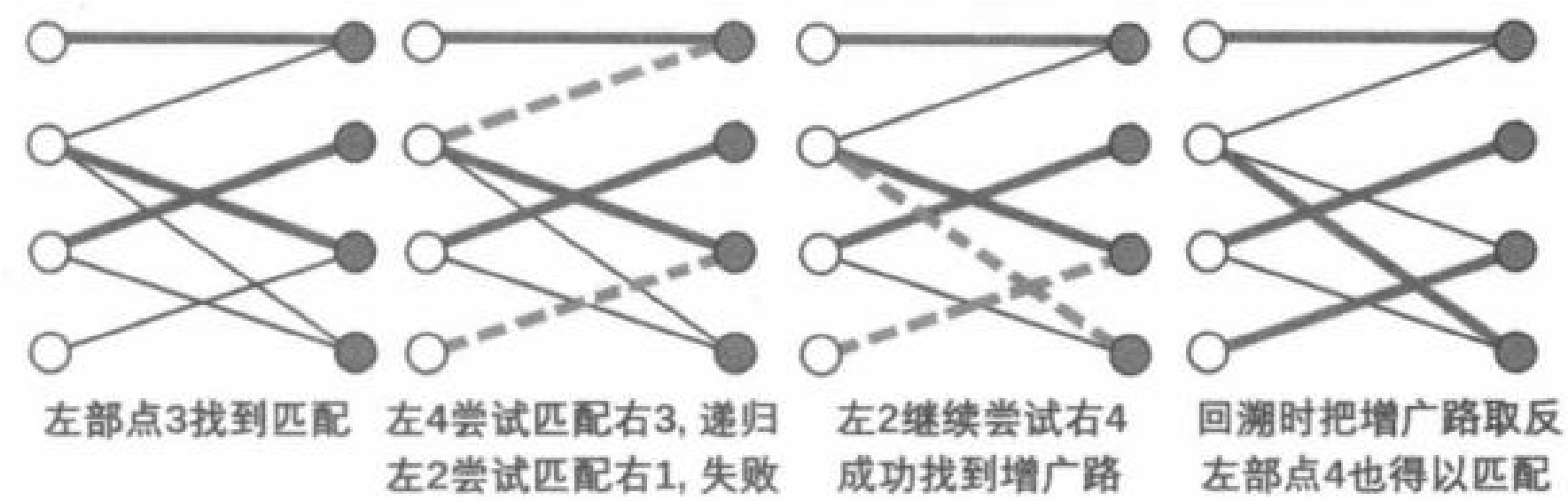
- 在实际程序实现中，采用深度优先搜索的框架，递归地从x出发寻找增广路。若找到，则在深搜回溯时，正好把路径上的状态取反。另外，可用全局bool数组标记结点的访问情况，避免重复搜索。



二分图的最大匹配

匈牙利算法（增广路算法）

- 在实际程序实现中，采用深度优先搜索的框架，递归地从x出发寻找增广路。若找到，则在深搜回溯时，正好把路径上的状态取反。另外，可用全局bool数组标记结点的访问情况，避免重复搜索。



二分图的最大匹配

匈牙利算法（增广路算法）

- 匈牙利算法的正确性基于贪心策略,它的一个重要特点是:当一个结点成为匹配点后，至多因为找到增广路而更换匹配对象，但是绝不会再变回非匹配点。
- 对于每个左部结点，寻找增广路最多遍历整张二分图一次。因此，**该算法的时间复杂度为O(NM)**，但实际运行时间一般远小于O(NM)。

二分图的最大匹配

匈牙利算法（增广路算法）

```
int n1,n2,m;

int head[MAXN],ed[MAXM],nex[MAXM],idx;
//此处可不用开两倍,因为只会用到左部结点到右部结点的边

int match[MAXN],res;//右部结点匹配的左部结点
int vis[MAXN];//标记当前x的右部结点是否被访问,与是否被之前已经匹配无关

void add(int a,int b){
    ed[idx]=b;
    nex[idx]=head[a];
    head[a]=idx++;
}
```

二分图的最大匹配

二分图最大匹配完整代码链接

匈牙利算法（增广路算法）

```
int dfs(int x){
    for(int i=head[x];i!=-1;i=nex[i]){
        int j=ed[i];
        if(vis[j]==0){
            vis[j]=1;
            if(match[j]==0||dfs(match[j])==1){
                match[j]=x;
                return 1;
            }
        }
    }
    return 0;

}
```

```
    for(int i=1;i<=n1;i++){//main函数
        memset(vis,0,sizeof(vis));
        if(dfs(i)==1)
            res++;
    }
```