21级实验室暑假第一讲

目录

- 单调栈&&单调队列

单调栈和单调队列都有共同的特性：**单调性**。所谓单调性就是在一段区间上，这段区间的数是**线性递增**的或**线性递减**的。栈和队列中一直维护一串单调的数字，那么这个栈和队列就叫做单调栈和单调队列。

- 单调栈

单调栈

- 单调递增栈：单调递增栈就是从栈底到栈顶数据是**从小到大**
- 单调递减栈：单调递减栈就是从栈底到栈顶数据是**从大到小**

单调栈的性质如下：

- ✓ 单调栈里面的元素具有单调性；
- ✓ 元素加入栈前会把栈顶破坏单调性的元素删除；
- ✓ 使用单调栈可以找到元素向左遍历的第一个比它**小**的元素（**单调递增栈**），也可以找到元素元素向左遍历第一个比它**大**的元素（**单调递减栈**）；
- ✓ 一般使用单调栈的题目具有以下的两点：**离自己最近（栈的后进先出的性质）；比自己大（小）、高（低）**

- 单调栈

举例：以 3　1　4　5　2　7 为例，其**单调递增栈**具体过程如下图所示：

从左到右依次入栈，如果**栈为空或者入栈元素大于栈顶元素值**入栈；否则，如果此时入栈则会破坏其单调性，则需要将比**入栈元素大的元素全部出栈再将入栈元素入栈**，单调递减栈反之。

- 3入栈时，栈为空，直接入栈，栈内元素为3；
- 1入栈时，栈顶元素3大于1，栈顶元素出栈，此时栈为空，直接入栈，栈内元素为1；
- 4入栈时，栈顶元素1小于4，直接入栈，栈内元素为1,4；
- 5入栈时，栈顶元素4小于5，直接入栈，栈内元素为1,4,5；
- 2入栈时，栈顶元素5大于2，栈顶元素出栈，此时栈顶元素4大于2，栈顶元素再次出栈，然后栈顶元素1小于2，直接入栈，栈内元素为1,2；
- 7入栈时，栈顶元素2小于7，直接入栈，栈内元素1,2,7。

- 单调栈

流程：

```
stack<int> sta;
//此处以单调递增栈为例
for(遍历需要入栈的数组)
{
    if( 栈空 || 栈顶元素小于入栈元素)
    {
        //更新结果;
        入栈元素入栈;
    }
    else
    {
        while( 栈不为空 && 栈顶元素大于等于入栈元素)
        {
            栈顶元素出栈;
            更新结果;
        }
        入栈元素入栈;
    }
}
```

- 单调栈

例题

给定一个长度为 N 的整数数列，输出每个数左边第一个比它小的数，如果不存在则输出 -1。

输入格式

第一行包含整数 N，表示数列长度。

第二行包含 N 个整数，表示整数数列。

输出格式

共一行，包含 N 个整数，其中第 i 个数表示第 i 个数的左边第一个比它小的数，如果不存在则输出 -1。

数据范围

$1 \leq N \leq 10^5$
$1 \leq$ 数列中元素 $\leq 10^9$

输入样例：

```
5
3 4 2 7 5
```

输出样例：

```
-1 3 -1 2 2
```

```cpp
#include <iostream>
#include <cstring>
#include <stack>

using namespace std;

const int MAXN=100010;

int n,a[MAXN],res[MAXN];
stack<int> sta;
```

```cpp
int main(){
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
        scanf("%d",&a[i]);
    for(int i=1;i<=n;i++){//遍历数组中每个元素

        if(sta.size()==0||a[i]>sta.top()){//单调递增栈入栈操作
            ;
        }
        else{
            while(sta.size()>0&&a[i]<=sta.top())//保证有序性
                sta.pop();
        }
        //更新结果
        if(sta.size()==0)
            res[i]=-1;
        else res[i]=sta.top();
        sta.push(a[i]);//入栈
    }
    for(int i=1;i<=n;i++)
        printf("%d ",res[i]);
    return 0;
}
```

代码链接

```cpp
#include <iostream>
#include <cstring>
#include <stack>

using namespace std;

const int MAXN=100010;

int n,a[MAXN],res[MAXN];
stack<int> sta;
```

```cpp
int main(){
    scanf("%d",&n);
    for(int i=1;i<=n;i++)
        scanf("%d",&a[i]);
    for(int i=1;i<=n;i++){//遍历数组中每个元素
        //更新结果
        if(sta.size()==0)
            res[i]=-1;
        else res[i]=sta.top();
        sta.push(a[i]);//入栈
    }
    for(int i=1;i<=n;i++)
        printf("%d ",res[i]);
    return 0;
}
```

此处可直接简写成如下部分：

```cpp
while(sta.size()>0&&a[i]<=sta.top())//保证有序性
    sta.pop();
```

[代码链接]

- 单调队列

单调队列

- 单调递增队列：单调递增队列就是从队头到队尾数据是**从小到大**
- 单调递减队列：单调递减队列就是从队头到队尾数据是**从大到小**

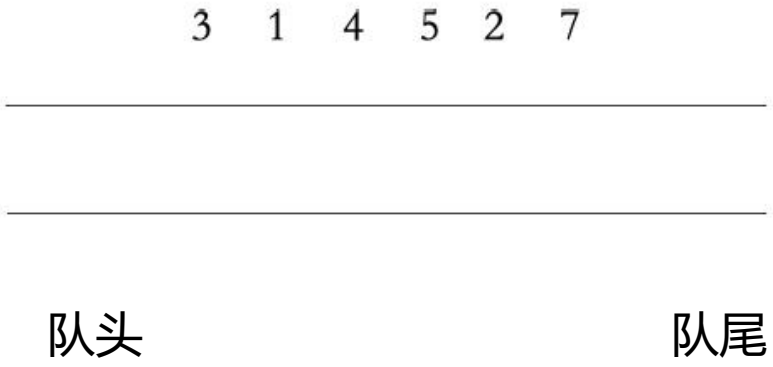
单调队列中元素之间的关系具有严格单调有序性，而且，**队首和队尾**都可以进行出队操作，只有**队尾**可以进行**入队**操作。因此，在单调队列中求**最值**十分容易

单调队列与普通队列有些不同，因为右端(队尾)既可以插入又可以删除，因此在代码中通常用**一份数组和front与rear两个指针**来实现，而不是使用 STL 中的queue。如果一定要使用 STL ，那么则可以使用双端队列(即两端都可以插入和删除)，即deque 。

- 单调队列

✓ 对于单调递增队列，设当前准备入队的元素为e，从队尾开始把队列中的元素逐个与e对比，把比e大或者与e相等的元素逐个删除，直到遇到一个比e小的元素或者队列为空为止，然后把当前元素e插入到队尾。
✓ 对于单调递减队列也是同样道理，只不过从队尾删除的是比e小或者与e相等的元素。

举例：以 3 1 4 5 2 7 为例，其**单调递增队列**具体过程如下图所示：



从左到右依次入队列，如果**队列为空或者入队元素大于队尾元素值**入队列；否则，如果此时入队列则会破坏其单调性，则需要将比**入队元素大的元素全部出队尾再将入队元素入队列**，单调递减队列反之。

- 3入队列时，队列为空，直接入队列，队列内元素为3；
- 1入队列时，队尾元素3大于1，队尾元素出队列，此时队列为空，直接入队列，队列内元素为1；
- 4入队列时，队尾元素1小于4，直接入队列，队列内元素为1,4；
- 5入队列时，队尾元素4小于5，直接入队列，队列内元素为1,4,5；
- 2入队列时，队尾元素5大于2，队尾元素出队列，此时队尾元素4大于2，队尾元素再次出队列，然后队尾元素1小于2，直接入队列，队列内元素为1,2；
- 7入队列时，队尾元素2小于7，直接入队列，队列内元素1,2,7。

- 单调队列

对比可发现，中间过程类似,但区别在于队列本身的性质是先进先出，可从队头弹出元素，故单调队列常用于解决**滑动窗口**问题。

- 单调队列

例题　滑动窗口

- 给你一个整数数组 nums，有一个大小为 k 的滑动窗口从数组的最左侧移动到数组的最右侧。你只可以看到在滑动窗口内的 k 个数字。滑动窗口每次只向右移动一位。
- 返回滑动窗口中的最大值, 最小值
- The array is [1 3 -1 -3 5 3 6 7], and k is 3.

Window position	Minimum value	Maximum value
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
1 3 -1 [-3 5 3] 6 7	-3	5
1 3 -1 -3 [5 3 6] 7	3	6
1 3 -1 -3 5 [3 6 7]	3	7

- 求最小值使用单调递增队列，而求最大值使用单调递减队列，维护好当前窗口的队列元素，每次取出队头即为所求最小(大)值，其中单调队列中的元素为元素**下标**

- 单调队列

```
#include <iostream>
#include <cstdio>

using namespace std;

const int MAXN=1000010;

int n,k,a[MAXN];
int q[MAXN],front,rear;
//队头front,队尾rear
int main()
{
    scanf("%d %d",&n,&k);
    for(int i=0;i<=n-1;i++)
        scanf("%d",&a[i]);
    front=0,rear=-1;//求最小值,使用单调递增队列
    for(int i=0;i<=n-1;i++){
        if(front<=rear&&q[front]<=i-k)//处理滑动窗口左端
            front++;
        while(front<=rear&&a[i]<=a[q[rear]])
        //把大于等于入队元素在队列中的值都剔除
            rear--;
        q[++rear]=i;//将该元素的下标放入队列中
        if(i>=k-1)//达到区间长度输出队头元素
            printf("%d ",a[q[front]]);
    }
    printf("\n");
    front=0,rear=-1;//求最大值,使用单调递减队列
    for(int i=0;i<=n-1;i++){
        if(front<=rear&&q[front]<=i-k)//处理滑动窗口左端
            front++;
        while(front<=rear&&a[i]>=a[q[rear]])
        //把小于等于入队元素在队列中的值都剔除
            rear--;
        q[++rear]=i;//将该元素的下标放入队列中
        if(i>=k-1)//达到区间长度输出队头元素
            printf("%d ",a[q[front]]);
    }
    return 0;
}
```

采用数组模拟队列，本人尝试很多次发现POJ使用C++不超时使用G++超时

代码链接

- 单调队列

```
#include <iostream>
#include <cstdio>
#include <queue>

using namespace std;

const int MAXN=1000010;

int n,k,a[MAXN];
deque<int> dq1,dq2;

int main()
{
    scanf("%d %d",&n,&k);
```

采用双端队列

代码链接

```
    for(int i=0;i<=n-1;i++){//求最小值,使用单调递增队列
        scanf("%d",&a[i]);
        if(dq1.size()>0&&dq1.front()<=i-k)//处理滑动窗口左端
            dq1.pop_front();//弹出队头
        while(dq1.size()>0&&a[i]<=a[dq1.back()])
        //把大于等于入队元素在队列中的值都剔除
            dq1.pop_back();
        dq1.push_back(i);//将该元素的下标放入队列中
        if(i>=k-1)//达到区间长度输出队头元素
            printf("%d ",a[dq1.front()]);
    }
    printf("\n");
    for(int i=0;i<=n-1;i++){//求最大值,使用单调递减队列
        if(dq2.size()>0&&dq2.front()<=i-k)//处理滑动窗口左端
            dq2.pop_front();
        while(dq2.size()>0&&a[i]>=a[dq2.back()])
        //把小于等于入队元素在队列中的值都剔除
            dq2.pop_back();
        dq2.push_back(i);//将该元素的下标放入队列中
        if(i>=k-1)//达到区间长度输出队头元素
            printf("%d ",a[dq2.front()]);
    }
    return 0;
}
```

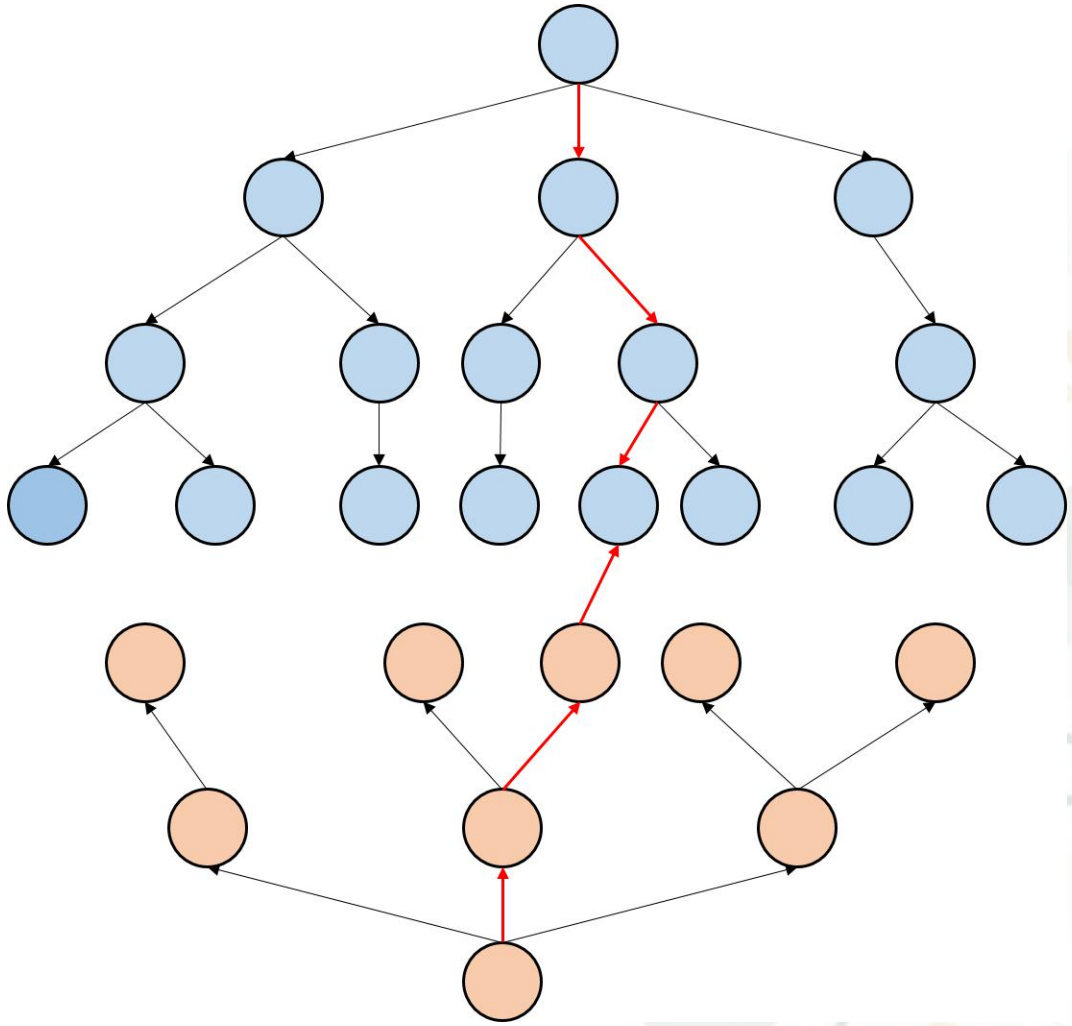
- 双向广搜（DBFS）

■ DBFS算法是对BFS算法的一种扩展。

- BFS算法从起始节点以广度优先的顺序不断扩展，直到遇到目的节点
- DBFS算法从两个方向以广度优先的顺序同时扩展，一个是从起始节点开始扩展，另一个是从目的节点扩展，直到一个扩展队列中出现另外一个队列中已经扩展的节点，也就相当于两个扩展方向出现了交点，那么可以认为找到了一条路径。

■ 比较

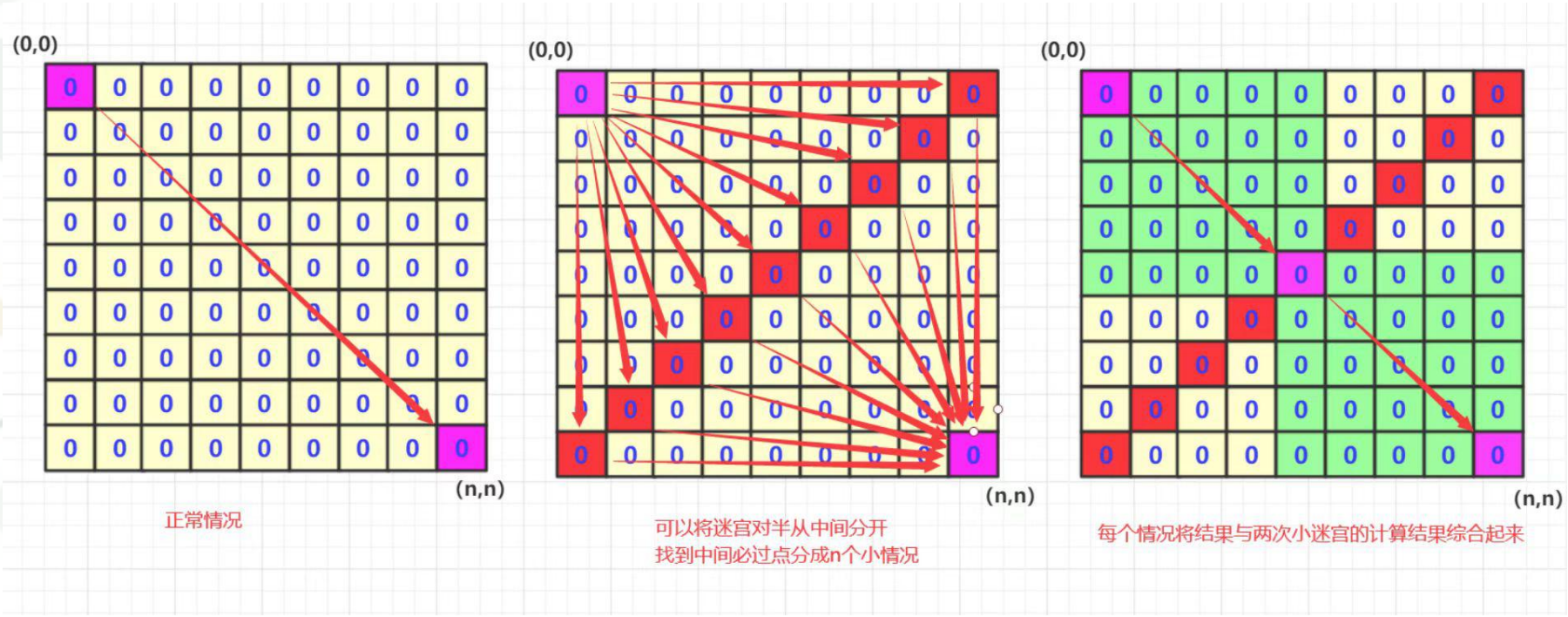
- DBFS算法相对于BFS算法来说，由于采用了从两个根开始扩展的方式，搜索树的深度得到了明显的减少，所以在算法的时间复杂度和空间复杂度上都有较大的优势。



- 双向广搜（DBFS）

广度双向搜索通常有两中方法:
1. 两个方向交替扩展
2. 选择结点个数较少的那个方向先扩展。

方法2克服了两方向结点的生成速度不平衡的状态,明显提高了效率。



- 双向广搜（DBFS）

```
int found;
void DBFS()//双向广搜{
    found=false;
    memset(vis,0,sizeof(vis));//判重数组
    while(Q1.size()>0)  Q1.pop();//正向队列
    while(Q2.size()>0)  Q2.pop();//反向队列
    //正向扩展的状态标记为1,反向扩展标记为2
    vis[s1]=1;//初始状态s1标记为1
    vis[s2]=2;//结束状态s2标记为2
    Q1.push(s1.state);//初始状态入正向队列
    Q2.push(s2.state);//结束状态入反向队列
    while(Q1.size()>0||Q2.size()>0){
        if(Q1.size()>0)
            BFS_expand(Q1,1);//在正向队列中搜索
        if(found==1)//搜索结束
            return ;
        if(Q2.size()>0)
            BFS_expand(Q2,0);//在反向队列中搜索
        if(found==1)//搜索结束
            return ;
    }
}
```

- 双向广搜（DBFS）

```
void BFS_expand(queue<Status>& Q,int flag){
    Status s=Q.front();//从队列中得到头结点        Q.pop();
    for(每个s的子结点t){
        t.state=Get_state(t);//获取子结点的状态
        if(flag==1){//正向队列中判断
            if(vis[t.state]!=1){//没有在正向队列中出现过
                if(vis[t.state]==2){//该状态在反向队列中出现过
                    //各种操作   found=1;
                    return ;
                }
                vis[t.state]=1;//标记为在正向队列中    Q.push(t);
            }
        }
        else//在反向队列中判断{
            if(vis[t.state]!=2){//没有在反向队列出现过
                if(vis[t.state]==1){//该状态在正向队列中出现过
                    //各种操作   found=1;
                    return ;
                }
                vis[t.state]=2;//标记为在反向队列中    Q.push(t);
            }
        }

    }
}
```

- 双向广搜（DBFS）

走迷宫
描述
一个迷宫由R行C列格子组成，有的格子里有障碍物，不能走；有的格子是空地，可以走。
给定一个迷宫，求从左上角走到右下角最少需要走多少步(数据保证一定能走到)。只能在水平方向或垂直方向走，不能斜着走。
输入
第一行是两个整数， R和C，代表迷宫的长和宽。 （ 1<= R，C <= 40)
接下来是R行，每行C个字符，代表整个迷宫。
空地格子用'.'表示，有障碍物的格子用'#'表示。迷宫左上角和右下角都是'.'。
输出
输出从左上角走到右下角至少要经过多少步（即至少要经过多少个空地格子）。计算步数要包括起点和终点。
样例输入
5 5
..###
#....
#.#.#
#.#.#
#.#..
样例输出
9

- 双向广搜（DBFS）
  - ✓ 两个方向交替扩展

```
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>

using namespace std;

const int MAXN=50;
typedef pair<int,int> PII;
int n,m;
int dist[MAXN][MAXN],vis[MAXN][MAXN];
char g[MAXN][MAXN];
int dir[4][2]={{-1,0},{0,1},{1,0},{0,-1}};
queue<PII> Q1,Q2;
int found;
```

代码链接

```
int main(){
    scanf("%d %d",&n,&m);
    for(int i=1;i<=n;i++)
        scanf("%s",g[i]+1);
    int res=dbfs();
    printf("%d\n",res);
    return 0;
}
```

- 双向广搜（DBFS）

✓ 两个方向交替扩展

```
int dbfs(){
    found=-1;
    memset(vis,0,sizeof(vis));
    while(Q1.size()>0)
        Q1.pop();
    while(Q2.size()>0)
        Q2.pop();
    vis[1][1]=1,vis[n][m]=2;
    Q1.push({1,1});Q2.push({n,m});//Q2.push(make_pair{n,m});
    dist[1][1]=dist[n][m]=0;
    while(Q1.size()>0||Q2.size()>0){
        PII temp;
        if(Q1.size()>0)
            temp=Q1.front(),BFS_expand(Q1,1);
        if(found==1)
            return dist[temp.first][temp.second]+2;
        //两种状态相遇,这里由于起点题目要求要算一步，且由(x0,y0) -> (x1,y1)状态也 要花费一步,故各需要+1
        if(Q2.size()>0)
            temp=Q2.front(),BFS_expand(Q2,0);
        if(found==1)
            return dist[temp.first][temp.second]+2;
    }
    return -1;
}
```

此处可使用&&号代替||,因为一旦扩展不出去,外面也进不去

- 双向广搜（DBFS）

✓ 两个方向交替扩展

```
void BFS_expand(queue<PII>& Q,int flag){
    PII s=Q.front();
    Q.pop();
    for(int i=0;i<=3;i++){
        int dx=s.first+dir[i][0],dy=s.second+dir[i][1];
        if(dx>=1&&dx<=n&&dy>=1&&dy<=m&&g[dx][dy]!='#'){
            if(flag==1){
                if(vis[dx][dy]!=1){
                    if(vis[dx][dy]==2){
                        dist[s.first][s.second]+=dist[dx][dy],found=1;
                        return ;
                    }
                    vis[dx][dy]=1;Q.push({dx,dy});
                    dist[dx][dy]=dist[s.first][s.second]+1;
                }
            }
            else{
                if(vis[dx][dy]!=2){
                    if(vis[dx][dy]==1){
                        dist[s.first][s.second]+=dist[dx][dy],found=1;
                        return ;
                    }
                    vis[dx][dy]=2;Q.push({dx,dy});
                    dist[dx][dy]=dist[s.first][s.second]+1;
                }
            }
        }
    }
}
```

• 双向广搜（DBFS）

✓ 选择结点个数较少的那个方向先扩展

代码链接

```
int dbfs(){
    int flag; PII temp;
    memset(vis,0,sizeof(vis));
    while(Q1.size()>0)   Q1.pop();
    while(Q2.size()>0)   Q2.pop();
    vis[1][1]=1,vis[n][m]=2;
    Q1.push({1,1});Q2.push({n,m});//Q2.push(make_pair{n,m});
    dist[1][1]=dist[n][m]=0;
    while(Q1.size()>0&&Q2.size()>0){
        if(Q1.size()>Q2.size())      temp=Q2.front(),Q2.pop(),flag=2;
        else
            temp=Q1.front(),Q1.pop(),flag=1;
        for(int i=0;i<=3;i++){
            int dx=temp.first+dir[i][0],dy=temp.second+dir[i][1];
            if(dx>=1&&dx<=n&&dy>=1&&dy<=m&&g[dx][dy]!='#'){
                if(vis[dx][dy]==0){
                    dist[dx][dy]=dist[temp.first][temp.second]+1,vis[dx][dy]=vis[temp.first][temp.second];
                    if(flag==1)      Q1.push({dx,dy});
                    else Q2.push({dx,dy});
                }
                else if(vis[temp.first][temp.second]+vis[dx][dy]==3)//一个为1,一个为2
                    return dist[temp.first][temp.second]+dist[dx][dy]+2;
            }
        }
    }
    return -1;
}
```

- 扩展域并查集（种类并查集）

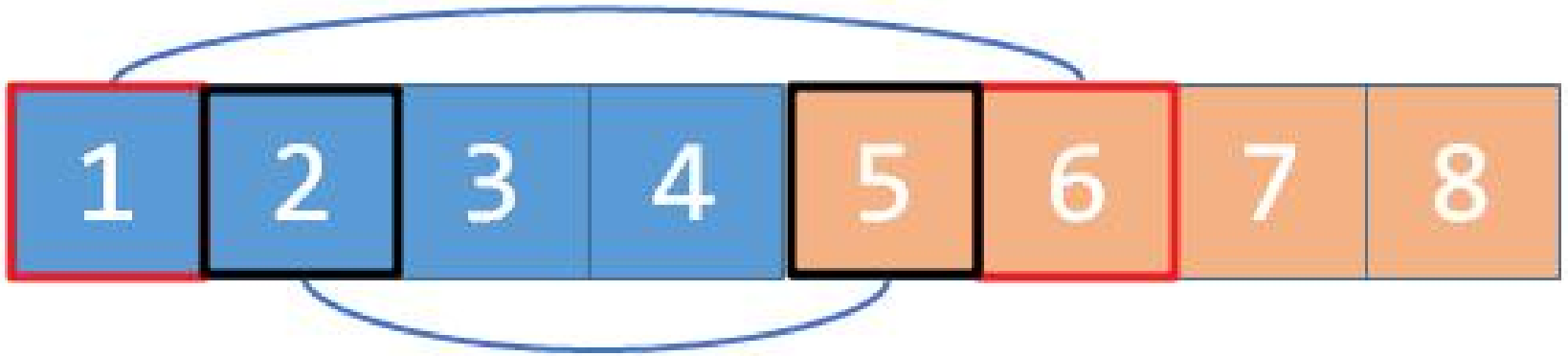
✓如果说一般的并查集，维护的是**等价、连通**关系，例如**朋友的朋友是朋友**。那么扩展域并查集(种类并查集)，维护的就是**对立**关系：**敌人的敌人是朋友**，或者更宽泛的说，是**多个种类集合间的一种循环对称的关系。**

✓常见的做法是将**原并查集扩大一倍规模**，并划分为两个种类。在同种类的并查集中合并，和原始的并查集没什么区别，仍然表达 他们是朋友 这个含义。在不同种类的并查集中进行合并，表达的则是 他们是敌人 这个含义。

- 扩展域并查集（种类并查集）

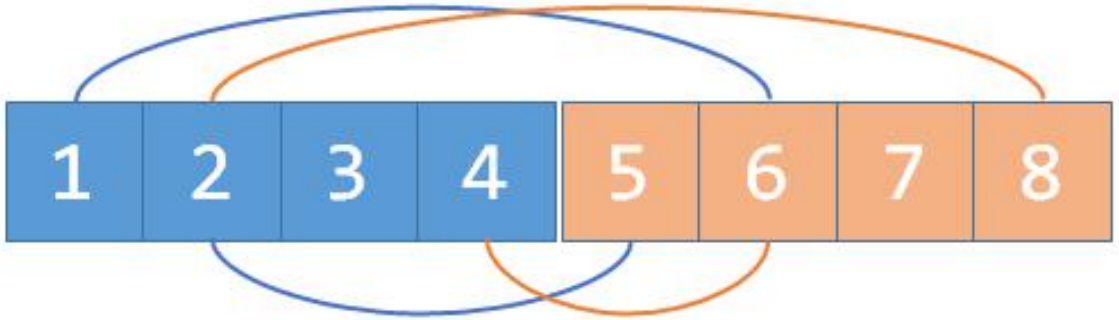
例如，要维护 4 个元素的种类并查集，要开 8 个单位的空间：

用 1-4 维护**朋友**关系，5-8 维护**敌人**关系。如果我们现在知道 1, 2 是敌人，该怎样做呢？我们用新的 merge 函数，$merge(1,2+n),merge(1+n,2)$。n 在这里等于4，对于编号为 i 的元素，i + n 是它的敌人。于是这两句表示**1是2的敌人，2是1的敌人**，如下图：



- 扩展域并查集（种类并查集）

现在我们知道 2, 4 是敌人，那么 $merge(2,4+n),merge(2+n,4)$ ；



于是，2, 4 是敌人，1, 2 是敌人。所以**有敌人的敌人就是朋友**， 1, 4 是朋友——通过 2 + n 这个元素将 1, 4 间接地连接起来了。**1, 4 现在在同一个集合之中**，表示它们是同一类人。

上述就是种类并查集的基本工作原理。我们把 1~n 看做是原本的并查集，n+1~2n 看做是表示敌对关系的并查集。

- 扩展域并查集（种类并查集）

例题

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`。

- 扩展域并查集（种类并查集)

输入 #1 复制

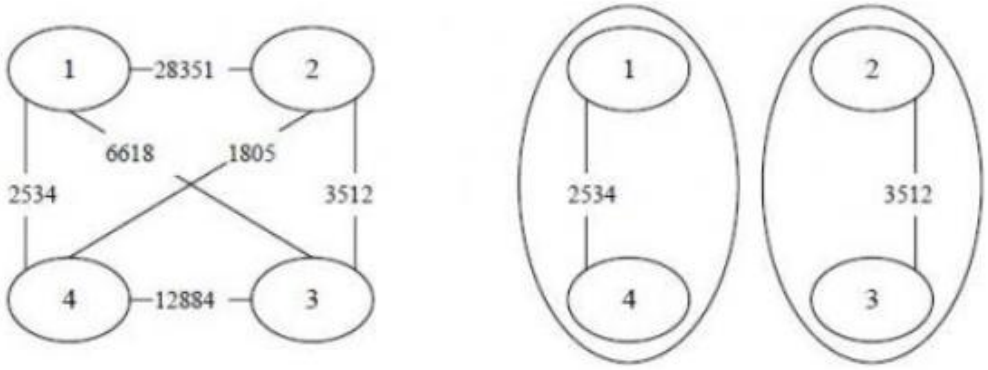
```
4 6
1 4 2534
2 3 3512
1 2 28351
1 3 6618
2 4 1805
3 4 12884
```

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。



代码链接

- 扩展域并查集（种类并查集）

```
#include <iostream>
#include <algorithm>
using namespace std;
const int MAXN=20010,MAXM=100010;
int n,m,flag=1,fa[2*MAXN];
struct node{
    int a;
    int b;
    int val;
}s[MAXM];
int cmp(node a,node b){
    return a.val>b.val;
}
void init(){
    for(int i=1;i<=2*n;i++)  fa[i]=i;
}
int find(int x)
{
    if(fa[x]!=x)
        fa[x]=find(fa[x]);
    return fa[x];
}
```

```
int main()
{
    scanf("%d %d",&n,&m);
    init();//并查集的初始化
    for(int i=1;i<=m;i++)
        scanf("%d %d %d",&s[i].a,&s[i].b,&s[i].val);
    sort(s+1,s+m+1,cmp);//排序
    for(int i=1;i<=m;i++){
        int fx=find(s[i].a),fy=find(s[i].b);//寻找父结点
        if(fx==fy){//如果在同一个并查集中，说明冲突爆发
            flag=0;
            printf("%lld\n",s[i].val);
            break;
        }//s[i].a和s[i].b是敌人(扩展并查集的应用)
        fa[fx]=find(s[i].b+n);
        fa[fy]=find(s[i].a+n);
    }

    if(flag==1)
        printf("0\n");

    return 0;
}
```

- 带权并查集

  ✓带权并查集与普通并查集的区别在于，其每条边/每个点都有权值。因此，我们需要额外用一个数组来维护每个点x到其当前祖先fa[x]的权值和，并在路径压缩的时候维护。

  ✓后面以一个例题理解带权并查集

- 带权并查集

例题

有一个划分为 N 列的星际战场，各列依次编号为 $1, 2, \ldots, N$。

有 N 艘战舰，也依次编号为 $1, 2, \ldots, N$，其中第 i 号战舰处于第 i 列。

有 T 条指令，每条指令格式为以下两种之一：

1. `M i j`，表示让第 i 号战舰所在列的全部战舰保持原有顺序，接在第 j 号战舰所在列的尾部。
2. `C i j`，表示询问第 i 号战舰与第 j 号战舰当前是否处于同一列中，如果在同一列中，它们之间间隔了多少艘战舰。

现在需要你编写一个程序，处理一系列的指令。

输入格式

第一行包含整数 T，表示共有 T 条指令。

接下来 T 行，每行一个指令，指令有两种形式：`M i j` 或 `C i j`。

其中 M 和 C 为大写字母表示指令类型，i 和 j 为整数，表示指令涉及的战舰编号。

输出格式

你的程序应当依次对输入的每一条指令进行分析和处理：

如果是 `M i j` 形式，则表示舰队排列发生了变化，你的程序要注意到这一点，但是不要输出任何信息；

如果是 `C i j` 形式，你的程序要输出一行，仅包含一个整数，表示在同一列上，第 i 号战舰与第 j 号战舰之间布置的战舰数目，如果第 i 号战舰与第 j 号战舰当前不在同一列上，则输出 -1。

- 带权并查集

输入 #1 复制

```
4
M 2 3
C 1 2
M 2 4
C 4 2
```

输出 #1 复制

```
-1
1
```

说明/提示

战舰位置图：表格中阿拉伯数字表示战舰编号

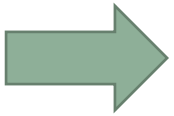
【样例说明】

战舰位置图：表格中阿拉伯数字表示战舰编号

	第一列	第二列	第三列	第四列	……
初始时	1	2	3	4	……
M 2 3	1		3 2	4	……
C 1 2	1 号战舰与 2 号战舰不在同一列，因此输出-1				
M 2 4	1			4 3 2	……
C 4 2	4 号战舰与 2 号战舰之间仅布置了一艘战舰，编号为 3，输出 1				

- 带权并查集
  - 一条“链”也是一棵树，只不过是树的特殊形态，因此可以把**每一列战舰看作一个集合**，用并查集维护，最初，N个战舰构成N个独立的集合。
  - 在没有路径压缩的情况下，fa[x]表示排在第x号战舰前面的那个战舰的编号。一个集合的代表就是位于最前边的战舰。另外，让树上每条边带权值1，这样**树上两点之间的距离减1就是两者之间间隔的战舰数量**。
  - 在考虑路径压缩的情况下，额外建立一个数组d，d[x]记录战舰x与fa[x]之间的边的权值。在路径压缩把x直接指向树根的同时，我们把d[x]更新为从x到树根的路径上的所有边权之和。对一般并查集的find()函数稍加修改，即可实现对d数组的维护：

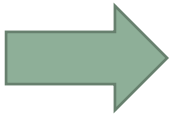
```
int find(int x)
{
    if(fa[x]!=x)
        fa[x]=find(fa[x]);
    return fa[x];
}
```



```
int find(int x)
{
    if(x==fa[x])    return x;
    int root=find(fa[x]);//递归计算集合代表
    d[x]+=d[fa[x]];//维护d数组,对边权求和
    return fa[x]=root;//路径压缩
}
```

- 带权并查集
  - 当接收到一个 C x y指令时，分别执行find(x)和find(y)完成查询和路径压缩。当二者的返回值相同，说明x和y处于同一列。因为x和y此时都已经指向树根,所以d[x]保存了位于x之前的战舰数量,d[y]保存了位于y之前的战舰数量，**两者之差的绝对值再减1**，就是x和y之间间隔的战舰数量。
  - 当接收到M x y指令时，把x的树根作为y的树根的子结点，连接的新边的权值应该设为合并之前集合y的大小(根据题意，集合y中的全部战舰都排在集合x之前)。因此，还需要一个siz数组在每个树根上记录集合大小。对一般并查集的join()函数稍加修改，即可实现对d数组的维护

```
void join(int x,int y)
{
    int fx=find(x),fy=find(y);
    if(fx!=fy)
        fa[fx]=fy;
}
```



```
void join(int x,int y)
{
    int fx=find(x),fy=find(y);
    if(fx!=fy)
    {
        fa[fx]=fy,d[fx]=siz[fy];
        siz[fy]+=siz[fx];
    }
}
```

- 带权并查集

```cpp
#include <iostream>
#include <algorithm>
using namespace std;
const int MAXN=30010;
int t;
int fa[MAXN],d[MAXN],siz[MAXN];

void init(){
    for(int i=1;i<MAXN;i++)
        fa[i]=i,siz[i]=1;//初始化时siz每个集合只有1个
}
int find(int x){
    if(x==fa[x])    return x;
    int root=find(fa[x]);//递归计算集合代表
    d[x]+=d[fa[x]];//维护d数组,对边权求和
    return fa[x]=root;//路径压缩
}
void join(int x,int y){
    int fx=find(x),fy=find(y);
    if(fx!=fy){
        fa[fx]=fy,d[fx]=siz[fy];
        siz[fy]+=siz[fx];
    }
}
```

[代码链接]

- 带权并查集

```
int main(){
    init();
    scanf("%d\n",&t);//此处换行相当于getchar();
    while(t--){
        int a,b;
        char op;
        scanf("%c %d %d ",&op,&a,&b);//注意输入字符的处理
        if(op=='M')//合并
            join(a,b);
        else//查询{
            if(a==b)      printf("0\n");
            else if(find(a)==find(b))
                printf("%d\n",abs(d[a]-d[b])-1);
            else printf("-1\n");
        }
    }
    return 0;
}
```

扩展： 食物链——考虑分别使用扩展域并查集和权值并查集解决

题目描述

复制Markdown 展开

动物王国中有三类动物 A,B,C，这三类动物的食物链构成了有趣的环形。A 吃 B，B 吃 C，C 吃 A。

现有 N 个动物，以 1 - N 编号。每个动物都是 A,B,C 中的一种，但是我们并不知道它到底是哪一种。

有人用两种说法对这 N 个动物所构成的食物链关系进行描述：

- 第一种说法是 `1 X Y`，表示 X 和 Y 是同类。
- 第二种说法是 `2 X Y`，表示 X 吃 Y 。

此人对 N 个动物，用上述两种说法，一句接一句地说出 K 句话，这 K 句话有的是真的，有的是假的。当一句话满足下列三条之一时，这句话就是假话，否则就是真话。

- 当前的话与前面的某些真的话冲突，就是假话
- 当前的话中 X 或 Y 比 N 大，就是假话
- 当前的话表示 X 吃 X，就是假话

你的任务是根据给定的 N 和 K 句话，输出假话的总数。

输入格式

第一行两个整数，N，K，表示有 N 个动物，K 句话。

第二行开始每行一句话（按照题目要求，见样例）

输出格式

一行，一个整数，表示假话的总数。

扩展： 食物链——考虑分别使用扩展域并查集和权值并查集解决

输入 #1 复制

```
100 7
1 101 1
2 1 2
2 2 3
2 3 3
1 1 3
2 3 1
1 5 5
```

输出 #1 复制

```
3
```

扩展： 食物链——考虑分别使用扩展域并查集和权值并查集解决

扩展域并查集

代码链接

把每个动物x拆成三个结点，同类域xself,捕食域xeat和天敌域xenemy

- x与y是同类：x的同类域y的同类一样,x捕食的物种与y捕食的物种,x的天敌与y的天敌一样,此时,合并xself与yself,xeat与yeat,xenemy与yenemy;
- x吃y:x捕食的物种是y的同类,x的同类都是y的天敌,且食物链是长度为3的环形,所以x的天敌就是y捕食的物种(x --> y --> z --> x),此时合并xeat与yself,xself与yenemy,xenemy与yeat

处理矛盾问题：

x与y是同类的矛盾：

1.xeat与yself在同一集合,说明x吃y

2.xself与yeat在同一集合,说明y吃x

x吃y矛盾:

1.xself和yself在同一集合,说明x与y是同类

2.yeat和xself在同一集合,说明y吃x

扩展： 食物链——考虑分别使用扩展域并查集和权值并查集解决

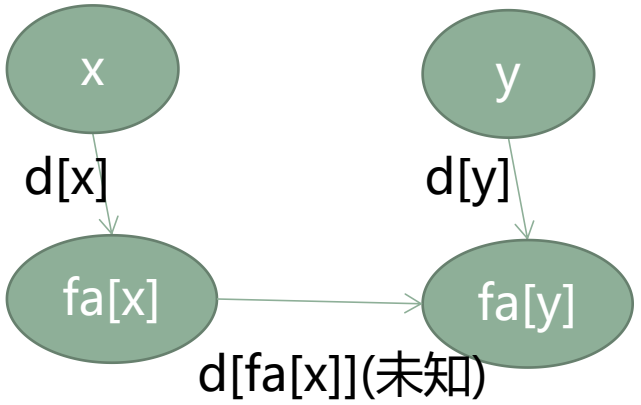
权值并查集

题目说有三类动物，形成了一个环形，A吃B，B吃C，C吃A，这就和数学中的取模很相似，0->1->2->3(3即是0)
我们采用数组d[i]来代表i节点到父节点的代数，通过d mod 3转化为0,1,2三代的关系。

d[x] mod 3 == d[y] mod 3　对应X与Y为同类，具体可以是，同为0代，同为1代，同为2代
(d[x]+1) mod 3 == d[y] mod 3 对应X吃Y，具体可以有三种情况
(d[x]+2) mod 3 == d[y] mod 3对应X被Y吃，同上

至此我们已经判断出X和Y的关系，也能很容易判断真话假话。

x,y为同类



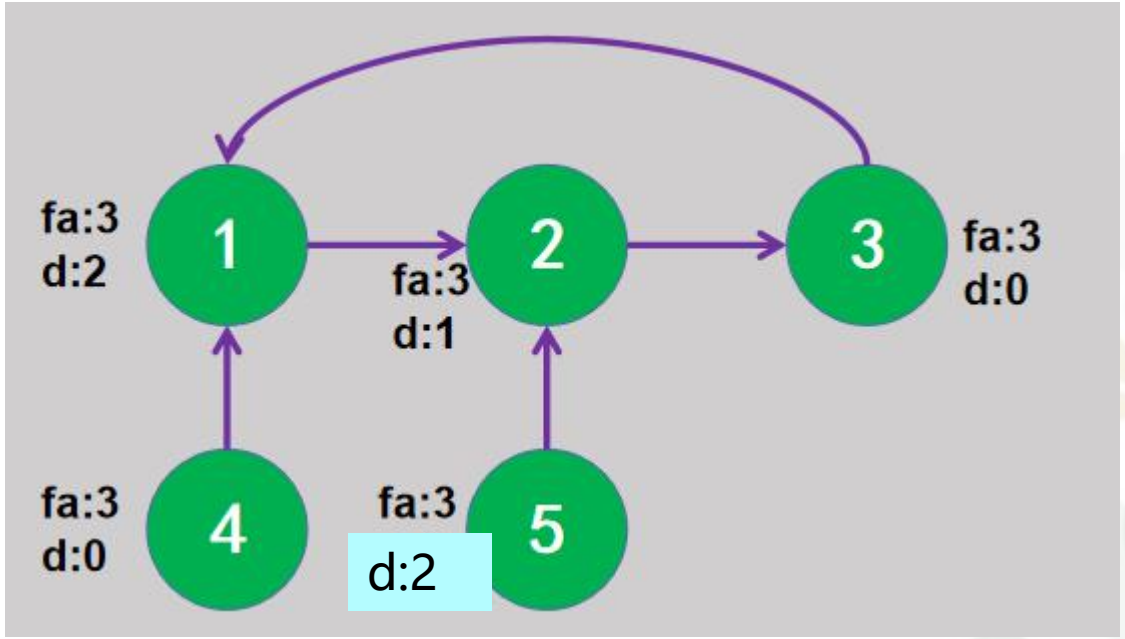
$$(d[x]+d[fa[x]]-d[y])\%3=0$$

$$d[fa[x]]=(d[y]-d[x])\%3$$

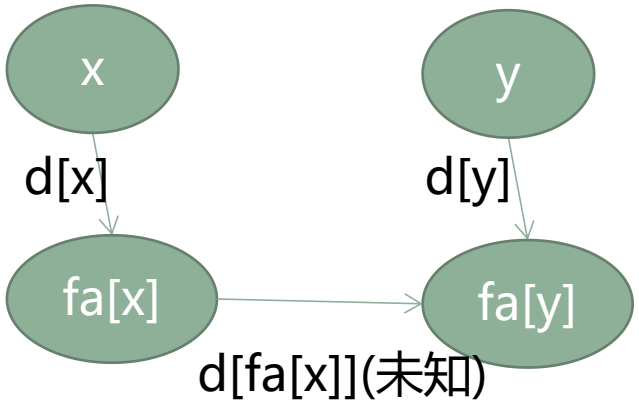
扩展： 食物链——考虑分别使用扩展域并查集和权值并查集解决

代码链接

权值并查集



x吃y



$$(d[x]+d[fa[x]]-d[y])\%3=1$$

$$d[fa[x]]=(d[y]-d[x]+1)\%3$$