## 树上倍增的应用

在寒假第一阶段的时候，我们学习了使用树上倍增在 $O(logn)$ 的时间求出树上两点的最近公共祖先。

但是树上倍增不止于此，只要题目里 **没有修改** 操作，它还可以维护很多有关于 **树上路径** 的信息。

### 例题：

给定一棵 $n$ 个点的树，每条边上有边权。 $q$ 次询问，每次给出两个点 $u, v$ ，求 $u$ 到 $v$ 的树上简单路径所经过的所有边的最小值。

其中 $n, q \leq 2 \times 10^5$ 。

### 思路：

回忆一下求 $lca$ 的过程，记 $par[u][i]$ 为 $u$ 点向上跳 $2^i$ 步所到达的点的编号。

我们利用一个递推： $par[u][i] = par[par[u][i-1]][i-1]$ ， $O(nlogn)$ 的预处理出了每个点向上跳的情况。

那么类似的，我们是否能使用这种相同的形式来预处理呢?

我们记 $val[u][i]$ 为 $u$ 点向上跳 $2^i$ 步所到达的点的路径上经过的最小的边权值。

显然我们也有这样一个递推： $val[u][i] = min(val[u][i-1], val[par[u][i-1]][i-1])$ ，那么我们同样在 $O(nlogn)$ 的预处理出了每个点向上跳所经过的路径的权值情况了。

现在我们考虑回答每个询问，我们记 $f[i][j]$ 为从 $i$ 到 $j$ 的路径的最小边权值。

对于每次询问的 $u, v$ ，我们很容易发现 $f[u][v] = min(f[u][lca(u,v)], f[v][lca(u,v)])$ ，那么我们是否要单独先求出 $lca(u,v)$ 再做之后的计算呢？显然不需要这样。因为在求 $lca(u,v)$ 的过程中，还是一个倍增往上跳跃的过程，所以我们只要在倍增的过程中同时维护我们的答案即可，那么当求出 $lca(u,v)$ 的时候， $f[u][v]$ 的值我们自然也就可以知道了。

### 代码：

```cpp
void dfs(int u, int f) {
    dep[u] = dep[f] + 1;
    for (auto &[v, x] : g[u]) {
        if (v == f) continue;
        //处理往上一个点的情况
        par[v][0] = u;
        val[v][0] = x;
        dfs(v, u);
    }
}

// 在求解lca的过程中额外维护我们要的信息
int query(int u, int v) {
    int ans = 1 << 30;
    if (dep[u] > dep[v]) swap(u, v);
    int d = dep[v] - dep[u];
    for (int j = 30; j >= 0; j--) if (d & (1 << j)) {
        ans = min(ans, val[v][j]);
        v = par[v][j];
```

```
    }
    if (u == v) return ans;
    for (int j = 30; j >= 0; j--) if (par[u][j] != par[v][j]) {
        ans = min({ans, val[u][j], val[v][j]});
        u = par[u][j];
        v = par[v][j];
    }
    ans = min({ans, val[u][0], val[v][0]});
    return ans;
}

int main() {
    cin >> n >> q;
    for (int i = 1, u, v, w; i < n; i++) {
        // u 到 v 的权值为 w 的边
        cin >> u >> v >> w;
        g[u].push_back({v, w});
        g[v].push_back({u, w});
    }
    dfs(1, 0);
    for (int j = 1; j <= 30; j++) {
        for (int u = 1; u <= n; u++) {
            par[u][j] = par[par[u][j - 1]][j - 1];
            val[u][j] = min(val[u][j - 1], val[par[u][j - 1]][j - 1]);
        }
    }
    for (int i = 1; i <= q; i++) {
        int u, v;
        cin >> u >> v;
        cout << query(u, v) << endl;
    }
}
```

### 总结：

只要信息可以在倍增过程中 **快速合并**。并且问题中 **不带修改**，我们都可以使用倍增来快速询问一些关于 **树上路径** 的信息。

## 树上差分

### 点差分

我们来看这样一个例子：

给定一棵 $n$ 个点的树，每个点初始点权都是 $0$ 。你要执行 $q$ 次操作，每次操作形如：

- $u\ \ v\ \ x$ ：表示对 $u$ 到 $v$ 路径上的所有点点权都增加 $x$ 。

在 $q$ 次操作之后，你需要输出所有点的点权。

是否有一点点熟悉的味道？它是不是很像这样一个题目：

给定一个长度为 $n$ 的数组 $a$ 。你要执行 $q$ 次操作，每次操作形如：

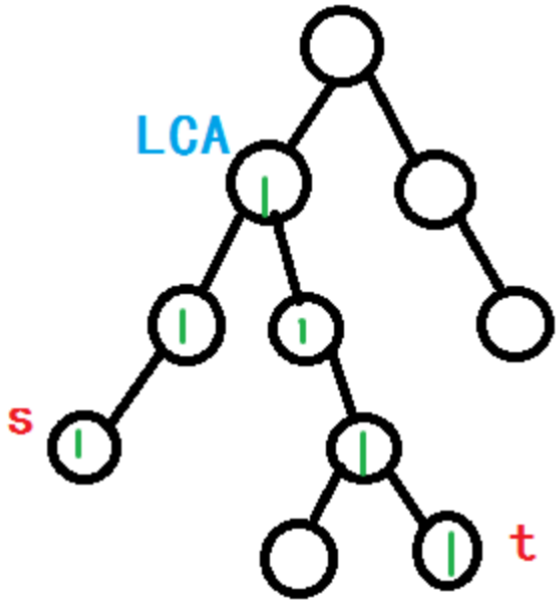
- $l\ \ r\ \ x$ ：表示对区间 $[l,r]$ 的每个数增加 $x$ 。

在 $q$ 次操作之后，你需要输出数组中每个数的值。

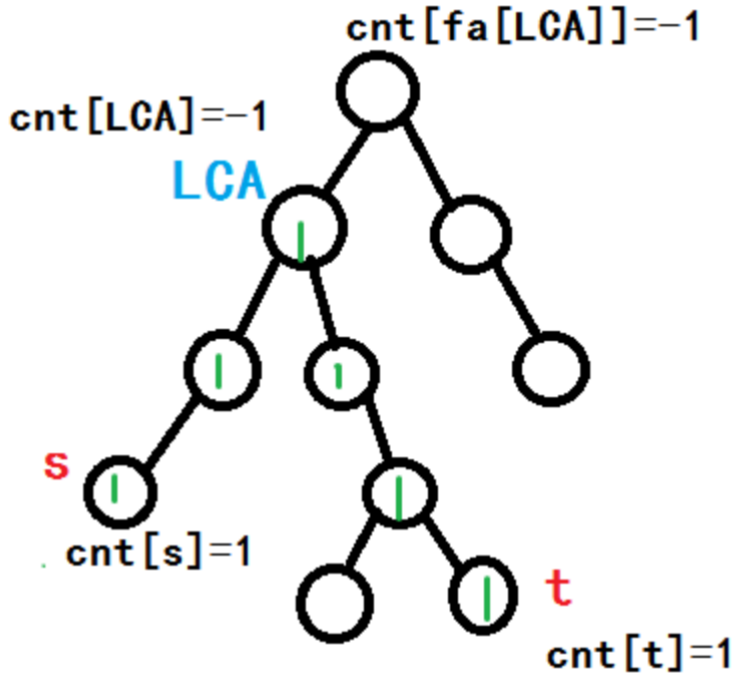
对于这个序列的版本，我们已经知道可以使用差分，把区间修改的操作转化到原数组的差分数组上，然后通过前缀和就可以复原数组，获得每一个数的值了，时间复杂度 $O(q+n)$ 。

那同样的，原题只不过是把这个背景从序列上搬到了树上。我们显然也可以使用类似的思想来解决这个问题。

假设我们对下图中的 $s$ 到 $t$ 结点路径上的点的点权都增加 $1$ 。




那么我们可以在差分数组 $cnt$ 上做如下操作：



其他点的 $cnt$ 都是 $0$ 。我们考虑如何复原，考虑在序列上的前缀和在树上的表现形式是什么，发现就是**子树和** 。所以每个点的点权只要求出它为根的子树的差分数组的点权和即可。这个过程可以在 $dfs$ 的时候回溯完成，类似一个极其简单的树形 $dp$ 。

这个差分的过程其实也是十分自然的，我们肯定要对 $u$ 和 $v$ 的差分数组上 $+1$ ，所以考虑转折点 $lca(u,v)$ ，它把两个点都包括在了自己的子树内，但是它只被经过了一次，所以要在 $lca(u,v)$ 的差分数组上 $-1$ ，但是这还没有结束，因为现在你发现 $lca(u,v)$ 的所有祖先因为都包含了 $lca(u,v)$ ，所以在计算时都是 $1$，但显然他们的值是 $0$ ，所以我们还要在 $lca(u,v)$ 的最近的祖先处的差分数组上 $-1$ 。显然这个点就是它的父亲。

这样，点差分就结束了。

## 边差分

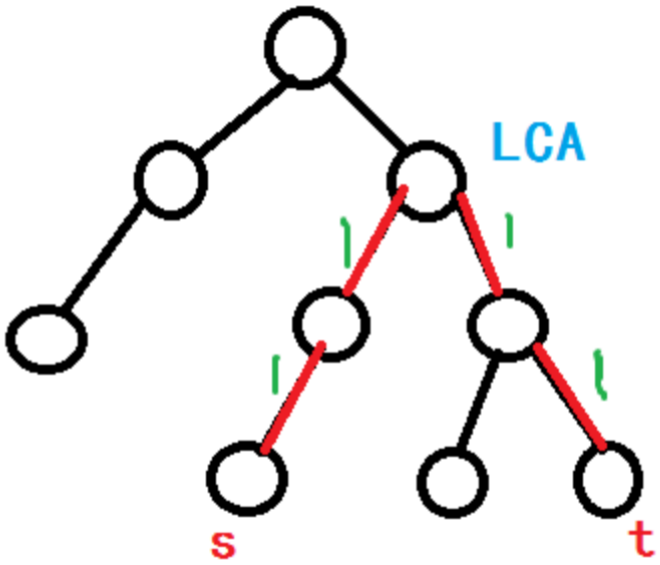
来看其他一个例子：

给定一棵 $n$ 个点的树，每条边的边权都是 $0$ 。你要执行 $q$ 次操作，每次操作形如：

- $u\ \ v\ \ x$ ：表示对 $u$ 到 $v$ 路径上的所有边边权都增加 $x$ 。

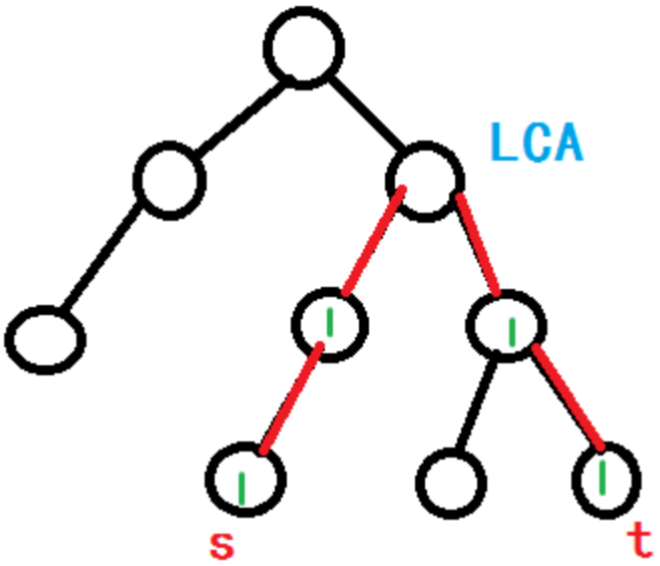
在 $q$ 次操作之后，你需要输出所有边的边权。

假设我们对下图中的 $s$ 到 $t$ 结点路径上的边的边权都增加 $1$ ：




这次的不同在于所有的操作都是对于边而言的。那么我们是否可以把边转化为点呢？

我们钦定这样一种对边标记的方法，设有一条边的两个点端点为 $u,v$ 。并且满足 $dep[u]>dep[v]$ ，$dep[i]$ 代表 $i$ 点的深度。我们就把这条边认为是 $u$ 的。这样一来由于树的每个节点有且仅有一个父亲，所以每个点只会被分到一条边。其中根节点没有分到任何一条边。这样一来， 上面的图就变成了这样：




接下来问题就回归了点差分。并且这个东西更简单了，因为它相当于对 $s$ 到 $t$ 的路径上除了 $lca(u,v)$ 的点点权加 $1$ 。

显然我们只需要对 $cnt[s]+1, cnt[t]+1, cnt[lca(s,t)]-2$ 即可。

这样，边差分也就结束了。

### 总结：

在序列上的区间修改，在树上的表现形式就是路径修改。

在序列上的前缀和，在树上的表现形式就是子树和。

对 $u$ 和 $v$ 路径上做 **点差分** 等价于
$cnt[u]+1, cnt[v]+1, cnt[lca(u,v)]-1, cnt[fa(lca(u,v))]-1$ 。

对 $u$ 和 $v$ 路径上做 **边差分** 等价于 $cnt[u]+1, cnt[v]-1, cnt[lca(u,v)]-2$ 。

## DFS序

我们接下来来强化一下上面的问题：

### 引入：

给定一棵 $n$ 个点的树，每个点初始点权都是 $0$ 。你要支持两种操作：

- $1\ \ u\ \ v\ \ x$ ：表示对 $u$ 到 $v$ 路径上的所有点点权都增加 $x$ 。
- $2\ \ u$ ：表示输出点 $u$ 的点权。

坏了，这下修改和查询会穿插进行了。我们还是先来考虑原来序列上的版本：

给定一个长度为 $n$ 的数组 $a$ 。你要支持两种操作：

- $1\ \ l\ \ r\ \ x$ ：表示对区间 $[l,r]$ 的每个数增加 $x$ 。
- $2\ \ d$ ：表示输出下标为 $d$ 的数。

显然，这个序列上的版本是普通的差分前缀和是无法完成的。我们需要使用 **树状数组或线段树** 维护差分或 **线段树打**$lazy$ 来完成。我们考虑前者，其实前者的本质还是差分和前缀和，差分还是可以直接做，只不过这个前缀和是动态的，所以我们无法直接获得，要使用 **树状数组** 维护。

考虑在树上的形式，显然问题是同样的。差分我们还是可以直接做，但是 **子树和** 这个东西我们只能通过 $dfs$ 完成一个简单树形 $dp$ 来获得。那么我们怎样才可以动态维护子树和呢？

## 更进一步：

我们得到了一个新的问题：

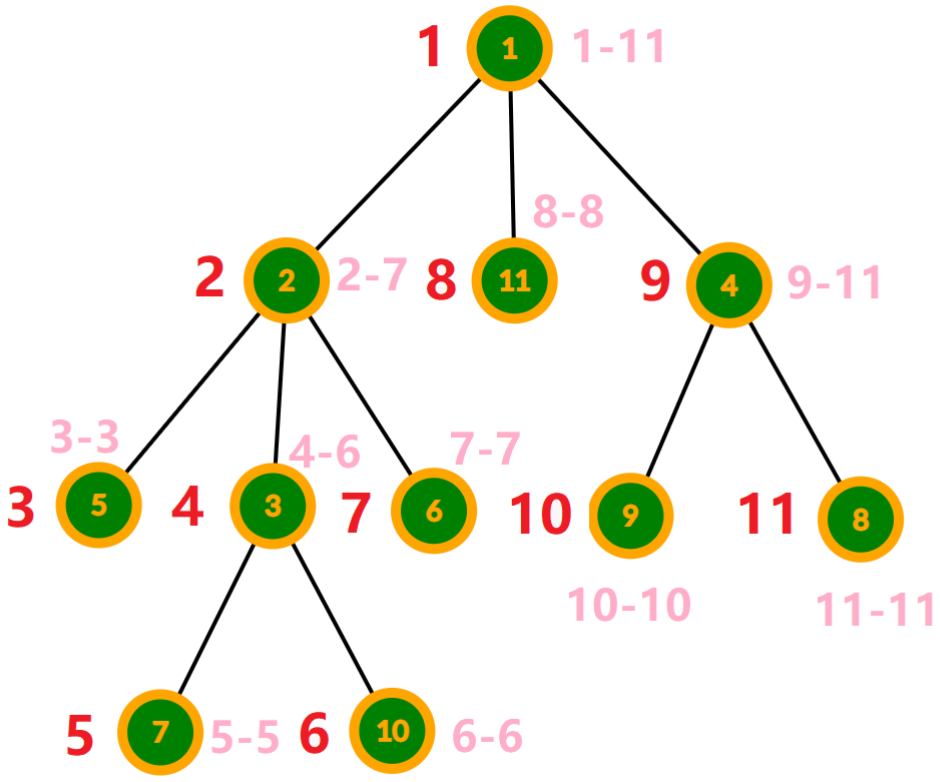
给定一棵 $n$ 个点的树，每个点初始点权都是 $0$ 。你要支持两种操作：

- $1\ \ u\ \ x$ ：表示对节点 $u$ 的点权增加 $x$ 。
- $2\ \ u$ ：表示输出点 $u$ 子树中所有点的点权和。

如何考虑？我们能否通过一种手段把这个树压扁，把每个点都拆下来然后摆成一条链，这样它不就等价于是一个数组了嘛？

但是不可以随便压，我们的操作 $2$ 是对于一棵子树的，这意味着我们在拆解完以后只要**每个点子树**中所有点都**在一起**，那么操作 $2$ 不就是一个简单的区间求和吗？而操作 $1$ 不就是单点修改吗？对于这个问题，我们可以使用树状数组完成。

我们引入一种叫 $dfs$ 序的东西。




故名思意，$dfs$ 序就是在对这棵树进行 $dfs$ 过程中每个结点按被遍历到的时间所标的序号。如上图，我们钦定对于多个儿子的结点，从左往右进行遍历。红色编号就是每个点的 $dfs$ 序，粉色范围就是每个点的子树中最大编号和最小编号。我们可以发现，每个点的子树的标号已经是一段连续的区间了。所以我们只要按照 $dfs$ 序展开，我们就可以把一棵树压成一个序列，并且使得每个点的子树中的点都是连续的。

```cpp
int L[N], R[N], tot;
// L[i] 即为 i 点为根的子树的最小点的编号
// R[i] 即为 i 点为根的子树的最大点的编号
void dfs(int x, int fa){
    L[x] = ++ tot;
    for(auto u : g[x]){
        if(u == fa) continue;
        dfs(u, x);
    }
    R[x] = tot;
}
```

那么上面的问题，我们也可以直接迎刃而解了。

在此我们只探讨狭义的 $dfs$ 序， 事实上，很多博客会把一种叫做欧拉序或者括号序的序列广义的称为 $dfs$ 序。

## 再举一个经典的例子：

给定一棵 $n$ 个点的树，每个点初始点权都是 $0$ 。你要支持两种操作：

- $1\ \ u\ \ x$ : 表示对点 $u$ 的点权增加 $x$ 。
- $2\ \ u\ \ v$ : 表示输出 $u$ 到 $v$ 路径上的点的点权和。

考虑询问，路径的和我们没法维护，就算使用 $dfs$ 序，也压扁压不出我们想要的样子。怎么可能任意两个点的树上路径所经过的点都可以是一个连续的区间。

路径最经典的转化就是到根。

设 $f_i$ 表示 $i$ 点到根的路径上所经过的点的点权和，显然对于一次询问 $u, v$ 的答案就等于 $f_u + f_v - 2 \times f_{lca(u,v)} + lca(u, v)$ 。

此时发现操作 $1$ 等价于对 $u$ 子树的每个点的 $f$ 值增加 $x$ 。

$dfs$ 序上树状数组维护差分即可。

## 总结：

$dfs$ 序可以把树拍扁，使得树上问题转化成序列上的问题，但是它仅仅只能保证某个点子树内的所有点连续。所以一般它只能将 **子树** 问题转化到序列上。对于 **路径** 问题 $dfs$ 序无能为力。