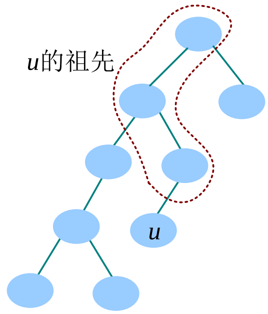
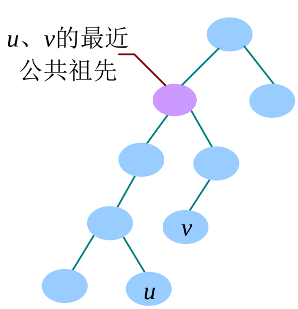
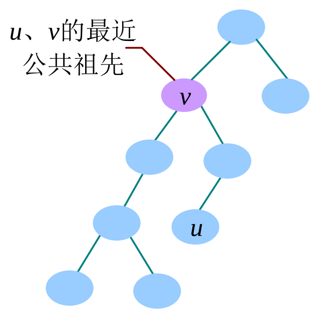
2.2　最近公共祖先 LCA

最近公共祖先（Lowest Common Ancestors，LCA）指有根树中距离两个节点最近的公共祖先。祖先指从当前节点到树根路径上的所有节点。

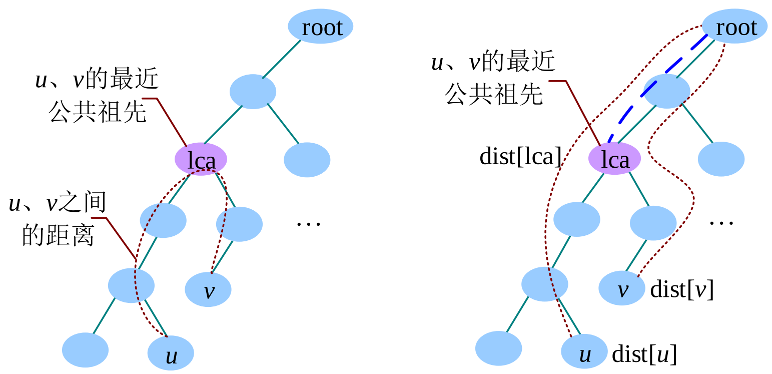


u 和 *v* 的公共祖先指一个节点既是 *u* 的祖先，又是 *v* 的祖先。*u* 和 *v* 的最近公共祖先指距离 *u* 和 *v* 最近的公共祖先。若 *v* 是 *u* 的祖先，则 *u* 和 *v* 的最近公共祖先是 *v*。





可以使用 LCA 求解树上任意两点之间的距离。求 *u* 和 *v* 之间的距离时，若 *u* 和 *v* 的最近公共祖先为 lca，则 *u* 和 *v* 之间的距离为 *u* 到树根的距离加上 *v* 到树根的距离减去 2 倍的 lca 到树根的距离：dist[*u*]+dist[*v*]−2×dist[lca]。



求解 LCA 的方法有很多，包括暴力搜索法、树上倍增法、在线 RMQ 算法、离线 Tarjan 算法和树链剖分。

在线算法：以序列化方式一个一个地处理输入，也就是说，在开始时并不需要知道所有输入，在解决一个问题后立即输出结果。

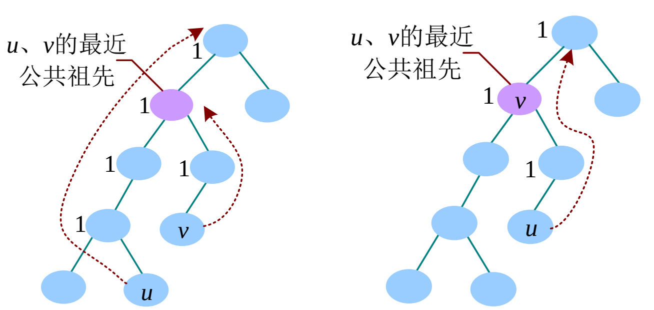
离线算法：在开始时已知问题的所有输入数据，可以一次性回答所有问题。

原理 1　暴力搜索法

暴力搜索法有两种：向上标记法和同步前进法。

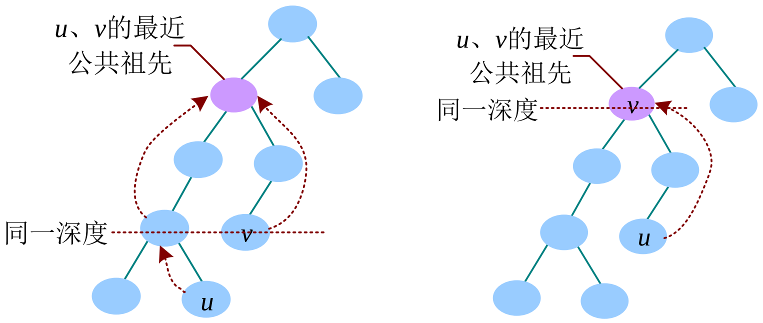
1．向上标记法

从 u 向上一直到根节点，标记所有经过的节点；若 v 已被标记，则 v 节点为 LCA(u, v)；否则 v 也向上走，第 1 次遇到已标记的节点时，该节点为 LCA(u, v)。



2．同步前进法

将 u、v 中较深的节点向上走到和深度较浅的节点同一深度，然后两个点一起向上走，直到走到同一个节点，该节点就是 u、v 的最近公共祖先，记作 LCA(u,v)。若较深的节点 u 到达 v 的同一深度时，那个节点正好是 v，则 v 节点为 LCA(u, v)。



3．算法分析

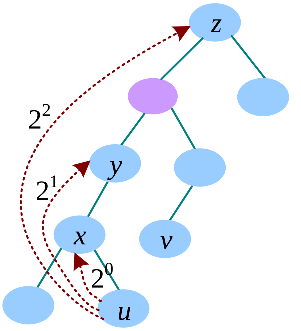
以暴力搜索法求解 LCA，两种方法的时间复杂度在最坏情况下均为 $O(n)$。

原理 2　树上倍增法

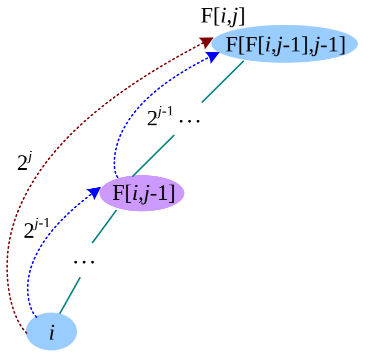
树上倍增法不仅可以解决 LCA 问题，还可以解决很多其他问题，掌握树上倍增法是很有必要的。

F[i, j]表示 i 的 2^j 辈祖先，即 i 节点向根节点走 2^j 步到达的节点。

u 节点向上走 2^0 步，则为 u 的父节点 x，F[u,0]=x；向上走 2^1 步，到达 y，F[u,1]=y；向上走 2^2 步，到达 z，F[u,2]=z；向上走 2^3 步，节点不存在，令 F[u,3]=0。



F[i, j]表示 i 的 2^j 辈祖先，即 i 节点向根节点走 2^j 步到达的节点。可以分两个步骤：i 节点先向根节点走 2^{j-1} 步得到 F[i, j–1]；再从 F[i, j–1]节点出发向根节点走 2^{j-1} 步，得到 F[F[i, j–1], j–1]，该节点为 F[i, j]。



递推公式：$F[i, j]=F[F[i, j-1], j-1]$，$i=1,2,\cdots n$，$j=0,1,2,\cdots k$，$2^k\leqslant n$，$k=\log_2 n$。

1．算法设计

（1）创建 ST。

（2）利用 ST 求解 LCA。

2．完美图解

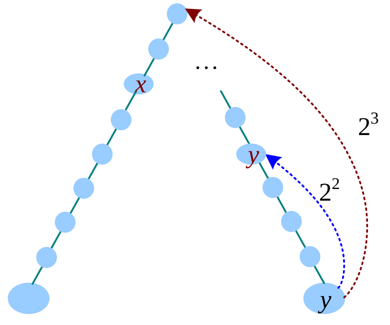
和前面暴力搜索中的同步前进法一样，先让深度大的节点 y 向上走到与 x 同一深度，然后 x、y 一起向上走。和暴力搜索不同的是，向上走是按照倍增思想走的，不是一步一步向上走的，因此速度较快。

问题一：怎么让深度大的节点 y 向上走到与 x 同一深度呢？

假设 y 的深度比 x 的深度大，需要 y 向上走到与 x 同一深度，k=3，则求解过程如下。

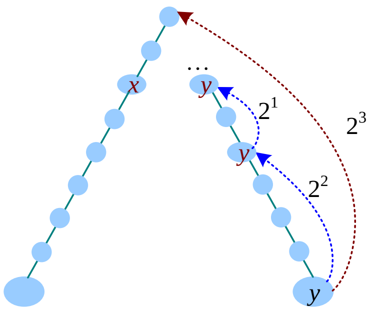
（1）y 向上走 2^3 步，到达的节点深度比 x 的深度小，什么也不做。

（2）减少增量，y 向上走 2^2 步，此时到达的节点深度比 x 的深度大，y 上移，$y=F[y][2]$。



（3）减少增量，y 向上走 2^1 步，此时到达的节点深度与 x 的深度相等，y 上移，$y=F[y][1]$。

（4）减少增量，y 向上走 2^0 步，到达的节点深度比 x 的深度小，什么也不做。此时 x、y 在同一深度。



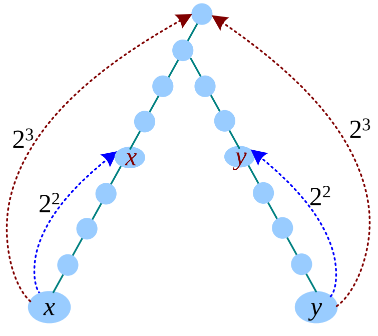
总结：按照增量递减的方式，到达的节点深度比 x 的深度小时，什么也不做；到达的节点深度大于或等于 x 的深度时，y 上移，直到增量为 0，此时 x、y 在同一深度。

问题二：x、y 一起向上走，怎么找最近的公共祖先呢？

假设 x、y 已到达同一深度，现在一起向上走，k=3，则其求解过程如下。

（1）x、y 同时向上走 2^3 步，到达的节点相同，什么也不做。

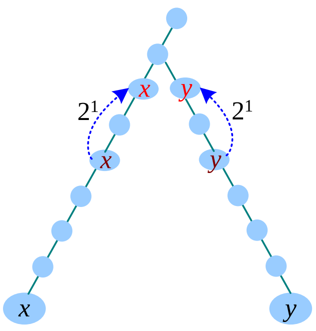
（2）减少增量，x、y 同时向上走 2^2 步，此时到达的节点不同，x、y 上移，x=F[x][2]，y=F[y][2]。

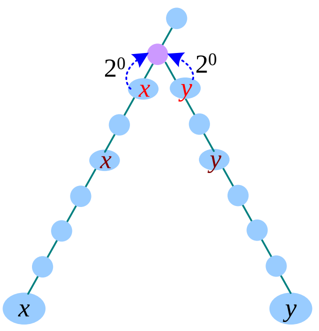


（3）减少增量，x、y 同时向上走 2^1 步，此时到达的节点不同，x、y 上移，x=F[x][1]，y=F[y][1]。

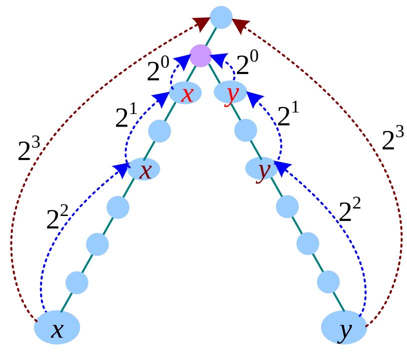
（4）减少增量，x、y 同时向上走 2^0 步，此时到达的节点相同，什么也不做。

此时 x、y 的父节点为最近公共祖先节点，即 LCA(x, y)=F[x][0]。





完整的求解过程如下图所示。



总结：按照增量递减的方式，到达的节点相同时，什么也不做；到达的节点不同时，同时上移，直到增量为 0。此时 x、y 的父节点为公共祖先节点。

3．算法实现

```
void ST_create(){//构造 ST
    for(int j=1;j<=k;j++)
        for(int i=1;i<=n;i++)//i 先走 2^(j-1)步到达 F[i][j-1]，再走 2^(j-1)步
            F[i][j]=F[F[i][j-1]][j-1];
}

int LCA_st_query(int x,int y) {//求 x、y 的最近公共祖先
    if(d[x]>d[y])//保证 x 的深度小于或等于 y
        swap(x,y);
    for(int i=k;i>=0;i--)//y 向上走到与 x 同一深度
        if(d[F[y][i]]>=d[x])
            y=F[y][i];
    if(x==y)
        return x;
    for(int i=k;i>=0;i--)//x、y 一起向上走
        if(F[x][i]!=F[y][i])
            x=F[x][i],y=F[y][i];
    return F[x][0];//返回 x 的父节点
}
```

4．算法分析

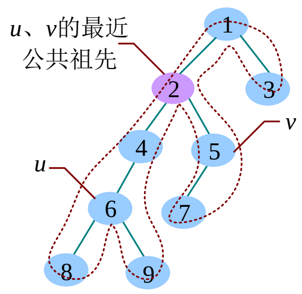
采用树上倍增法求解 LCA，创建 ST 需要 $O(n\log n)$时间，每次查询都需要 $O(\log n)$时间。一次建表、多次使用，该算法是基于倍增思想的动态规划，适用于多次查询的情况。若只有几次查询，则预处理需要 $O(n\log n)$时间，还不如暴力搜索快。

📖 原理 3　在线 RMQ 算法

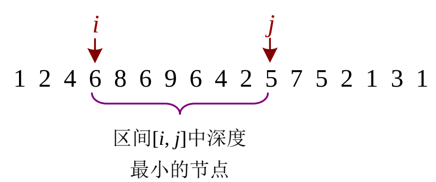
两个节点的 LCA 一定是两个节点之间欧拉序列中深度最小的节点，寻找深度最小值时可以使用 RMQ 算法。

1. 完美图解

欧拉序列指在深度遍历过程中把依次经过的节点记录下来，把回溯时经过的节点也记录下来，一个节点可能被记录多次，相当于从树根开始，一笔画出一个经过所有节点的回路。



该树的欧拉序列为 1 2 4 6 8 6 9 6 4 2 5 7 5 2 1 3 1，搜索时得到 6 和 5 首次出现的下标 i、j，然后查询该区间深度最小的节点，为 6 和 5 号节点的最近公共祖先。



2. 算法实现

（1）深度遍历，得到 3 个数组：首次出现的下标是 pos[]，深度遍历得到的欧拉序列是 seq[]，深度是 dep[]。

```
pos[u]=++tot;//u 首次出现的下标
seq[tot]=u;//dfs 遍历得到的欧拉序列
dep[tot]=d;//深度
void dfs(int u,int d) {//dfs 序
    vis[u]=true;
    pos[u]=++tot;//u 首次出现的下标
    seq[tot]=u;//dfs 遍历得到的欧拉序列
    dep[tot]=d;//深度
    for(int i=head[u];i;i=e[i].next){
        int v=e[i].to,w=e[i].c;
        if(vis[v])
```

```
            continue;
        dist[v]=dist[u]+w;
        dfs(v,d+1);
        seq[++tot]=u;//dfs 遍历序列
        dep[tot]=d;//深度
    }
}
```

（2）根据欧拉序列的深度，创建区间最值查询的 ST。F(i, j)表示[i, i+2^j–1]区间深度最小的节点下标。

```
void ST_create(){//创建 ST
    for(int i=1;i<=tot;i++)//初始化
        F[i][0]=i;//记录下标，不是最小深度
    int k=log2(tot);
    for(int j=1;j<=k;j++)
        for(int i=1;i<=tot-(1<<j)+1;i++)//tot-2^j+1
            if(dep[F[i][j-1]]<dep[F[i+(1<<(j-1))][j-1]])
                F[i][j]=F[i][j-1];
            else
                F[i][j]=F[i+(1<<(j-1))][j-1];
}
```

（3）查询[l, r]区间深度最小的节点下标，与 RMQ 区间查询类似。

```
int RMQ_query(int l,int r){//查询[l,r]的区间最值
    int k=log2(r-l+1);
    if(dep[F[l][k]]<dep[F[r-(1<<k)+1][k]])
        return  F[l][k];
    else
        return  F[r-(1<<k)+1][k];//返回深度最小的节点下标
}
```

（4）求 x、y 的最近公共祖先，先得到 x、y 首次出现在欧拉序列中的下标，然后查询该区间深度最小的节点的下标，根据下标读取欧拉序列的节点即可。

```
int LCA(int x,int y) {//求 x、y 的最近公共祖先
    int l=pos[x],r=pos[y];//读取第 1 次出现的下标
    if(l>r)
        swap(l,r);
    return seq[RMQ_query(l,r)];//返回节点
}
```

5. 算法分析

在线 RMQ 算法是基于倍增和 RMQ 的动态规划算法，其预处理包括深度遍历和创建 ST，

需要 $O(n\log n)$时间，每次查询都需要 $O(1)$时间。

注意：虽然都用到了 ST，但是在线 RMQ 算法中的 ST 和树上倍增算法中的 ST，其表达的含义是不同的，前者表示区间最值，后者表示向上走的步数。

原理 4　Tarjan 算法

这里的 Tarjan 算法是用于解决 LCA 问题的离线算法，在《算法训练营：海量图解+竞赛刷题（入门篇）》中会讲解求连通分量的 Tarjan 算法。在线算法指每读入一个查询（求一次 LCA 就叫作一次查询），都需要运行一次程序得到本次查询答案。若一次查询需要 $O(\log n)$时间，则 m 次查询需要 $O(m\log n)$时间。离线算法指首先读入所有查询，然后运行一次程序得到所有查询答案。Tarjan 算法利用并查集优越的时空复杂性，可以在 $O(n+m)$时间内解决 LCA 问题。

1. Tarjan 算法

（1）初始化集合号数组和访问数组，fa[i]=i，vis[i]=0。

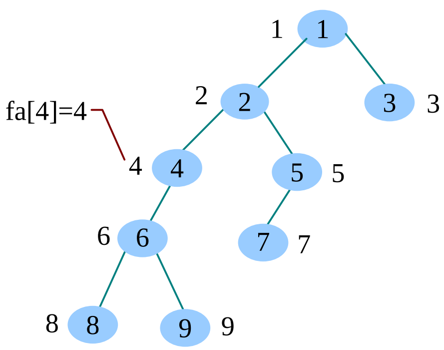
（2）从 u 出发深度优先遍历，标记 vis[u]=1，深度优先遍历 u 所有未被访问的邻接点，在遍历过程中更新距离，回退时更新集合号。

（3）当 u 的邻接点全部遍历完毕时，检查关于 u 的所有查询，若存在一个查询 u、v，而 vis[v]=1，则利用并查集查找 v 的祖宗，找到的节点就是 u、v 的最近公共祖先。

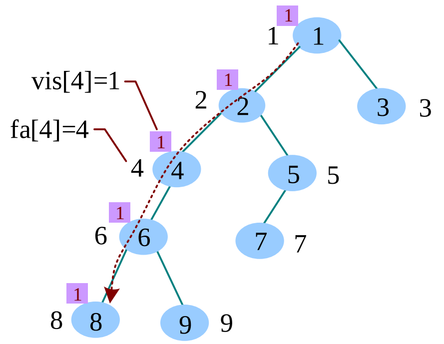
2. 完美图解

在树中求 5、6 的最近公共祖先，求解过程如下。

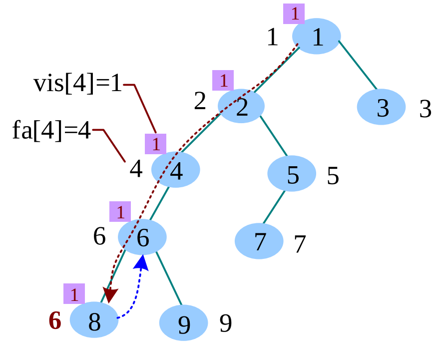
（1）初始化所有节点的集合号都等于自己，fa[i]=i，vis[i]=0。



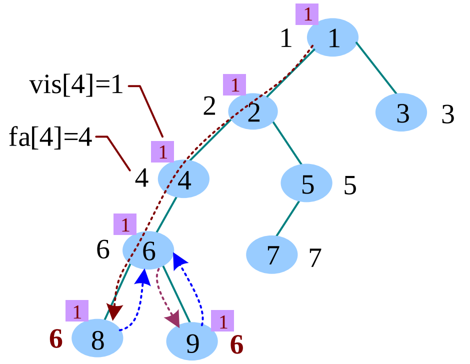
（2）从根节点开始深度优先遍历，在遍历过程中标记 vis[]=1。



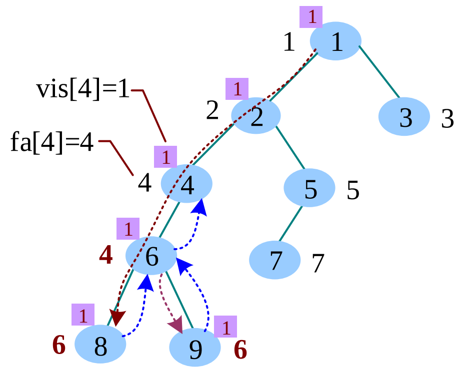
（3）8 号节点的邻接点已访问完毕，更新 fa[8]=6，没有 8 相关的查询，回退到 6。



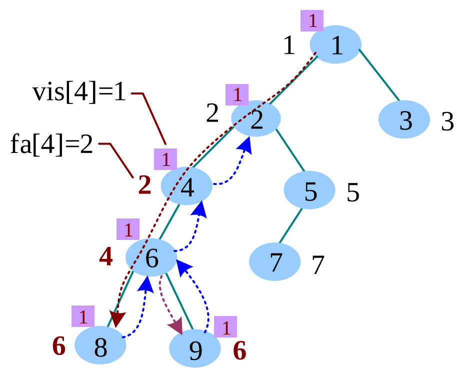
（4）遍历 6 号节点的下一个邻接点 9，标记 vis[9]=1，9 号节点的邻接点已访问完毕，更新 fa[9]=6，没有 9 相关的查询，回退到 6。



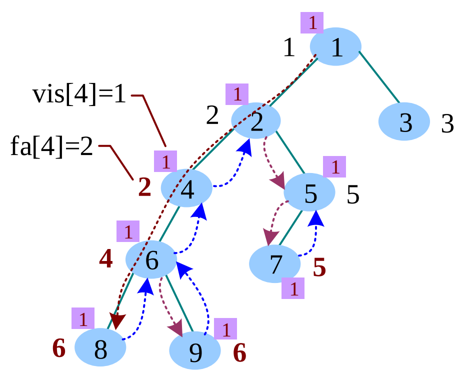
（5）6 号节点的邻接点已访问完毕，更新 fa[6]=4，有 6 相关的查询 5（查询 5 6），但是 vis[5]≠1，什么也不做，返回到 4。



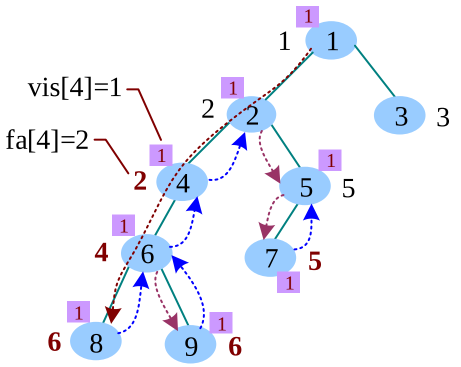
（6）4 号节点的邻接点已访问完毕，更新 fa[4]=2，没有 4 相关的查询，返回到 2。



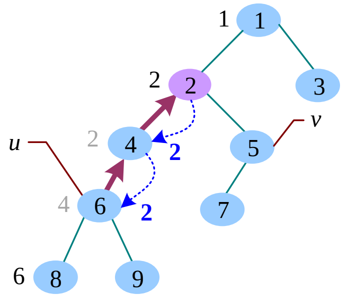
（7）遍历 2 号节点的下一个邻接点 5，标记 vis[5]=1，继续深度遍历到 7，标记 vis[7]=1，7 号节点的邻接点已访问完毕，更新 fa[7]=5，没有 7 相关的查询，回退到 5。



（8）5 号节点的邻接点已访问完毕，更新 fa[5]=2，有 5 相关的查询 6（查询 5 6），且 vis[6]=1，此时需要从 6 号节点开始使用并查集查找祖宗。



（9）从 6 号节点开始利用并查集查找祖宗的的过程如下。首先判断 6 的集合号 fa[6]=4，找 4 的集合号 fa[4]=2，找 2 的集合号 fa[2]=2，找到祖宗（集合号为其自身）后返回，并更新祖宗到当前节点路径上所有节点的集合号，即更新 6、4 的父节点 fa[4]=2，fa[6]=2，此时 fa[6]就是 5 和 6 的最近公共祖先。



总结：在当前节点 u 的邻接点已访问完毕时，检查 u 相关的所有查询 v，若 vis[v]≠1，则什么也不做；若 vis[v]=1，则利用并查集查找 v 的祖宗，lca(u,v)=fa[v]。实际上，u 的祖宗就是 u 向上查找第 1 个邻接点未访问完的节点，它的 fa[]还没有更新，仍满足 fa[i]=i，它就是 v 的祖宗。

3．算法实现

```
int find(int x){//并查集找祖宗
    if(x!=fa[x])
        fa[x]=find(fa[x]);
    return fa[x];
}

void tarjan(int u){//Tarjan 算法
    vis[u]=1;
    for(int i=head[u];i;i=e[i].next){
        int v=e[i].to,w=e[i].c;
        if(vis[v])
```

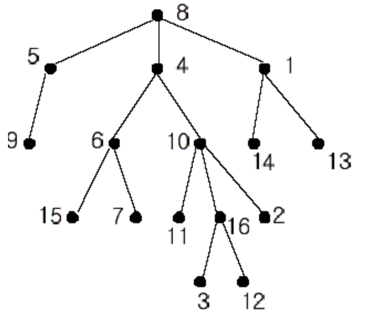
```
            continue;
        dis[v]=dis[u]+w;
        tarjan(v);
        fa[v]=u;
    }
    for(int i=0;i<query[u].size();i++){//u 相关的所有查询
        int v=query[u][i];
        int id=query_id[u][i];
        if(vis[y]){
            int lca=find(v);
            ans[id]=dis[u]+dis[v]-2*dis[lca];
        }
    }
}
```

4．算法分析

离线 Tarjan 算法用到了并查集的优越性，m 次查询的时间为 $O(n+m)$。

训练 1　最近公共祖先

题目描述（POJ1330）：一棵树如下图所示，每个节点都标有{1,2,⋯,16}的整数，节点 8 是树根。若节点 x 位于根和 y 之间的路径中，则 x 是 y 的祖先，节点也是自己的祖先。8、4、10 和 16 是 16 的祖先，8、4、6 和 7 是 7 的祖先。若 x 是 y 的祖先和 z 的祖先，则 x 被称为 y 和 z 的公共祖先，因此 8 和 4 是 16 和 7 的公共祖先。若 x 是 y 和 z 的公共祖先并且在它们的公共祖先中最接近 y 和 z，则 x 被称为 y 和 z 的最近公共祖先，16 和 7 的最近公共祖先是 4。若 y 是 z 的祖先，则 y 和 z 的最近公共祖先是 y，4 和 12 的最近公共祖先是 4。编写一个程序，找到树中两个不同节点的最近公共祖先。



输入：第 1 行包含一个整数 T，表示测试用例的数量。每个测试用例的第 1 行都包含整数 N（$2 \leqslant N \leqslant 10{,}000$），表示树中的节点数。节点用 1～$N$ 标记。接下来的 N–1 行，每行都包含一

对表示边的整数，第 1 个整数是第 2 个整数的父节点（有 N 个节点的树则恰好有 N–1 条边）。每个测试用例的最后一行都包含两个不同的整数，求其最近公共祖先。

输出：对每个测试用例，都单行输出两个节点的最近公共祖先。

输入样例
```
2
16
1 14
8 5
10 16
5 9
4 6
8 4
4 10
1 13
6 15
10 11
6 7
10 2
16 3
8 1
16 12
16 7
5
2 3
3 4
3 1
1 5
3 5
```

输出样例
```
4
3
```

题解：由于本题数据量不大，所以可以暴力求解最近公共祖先 LCA。

1. 算法设计

（1）初始化父节点 fa[i]=i，访问标记 flag[i]=0。

（2）从 u 向上标记到树根。

（3）v 向上，第 1 个遇到的带有标记的节点即为 u、v 的最近公共祖先。

2. 算法实现

```
int LCA(int u,int v){//暴力求解最近公共祖先
    if(u==v)
        return u;
    flag[u]=1;
    while(fa[u]!=u){//u 向上走到根
        u=fa[u];
```

```
            flag[u]=1;
        }
        if(flag[v])
            return v;
        while(fa[v]!=v){//v 向上
            v=fa[v];
            if(flag[v])
                return v;
        }
        return 0;
}
```

训练 2　树上距离

题目描述（HDU2586）：有 n 栋房屋，由一些双向道路连接起来。每两栋房屋之间都有一条独特的简单道路（“简单”意味着不可以通过两条道路去一个地方）。人们每天总是喜欢这样问：“我从 A 房屋到 B 房屋需要走多远？”

输入：第 1 行是单个整数 T（$T \leqslant 10$），表示测试用例的数量。每个测试用例的第 1 行都包含 n（$2 \leqslant n \leqslant 40000$）和 m（$1 \leqslant m \leqslant 200$），表示房屋数量和查询数量。下面的 $n-1$ 行，每行都包含三个数字 i、j、k，表示有一条道路连接房屋 i 和房屋 j，长度为 k（$0<k \leqslant 40000$），房屋被标记为 $1 \sim n$。接下来的 m 行，每行都包含两个不同的整数 i 和 j，求房屋 i 和房屋 j 之间的距离。

输出：对每个测试用例，都输出 m 行查询答案，在每个测试用例后都输出一个空行。

输入样例	输出样例
2 3 2 1 2 10 3 1 15 1 2 2 3 2 2 1 2 100 1 2 2 1	10 25 100 100

题解：本题中任意两个房子之间的路径都是唯一的，是连通无环图，属于树形结构，所以求两个房子之间的距离相当于求树中两个节点之间的距离。可以采用最近公共祖先 LCA 的方法求解。求解 LCA 的方法有很多，在此使用树上倍增+ST 解决。

1. 算法设计

（1）根据输入数据采用链式前向星存储图。

（2）深度优先搜索，求深度、距离，初始化 F[*v*][0]。

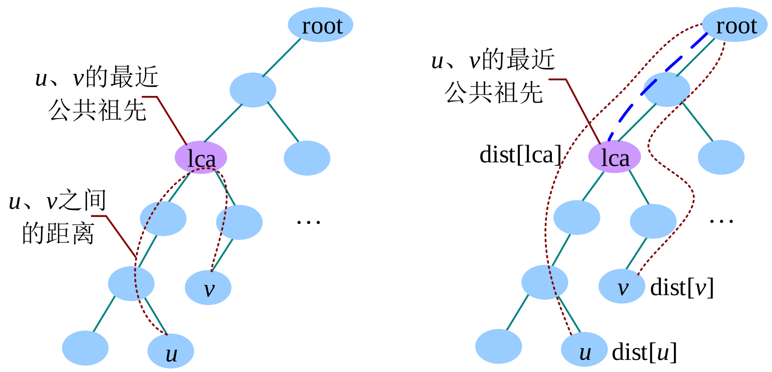
（3）创建 ST。

（4）查询 *x*、*y* 的最近公共祖先 lca。

（5）输出 *x*、*y* 的距离 dist[*x*]+dist[*y*]–2×dist[lca]。

2. 完美图解

求 *u* 和 *v* 之间的距离，若 *u* 和 *v* 的最近公共祖先为 lca，则 *u* 和 *v* 之间的距离为 *u* 到树根的距离加上 *v* 到树根的距离，再减去 2 倍的 lca 到树根的距离：dist[*u*]+dist[*v*]–2×dist[lca]。



3. 算法实现

```
void dfs(int u){//求深度、距离，初始化 F[v][0]
    for(int i=head[u];i;i=e[i].next){
        int v=e[i].to;
        if(v==F[u][0])
            continue;
        d[v]=d[u]+1;//深度
        dist[v]=dist[u]+e[i].c;//距离
        F[v][0]=u; //F[v][0]存放 v 的父节点
        dfs(v);
    }
}

void ST_create(){//构造 ST
    for(int j=1;j<=k;j++)
        for(int i=1;i<=n;i++)//i 先走 2^(j-1)步到达 F[i][j-1]，再走 2^(j-1)步
            F[i][j]=F[F[i][j-1]][j-1];
}
```

```
int LCA_st_query(int x,int y){//求 x、y 的最近公共祖先
    if(d[x]>d[y])//保证 x 的深度小于或等于 y
        swap(x,y);
    for(int i=k;i>=0;i--)//y 向上走到与 x 同一深度
        if(d[F[y][i]]>=d[x])
            y=F[y][i];
    if(x==y)
        return x;
    for(int i=k;i>=0;i--)//x、y 一起向上走
        if(F[x][i]!=F[y][i])
            x=F[x][i],y=F[y][i];
    return F[x][0];//返回 x 的父节点
}
```

训练 3　距离查询

题目描述（POJ1986）：约翰有 N 个农场，标记为 1～N。有 M 条垂直和水平的道路连接农场，每条道路的长度各不相同。每个农场都可以直接连接到北部（N）、南部（S）、东部（E）或西部（W）最多 4 个其他农场。农场位于道路的终点，正好一条道路连接一对农场，没有两条道路交叉。他希望知道两个农场之间的道路长度，农场的地图如下图所示。“1 6 13 E”表示从 F1 到 F6 有一条长度为 13 的道路，F6 在 F1 的东部。

```
F1 --- (13) ---- F6 --- (9) ----- F3
 |                                 |
(3)                                |
 |                                (7)
F4 --- (20) -------- F2            |
 |                                 |
(2)                               F5
 |
F7
```

输入：第 1 行包含两个整数 N（$2 \leqslant N \leqslant 40{,}000$）和 M（$1 \leqslant M < 40{,}000$）。第 2..M+1 行，每行都包含 4 个字符 a、b、l、d，表示两个农场 a 和 b 由一条路相连，长度为 l（$1 \leqslant l \leqslant 1000$），$d$ 是字符“N”“S”“E”或“W”，表示从 a 到 b 的道路方向。第 M+2 行包含单个整数 K（$1 \leqslant K \leqslant 10{,}000$），表示查询个数。接下来的 K 行，每行都包含距离查询的两个农场的编号。

输出：对每个查询，都单行输出两个农场的距离。

输入样例

```
7 6
```

输出样例

```
13
```

1 6 13 E 6 3 9 E 3 5 7 S 4 1 3 N 2 4 20 W 4 7 2 S 3 1 6 1 4 2 6	3 36

题解：本题实际上为树上距离查询问题，可以采用 Tarjan 算法离线处理所有查询。

1．算法设计

（1）根据输入数据采用链式前向星存储图。

（2）采用 Tarjan 算法离线处理所有查询。

2．算法实现

```
void LCA(int u){//最近公共祖先
   fa[u]=u;
   vis[u]=true;
   for(int i=head[u];i!=-1;i=E[i].next){
      int v=E[i].to;
      if(!vis[v]){
         dis[v]=dis[u]+E[i].lca;//E[i].lca 为 u、v 的边权
         LCA(v);
         fa[v]=u;
      }
   }
   for(int i=qhead[u];i!=-1;i=QE[i].next){
      int v=QE[i].to;
      if(vis[v]){
         QE[i].lca=dis[u]+dis[v]-2*dis[find(v)];//Find(v)为并查集找祖宗
         QE[i^1].lca=QE[i].lca;//i^1 为 i 的反向边，QE[i].lca 表示查询 u、v 的最短距离
      }
   }
}
```

训练 4　城市之间的联系

题目描述（HDU2874）：由于大部分道路在战争期间已被完全摧毁，所以两个城市之间可能没有路径，也没有环。已知道路状况，想知道任意两个城市之间是否存在路径。若答案是肯定的，则输出它们之间的最短距离。

输入：输入包含多个测试用例。每个用例的第 1 行都包含 3 个整数 n、m、c（$2 \leqslant n \leqslant 10000$，$0 \leqslant m < 10000$，$1 \leqslant c \leqslant 1000000$）。$n$ 表示城市数，编号为 $1 \sim n$。接下来的 m 行，每行都包含 3 个整数 i、j 和 k，表示城市 i 和城市 j 之间的道路，长度为 k。最后 c 行，每行都包含 i、j 两个整数，表示查询城市 i 和城市 j 之间的最短距离。

输出：对每个查询，若两个城市之间没有路径，则输出“Not connected”，否则输出它们之间的最短距离。

输入样例	输出样例
5 3 2 1 3 2 2 4 3 5 2 3 1 4 4 5	Not connected 6

题解：本题的两点之间无环，且有可能不连通，有可能不是一棵树，而是由多棵树组成的森林。因此需要判断是否在同一棵树中，若不在同一棵树中，则输出“Not connected”，否则可以使用求解最近公共祖先的 Tarjan 算法求解。

1．算法设计

（1）根据输入的数据，采用链式前向星存储图。

（2）采用 Tarjan 算法离线处理所有查询。因为本题的操作对象可能有多棵树，因此需要注意两个问题：①修改 Tarjan 算法，引入一个 root 参数，用来判断待查询的两个节点是否在同一棵树中；②对未访问过的节点再次执行 Tarjan 算法。

（3）将每个查询中两个节点之间的距离都存储在答案数组中。

2．算法实现

```
void LCA(int u,int deep,int root){//求解最近公共祖先
	fa[u]=u;
	dis[u]=deep;
	vis[u]=root;//标记 u 属于根为 root 的树
	for(int i=ehead[u];~i;i=e[i].next){
		int v=e[i].to;
		if(vis[v]==-1){
			LCA(v,deep+e[i].w,root);
			fa[v]=u;
		}
	}
	for(int i=qhead[u];~i;i=qe[i].next){
		int v=qe[i].to;
```

```
        if(vis[v]==root)//v 和 u 在同一棵树中
            ans[qe[i].id]=dis[v]+dis[u]-2*dis[Find(v)];//Find(v)为并查集找祖宗
    }
}

for(int i=1;i<=n;i++){
    if(vis[i]==-1)//若未访问，则从 i 开始执行
        LCA(i,0,i);
}
```