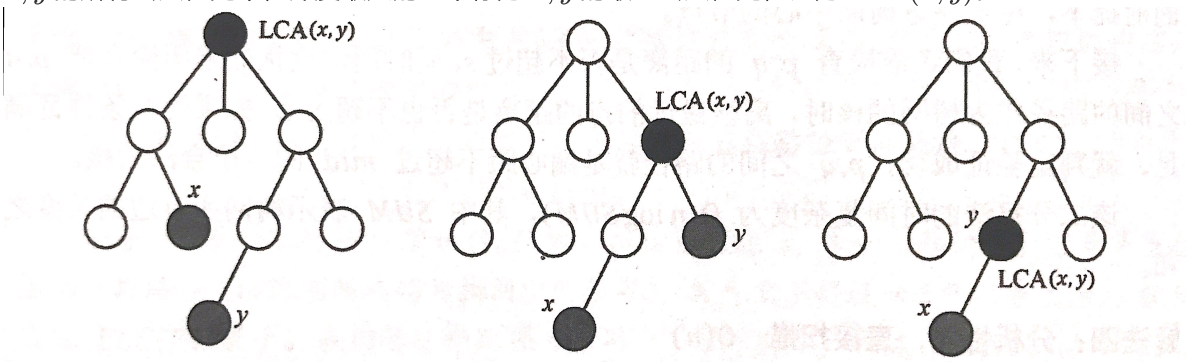
给定一棵有根树，若节点 z 既是节点 x 的祖先，也是节点 y 的祖先，则称 z 是 x,y 的公共祖先。在 x,y 的所有公共祖先中，深度最大的一个称为 x,y 的最近公共祖先，记为 $\mathrm{LCA}(x,y)$。



$\mathrm{LCA}(x,y)$ 是 x 到根的路径与 y 到根的路径的交会点。它也是 x 与 y 之间的路径上深度最小的节点。求最近公共祖先的方法通常有三种:

向上标记法

从 x 向上走到根节点，并标记所有经过的节点。

从 y 向上走到根节点，当第一次遇到已标记的节点时，就找到了 $LCA(x,y)$。

对于每个询问，向上标记法的时间复杂度最坏为 $O(n)$。

树上倍增法

树上倍增法是一个很重要的算法。除了求 LCA 之外，它在很多问题中都有广泛应用。设 $F[x][k]$ 表示 x 的 2^k 辈祖先，即从 x 向根节点走 2^k 步到达的节点。特别地，若该节点不存在，则令 $F[x][k]=0$。$F[x][0]$ 就是 x 的父节点。除此之外，$\forall k\in[1,\log\ n], F[x][k]=F[F[x][k-1],k-1]$ 。

这类似于一个动态规划的过程，“阶段” 就是节点的深度。因此，我们可以对树进行广度优先遍历，按照层次顺序，在节点入队之前，计算它在 F 数组中相应的值。

以上部分是预处理，时间复杂度为 $O(n\log n)$，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 $O(\log\ n)$。

基于 F 数组计算 $\mathrm{LCA}(x,y)$ 分为以下几步:

1.设 $d[x]$ 表示 x 的深度。不妨设 $d[x]\geq d[y]$ （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。

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，检查到达的节点是否比 y 深。在每次检查中，若是，则令 $x=F[x][k]$。

3.若此时 $x=y$，说明已经找到了 LCA，LCA 就等于 y。

这就是上面的图中的第三种情况。

4.用二进制拆分思想，把 x,y 同时向上调整，并保持深度一致且二者不相会。

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 $F[x][k]\neq F[y][k]$（即仍未相会），则令 $x=F[x][k], y=F[y][k]$。

5. 此时 x,y 必定只差一步就相会了，它们的父节点 $F[x][0]$ 就是 LCA。

多次查询树上两点之间的距离，时间复杂度为 $O((n+m)\log n)$。

【例题】HDU-2586 How far away ?

给定一棵 n 个点的树，m 次询问，每次询问输出 x, y 两点之间的最短距离 $(T \le 10, \le n \le 40000, 1 \le m \le 200)$。

代码

```
#include<bits/stdc++.h>
using namespace std;
const int SIZE=50010;
int f[SIZE][20],d[SIZE],dist[SIZE];
struct Edge
{
    int to;
    int dis;
    int Next;
}edge[SIZE<<1];
int n,m,head[SIZE],num_edge,t;
queue<int> Q;
void add_edge(int from,int to,int dis)
{
    edge[++num_edge].to=to;
    edge[num_edge].dis=dis;
    edge[num_edge].Next=head[from];
    head[from]=num_edge;
}
void bfs()//预处理
{
    Q.push(1);
    d[1]=1;
    while(!Q.empty())
    {
        int x=Q.front();
        Q.pop();
        for(int i=head[x];i!=-1;i=edge[i].Next)
        {
            int y=edge[i].to;
            if(d[y])
                continue;
            d[y]=d[x]+1;
            dist[y]=dist[x]+edge[i].dis;
            f[y][0]=x;
            for(int j=1;j<=t;j++)
                f[y][j]=f[f[y][j-1]][j-1];
            Q.push(y);
        }
    }
}
int lca(int x,int y)
{
    if(d[x]>d[y])
        swap(x,y);
    for(int i=t;i>=0;i--)
        if(d[f[y][i]]>=d[x])
            y=f[y][i];
    if(x==y)
        return x;
```

```
    for(int i=t;i>=0;i--)
    {
        if(f[x][i]!=f[y][i])
        {
            x=f[x][i];
            y=f[y][i];
        }
    }
    return f[x][0];
}
int main()
{
    int T;
    cin>>T;
    while(T--)
    {
        cin>>n>>m;
        t=(int)(log(n)/log(2))+1;
        for(int i=1;i<=n;i++)
        {
            head[i]=-1;
            d[i]=0;
        }
        num_edge=0;
        for(int i=1;i<=n-1;i++)
        {
            int x,y,val;
            scanf("%d%d%d",&x,&y,&val);
            add_edge(x,y,val);
            add_edge(y,x,val);
        }
        bfs();
        for(int i=1;i<=m;i++)
        {
            int x,y;
            scanf("%d%d",&x,&y);
            printf("%d\n",dist[x]+dist[y]-2*dist[lca(x,y)]);
        }
    }
    return 0;
}
```

LCA的Tarjan算法

Tarjan 算法本质上是使用并查集对 “向上标记法” 的优化。它是一个离线算法，需要把 m 个询问一次性读入，统一计算，最后统一输出。时间复杂度为 $O(n+m)$。

在深度优先遍历的任意时刻，树中节点分为三类：

1.已经访问完毕并且回溯的节点。在这些节点上标记一个整数 2。

2.已经开始递归，但尚未回溯的节点。这些节点就是当前正在访问的节点 x 以及 x 的祖先。在这些节点上标记一个整数 1。

3.尚未访问的节点。这些节点没有标记。

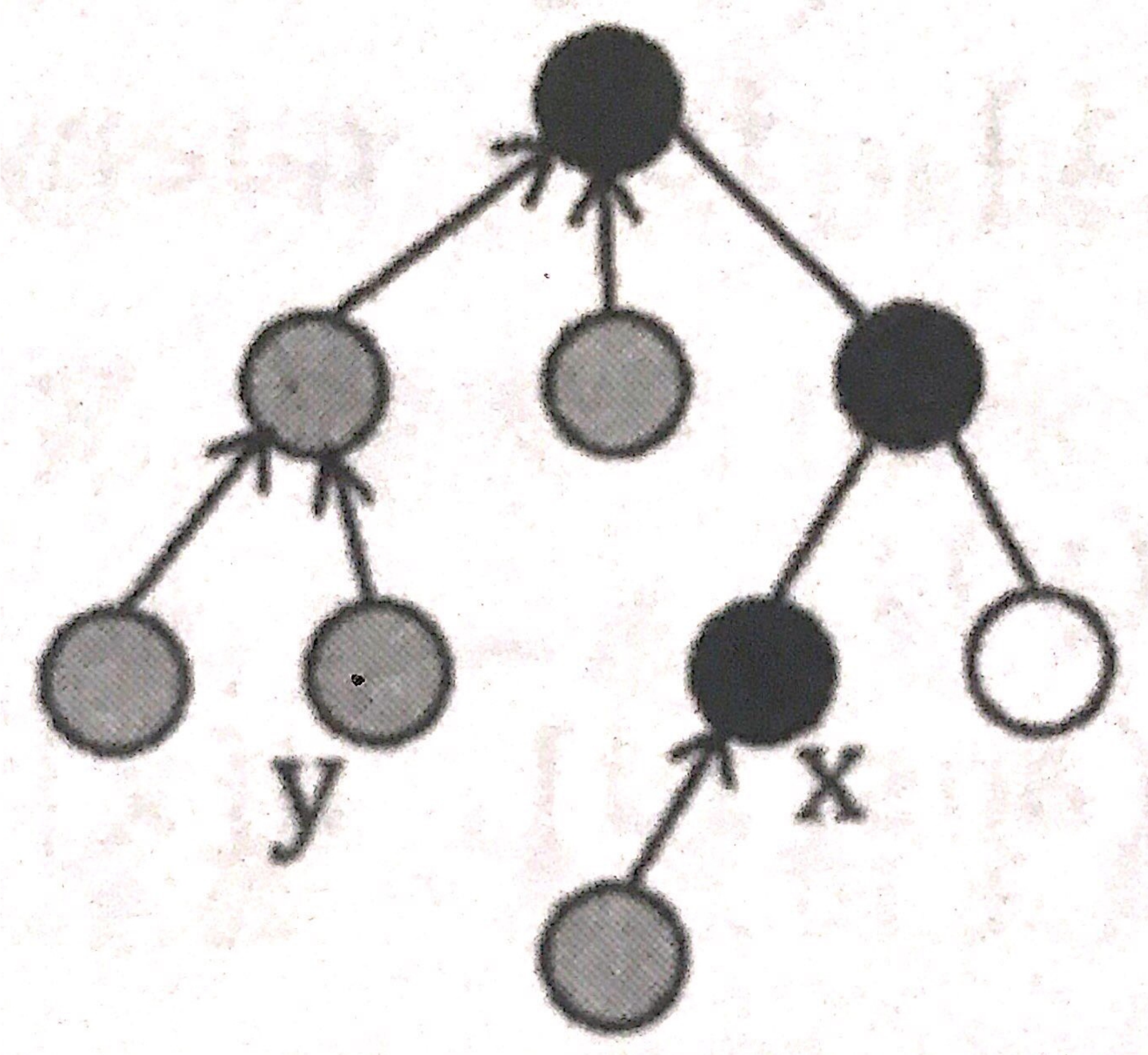
对于正在访问的节点 x，它到根节点的路径已经标记为 1。若 y 是已经访问完毕并且回溯的节点，则 $\mathrm{LCA}(x,y)$ 就是从 y 向上走到根，第一个遇到的标记为 1 的节点。

可以利用并查集进行优化，当一个节点获得整数 2 的标记时，把它所在的集合合并到它的父节点所在的集合中（合并时它的父节点标记一定为 1，且单独构成一个集合）。

这相当于每个完成回溯的节点都有一个指针指向它的父节点，只需查询 y 所在集合的代表元素（并查集的 get 操作），就等价于从 y 向上一直走到一个开始递归但尚未回溯的节点（具有标记 1），即 LCA(x,y)。

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get(y)函数的结果）。



时间复杂度为 $O(n+m)$。

```
#include<bits/stdc++.h>
using namespace std;
const int SIZE=50010;
int head[SIZE];
int fa[SIZE],d[SIZE],vis[SIZE],lca[SIZE],ans[SIZE];
vector<int> query[SIZE],query_id[SIZE];
int T,n,m,num_edge,t;
struct Edge
{
    int to;
    int dis;
    int Next;
```

```
}edge[2*SIZE];
void add_edge(int from,int to,int dis)
{
    edge[++num_edge].to=to;
    edge[num_edge].dis=dis;
    edge[num_edge].Next=head[from];
    head[from]=num_edge;
}
void add_query(int x,int y,int id)
{
    query[x].push_back(y),query_id[x].push_back(id);
    query[y].push_back(x),query_id[y].push_back(id);
}
int get(int x)
{
    if(x==fa[x])
        return x;
    else
        return fa[x]=get(fa[x]);
}
void tarjan(int x)
{
    vis[x]=1;
    for(int i=head[x];i;i=edge[i].Next)
    {
        int y=edge[i].to;
        if(vis[y])
            continue;
        d[y]=d[x]+edge[i].dis;
        tarjan(y);
        fa[y]=x;
    }
    for(int i=0;i<query[x].size();i++)
    {
        int y=query[x][i],id=query_id[x][i];
        if(vis[y]==2)
        {
            int lca=get(y);
            ans[id]=min(ans[id],d[x]+d[y]-2*d[lca]);
        }
    }
    vis[x]=2;
}
int main()
{
    cin>>T;
    while(T--)
    {
        cin>>n>>m;
        for(int i=1;i<=n;i++)
        {
            head[i]=0;
            fa[i]=i;
            vis[i]=0;
            query[i].clear();
            query_id[i].clear();
        }
        num_edge=0;
```

```
        for(int i=1;i<=n-1;i++)
        {
            int x,y,z;
            scanf("%d %d %d",&x,&y,&z);
            add_edge(x,y,z);
            add_edge(y,x,z);
        }
        for(int i=1;i<=m;i++)
        {
            int x,y;
            scanf("%d %d",&x,&y);
            if(x==y)
                ans[i]=0;
            else
            {
                add_query(x,y,i);
                ans[i]=1<<30;
            }
        }
        tarjan(1);
        for(int i=1;i<=m;i++)
            printf("%d\n",ans[i]);
    }
    return 0;
}
```

树上差分

在“前缀和与差分”中，我们定义了一个序列的前缀和与差分序列，并通过差分技巧，把“区间”的增减转化为“左端点加 1，右端点减 1”。根据“差分序列的前缀和是原序列”这一原理，在树上可以进行类似的简化，其中“区间操作”对应为“路径操作”，“前缀和”对应为“子树和”。