最近公共祖先

定义

最近公共祖先简称 LCA（Lowest Common Ancestor）。两个节点的最近公共祖先，就是这两个点的公共祖先里面，离根最远的那个。 为了方便，我们记某点集 $S = \{v_1, v_2, \ldots, v_n\}$ 的最近公共祖先为 $\operatorname{LCA}(v_1, v_2, \ldots, v_n)$ 或 $\operatorname{LCA}(S)$。

性质

> 本节 **性质** 部分内容翻译自 wcipeg，并做过修改。

1. $\operatorname{LCA}(\{u\}) = u$；
2. u 是 v 的祖先，当且仅当 $\operatorname{LCA}(u, v) = u$；
3. 如果 u 不为 v 的祖先并且 v 不为 u 的祖先，那么 u, v 分别处于 $\operatorname{LCA}(u, v)$ 的两棵不同子树中；
4. 前序遍历中，$\operatorname{LCA}(S)$ 出现在所有 S 中元素之前，后序遍历中 $\operatorname{LCA}(S)$ 则出现在所有 S 中元素之后；
5. 两点集并的最近公共祖先为两点集分别的最近公共祖先的最近公共祖先，即 $\operatorname{LCA}(A \cup B) = \operatorname{LCA}(\operatorname{LCA}(A), \operatorname{LCA}(B))$；
6. 两点的最近公共祖先必定处在树上两点间的最短路上；
7. $d(u, v) = h(u) + h(v) - 2h(\operatorname{LCA}(u, v))$，其中 d 是树上两点间的距离，h 代表某点到树根的距离。

求法

朴素算法

过程

可以每次找深度比较大的那个点，让它向上跳。显然在树上，这两个点最后一定会相遇，相遇的位置就是想要求的 LCA。 或者先向上调整深度较大的点，令他们深度相同，然后再共同向上跳转，最后也一定会相遇。

性质

朴素算法预处理时需要 dfs 整棵树，时间复杂度为 $O(n)$，单次查询时间复杂度为 $\Theta(n)$。如果树满足随机性质，则时间复杂度与这种随机树的期望高度有关。

倍增算法

过程

倍增算法是最经典的 LCA 求法，他是朴素算法的改进算法。通过预处理 $\mathrm{fa}_{x,i}$ 数组，游标可以快速移动，大幅减少了游标跳转次数。$\mathrm{fa}_{x,i}$ 表示点 x 的第 2^i 个祖先。$\mathrm{fa}_{x,i}$ 数组可以通过 dfs 预处理出来。

现在我们看看如何优化这些跳转： 在调整游标的第一阶段中，我们要将 u, v 两点跳转到同一深度。我们可以计算出 u, v 两点的深度之差，设其为 y。通过将 y 进行二进制拆分，我们将 y 次游标跳转优化为「y 的二进制表示所含 `1` 的个数」次游标跳转。 在第二阶段中，我们从最大的 i 开始循环尝试，一直尝试到 0（包括 0），如果 $\mathrm{fa}_{u,i} \neq \mathrm{fa}_{v,i}$，则 $u \leftarrow \mathrm{fa}_{u,i}, v \leftarrow \mathrm{fa}_{v,i}$，那么最后的 LCA 为 $\mathrm{fa}_{u,0}$。

性质

倍增算法的预处理时间复杂度为 $O(n \log n)$，单次查询时间复杂度为 $O(\log n)$。 另外倍增算法可以通过交换 `fa` 数组的两维使较小维放在前面。这样可以减少 cache miss 次数，提高程序效率。

例题

HDU 2586 How far away? 树上最短路查询。

可先求出 LCA，再结合性质 7 进行解答。也可以直接在求 LCA 时求出结果。

参考代码

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

constexpr int MXN = 40005;
using namespace std;
vector<int> v[MXN];
vector<int> w[MXN];

int fa[MXN][31], cost[MXN][31], dep[MXN];
int n, m;
int a, b, c;

// dfs，用来为 lca 算法做准备。接受两个参数：dfs 起始节点和它的父亲节点。
void dfs(int root, int fno) {
  // 初始化：第 2^0 = 1 个祖先就是它的父亲节点，dep 也比父亲节点多 1。
  fa[root][0] = fno;
  dep[root] = dep[fa[root][0]] + 1;
  // 初始化：其他的祖先节点：第 2^i 的祖先节点是第 2^(i-1) 的祖先节点的第
  // 2^(i-1) 的祖先节点。
  for (int i = 1; i < 31; ++i) {
    fa[root][i] = fa[fa[root][i - 1]][i - 1];
    cost[root][i] = cost[fa[root][i - 1]][i - 1] + cost[root][i - 1];
  }
  // 遍历子节点来进行 dfs。
  int sz = v[root].size();
  for (int i = 0; i < sz; ++i) {
    if (v[root][i] == fno) continue;
    cost[v[root][i]][0] = w[root][i];
    dfs(v[root][i], root);
  }
}

// lca。用倍增算法算取 x 和 y 的 lca 节点。
int lca(int x, int y) {
  // 令 y 比 x 深。
  if (dep[x] > dep[y]) swap(x, y);
  // 令 y 和 x 在一个深度。
  int tmp = dep[y] - dep[x], ans = 0;
  for (int j = 0; tmp; ++j, tmp >>= 1)
    if (tmp & 1) ans += cost[y][j], y = fa[y][j];
  // 如果这个时候 y = x，那么 x，y 就都是它们自己的祖先。
  if (y == x) return ans;
  // 不然的话，找到第一个不是它们祖先的两个点。
  for (int j = 30; j >= 0 && y != x; --j) {
    if (fa[x][j] != fa[y][j]) {
```

```
      ans += cost[x][j] + cost[y][j];
      x = fa[x][j];
      y = fa[y][j];
    }
  }
  // 返回结果。
  ans += cost[x][0] + cost[y][0];
  return ans;
}

void Solve() {
  cin.tie(nullptr)->sync_with_stdio(false);
  // 初始化表示祖先的数组 fa，代价 cost 和深度 dep。
  memset(fa, 0, sizeof(fa));
  memset(cost, 0, sizeof(cost));
  memset(dep, 0, sizeof(dep));
  // 读入树：节点数一共有 n 个，查询 m 次，每一次查找两个节点的 lca
点。
  cin >> n >> m;
  // 初始化树边和边权
  for (int i = 1; i <= n; ++i) {
    v[i].clear();
    w[i].clear();
  }
  for (int i = 1; i < n; ++i) {
    cin >> a >> b >> c;
    v[a].push_back(b);
    v[b].push_back(a);
    w[a].push_back(c);
    w[b].push_back(c);
  }
  // 为了计算 lca 而使用 dfs。
  dfs(1, 0);
  for (int i = 0; i < m; ++i) {
    cin >> a >> b;
    cout << lca(a, b) << '\n';
  }
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int T;
  cin >> T;
  while (T--) Solve();
  return 0;
}
```

Tarjan 算法

过程

Tarjan 算法是一种 **离线算法**，需要使用 并查集 记录某个结点的祖先结点。做法如下：

1. 首先接受输入边（邻接链表）、查询边（存储在另一个邻接链表内）。查询边其实是虚拟加上去的边，为了方便，每次输入查询边的时候，将这个边及其反向边都加入到 `queryEdge` 数组里。
2. 然后对其进行一次 DFS 遍历，同时使用 `visited` 数组进行记录某个结点是否被访问过、`parent` 记录当前结点的父亲结点。
3. 其中涉及到了 **回溯思想**，我们每次遍历到某个结点的时候，认为这个结点的根结点就是它本身。让以这个结点为根节点的 DFS 全部遍历完毕了以后，再将这个结点的根节点设置为这个结点的父一级结点。
4. 回溯的时候，如果以该节点为起点，`queryEdge` 查询边的另一个结点也恰好访问过了，则直接更新查询边的 LCA 结果。
5. 最后输出结果。

性质

Tarjan 算法需要初始化并查集，所以预处理的时间复杂度为 $O(n)$。

朴素的 Tarjan 算法处理所有 m 次询问的时间复杂度为 $O(m\alpha(m+n,n)+n)$，但是 Tarjan 算法的常数比倍增算法大。存在 $O(m+n)$ 的实现。

> **注意**
>
> 并不存在「朴素 Tarjan LCA 算法中使用的并查集性质比较特殊，单次调用 `find()` 函数的时间复杂度为均摊 $O(1)$」这种说法。
>
> 以下的朴素 Tarjan 实现复杂度为 $O(m\alpha(m+n,n)+n)$。如果需要追求严格线性，可以参考 Gabow 和 Tarjan 于 1983 年的论文。其中给出了一种复杂度为 $O(m+n)$ 的做法。

实现

参考代码

```cpp
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;

class Edge {
 public:
  int toVertex, fromVertex;
  int next;
  int LCA;
  Edge() : toVertex(-1), fromVertex(-1), next(-1), LCA(-1) {};
  Edge(int u, int v, int n) : fromVertex(u), toVertex(v),
next(n), LCA(-1) {};
};

constexpr int MAX = 100;
int head[MAX], queryHead[MAX];
Edge edge[MAX], queryEdge[MAX];
int parent[MAX], visited[MAX];
int vertexCount, queryCount;

int find(int x) {
  if (parent[x] == x) {
    return x;
  } else {
    return parent[x] = find(parent[x]);
  }
}

void tarjan(int u) {
  parent[u] = u;
  visited[u] = 1;

  for (int i = head[u]; i != -1; i = edge[i].next) {
    Edge& e = edge[i];
    if (!visited[e.toVertex]) {
      tarjan(e.toVertex);
      parent[e.toVertex] = u;
    }
  }

  for (int i = queryHead[u]; i != -1; i = queryEdge[i].next) {
    Edge& e = queryEdge[i];
    if (visited[e.toVertex]) {
      queryEdge[i ^ 1].LCA = e.LCA = find(e.toVertex);
    }
  }
}

```

```cpp
int main() {
  memset(head, 0xff, sizeof(head));
  memset(queryHead, 0xff, sizeof(queryHead));

  cin >> vertexCount >> queryCount;
  int count = 0;
  for (int i = 0; i < vertexCount - 1; i++) {
    int start = 0, end = 0;
    cin >> start >> end;

    edge[count] = Edge(start, end, head[start]);
    head[start] = count;
    count++;

    edge[count] = Edge(end, start, head[end]);
    head[end] = count;
    count++;
  }

  count = 0;
  for (int i = 0; i < queryCount; i++) {
    int start = 0, end = 0;
    cin >> start >> end;

    queryEdge[count] = Edge(start, end, queryHead[start]);
    queryHead[start] = count;
    count++;

    queryEdge[count] = Edge(end, start, queryHead[end]);
    queryHead[end] = count;
    count++;
  }

  tarjan(1);

  for (int i = 0; i < queryCount; i++) {
    Edge& e = queryEdge[i * 2];
    cout << "(" << e.fromVertex << "," << e.toVertex << ") " <<
e.LCA << endl;
  }

  return 0;
}
```

用欧拉序列转化为 RMQ 问题

定义

对一棵树进行 DFS，无论是第一次访问还是回溯，每次到达一个结点时都将编号记录下来，可以得到一个长度为 $2n-1$ 的序列，这个序列被称作这棵树的欧拉序列。

在下文中，把结点 u 在欧拉序列中第一次出现的位置编号记为 $pos(u)$（也称作节点 u 的欧拉序），把欧拉序列本身记作 $E[1..2n-1]$。

过程

有了欧拉序列，LCA 问题可以在线性时间内转化为 RMQ 问题，即 $pos(LCA(u,v)) = \min\{pos(k)|k \in E[pos(u)..pos(v)]\}$。

这个等式不难理解：从 u 走到 v 的过程中一定会经过 $LCA(u,v)$，但不会经过 $LCA(u,v)$ 的祖先。因此，从 u 走到 v 的过程中经过的欧拉序最小的结点就是 $LCA(u,v)$。

用 DFS 计算欧拉序列的时间复杂度是 $O(n)$，且欧拉序列的长度也是 $O(n)$，所以 LCA 问题可以在 $O(n)$ 的时间内转化成等规模的 RMQ 问题。

实现

参考代码

```
int dfn[N << 1], pos[N], tot, st[30][(N << 1) + 2],
    rev[30][(N << 1) + 2];  // rev表示最小深度对应的节点编号

void dfs(int cur, int dep) {
  dfn[++tot] = cur;
  depth[tot] = dep;
  pos[cur] = tot;
  for (int i = head[t]; i; i = side[i].next) {
    int v = side[i].to;
    if (!pos[v]) {
      dfs(v, dep + 1);
      dfn[++tot] = cur, depth[tot] = dep;
    }
  }
}

void init() {
  for (int i = 2; i <= tot + 1; ++i)
    lg[i] = lg[i >> 1] + 1;  // 预处理 lg 代替库函数 log2 来优化常
数
  for (int i = 1; i <= tot; i++) st[0][i] = depth[i], rev[0][i] =
dfn[i];
  for (int i = 1; i <= lg[tot]; i++)
    for (int j = 1; j + (1 << i) - 1 <= tot; j++)
      if (st[i - 1][j] < st[i - 1][j + (1 << i - 1)])
        st[i][j] = st[i - 1][j], rev[i][j] = rev[i - 1][j];
      else
        st[i][j] = st[i - 1][j + (1 << i - 1)],
        rev[i][j] = rev[i - 1][j + (1 << i - 1)];
}

int query(int l, int r) {
  int k = lg[r - l + 1];
  return st[k][l] < st[k][r + 1 - (1 << k)] ? rev[k][l]
                                            : rev[k][r + 1 - (1
<< k)];
}
```

当我们需要查询某点对 (u, v) 的 LCA 时，查询区间 $[\min\{pos[u], pos[v]\}, \max\{pos[u], pos[v]\}]$ 上最小值的所代表的节点即可。

若使用 ST 表来解决 RMQ 问题，那么该算法不支持在线修改，预处理的时间复杂度为 $O(n \log n)$，每次查询 LCA 的时间复杂度为 $O(1)$。

树链剖分

LCA 为两个游标跳转到同一条重链上时深度较小的那个游标所指向的点。

树链剖分的预处理时间复杂度为 $O(n)$，单次查询的时间复杂度为 $O(\log n)$，并且常数较小。

Link Cut Tree

在 Link Cut Tree 中，设连续两次 access 操作的点分别为 `u` 和 `v`，则第二次 access 操作返回的点即为 `u` 和 `v` 的 LCA.

在无 link 和 cut 等操作的情况下，使用 Link Cut Tree 单次查询的时间复杂度为 $O(\log n)$。

标准 RMQ

前面讲到了借助欧拉序将 LCA 问题转化为 RMQ 问题，其瓶颈在于 RMQ。如果能做到 $O(n) \sim O(1)$ 求解 RMQ，那么也就能做到 $O(n) \sim O(1)$ 求解 LCA。

注意到欧拉序满足相邻两数之差为 1 或者 -1，所以可以使用 $O(n) \sim O(1)$ 的 加减 1RMQ 来做。

时间复杂度 $O(n) \sim O(1)$，空间复杂度 $O(n)$，支持在线查询，常数较大。

例题 Luogu P3379【模板】最近公共祖先（LCA）

参考代码

```cpp
#include <algorithm>
#include <cmath>
#include <iostream>
using namespace std;

constexpr int N = 5e5 + 5;

struct PlusMinusOneRMQ {  // RMQ
  // Copyright (C) 2018 Skqliao. All rights served.
  constexpr static int M = 9;

  int blocklen, block, Minv[N], F[N / M * 2 + 5][M << 1], T[N],
f[1 << M][M][M],
      S[N];

  void init(int n) {  // 初始化
    blocklen = std::max(1, (int)(log(n * 1.0) / log(2.0)) / 2);
    block = n / blocklen + (n % blocklen > 0);
    int total = 1 << (blocklen - 1);
    for (int i = 0; i < total; i++) {
      for (int l = 0; l < blocklen; l++) {
        f[i][l][l] = l;
        int now = 0, minv = 0;
        for (int r = l + 1; r < blocklen; r++) {
          f[i][l][r] = f[i][l][r - 1];
          if ((1 << (r - 1)) & i) {
            now++;
          } else {
            now--;
            if (now < minv) {
              minv = now;
              f[i][l][r] = r;
            }
          }
        }
      }
    }
    T[1] = 0;
    for (int i = 2; i < N; i++) {
      T[i] = T[i - 1];
      if (!(i & (i - 1))) {
        T[i]++;
      }
    }
  }

  void initmin(int a[], int n) {
    for (int i = 0; i < n; i++) {
      if (i % blocklen == 0) {
```

```
        Minv[i / blocklen] = i;
        S[i / blocklen] = 0;
      } else {
        if (a[i] < a[Minv[i / blocklen]]) {
          Minv[i / blocklen] = i;
        }
        if (a[i] > a[i - 1]) {
          S[i / blocklen] |= 1 << (i % blocklen - 1);
        }
      }
    }
    for (int i = 0; i < block; i++) {
      F[i][0] = Minv[i];
    }
    for (int j = 1; (1 << j) <= block; j++) {
      for (int i = 0; i + (1 << j) - 1 < block; i++) {
        int b1 = F[i][j - 1], b2 = F[i + (1 << (j - 1))][j - 1];
        F[i][j] = a[b1] < a[b2] ? b1 : b2;
      }
    }
  }

  int querymin(int a[], int L, int R) {
    int idl = L / blocklen, idr = R / blocklen;
    if (idl == idr)
      return idl * blocklen + f[S[idl]][L % blocklen][R %
blocklen];
    else {
      int b1 = idl * blocklen + f[S[idl]][L % blocklen][blocklen
- 1];
      int b2 = idr * blocklen + f[S[idr]][0][R % blocklen];
      int buf = a[b1] < a[b2] ? b1 : b2;
      int c = T[idr - idl - 1];
      if (idr - idl - 1) {
        int b1 = F[idl + 1][c];
        int b2 = F[idr - 1 - (1 << c) + 1][c];
        int b = a[b1] < a[b2] ? b1 : b2;
        return a[buf] < a[b] ? buf : b;
      }
      return buf;
    }
  }
} rmq;

int n, m, s;

struct Edge {
  int v, nxt;
} e[N * 2];

int tot, head[N];

```

```cpp
void init(int n) {
  tot = 0;
  fill(head, head + n + 1, 0);
}

void addedge(int u, int v) {  // 加边
  ++tot;
  e[tot] = Edge{v, head[u]};
  head[u] = tot;

  ++tot;
  e[tot] = Edge{u, head[v]};
  head[v] = tot;
}

int dfs_clock, dfn[N * 2], dep[N * 2], st[N];

void dfs(int u, int fa, int d) {
  st[u] = dfs_clock;

  dfn[dfs_clock] = u;
  dep[dfs_clock] = d;
  ++dfs_clock;

  int v;
  for (int i = head[u]; i; i = e[i].nxt) {
    v = e[i].v;
    if (v == fa) continue;
    dfs(v, u, d + 1);
    dfn[dfs_clock] = u;
    dep[dfs_clock] = d;
    ++dfs_clock;
  }
}

void build_lca() {  // like init
  rmq.init(dfs_clock);
  rmq.initmin(dep, dfs_clock);
}

int LCA(int u, int v) {  // 求解LCA，看题解用RMQ的方法
  int l = st[u], r = st[v];
  if (l > r) swap(l, r);
  return dfn[rmq.querymin(dep, l, r)];
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m >> s;

  init(n);
  int u, v;
```

```
154     for (int i = 1; i <= n - 1; ++i) {
155       cin >> u >> v;
156       addedge(u, v);
157     }
158
159     dfs_clock = 0;
160     dfs(s, s, 0);
161
162     build_lca();
163
164     for (int i = 1; i <= m; ++i) {
165       cin >> u >> v;
166       cout << LCA(u, v) << '\n';
167     }

       return 0;
     }
```

习题

- 祖孙询问
- 货车运输
- 点的距离

本页面最近更新：2025/6/19 04:38:13，更新历史

发现错误？想一起完善？ 在 GitHub 上编辑此页！

本页面贡献者：H-J-Granger, Ir1d, countercurrent-time, Early0v0, Enter-tainer, NachtgeistW, StudyingFather, therehello, CCXXXI, cjsoft, diauweb, HeRaNO, Konano, ouuan, sshwy, Henry-ZHR, hsfzLZH1, Tiphereth-A, AngelKitty, Backl1ght, billchenchina, ChickenHu, ChungZH, EtaoinWu, ezoixx130, GekkaSaori, huaruoji, Hunter19019, iamtwz, imp2002, kenlig, LovelyBuggies, Makkiy, Marcythm, Menci, mgt, minghu6, P-Y-Y, PeterlitsZo, PotassiumWings, psz2007, SamZhangQingChuan, shuzhouliu, SkqLiao, SukkaW, Suyun514, ttzztztz, vincent-163, WAAutoMaton, weiyong1024, c-forrest, Chlero, GavinZhengOI, Gesrua, H-Shen, Haohu Shen, kxccc, Lyccrius, lyccrius, lychees, Peanut-Tang, TrisolarisHD, TrisolarisHD

© 本页面的全部内容在 **CC BY-SA 4.0 和 SATA** 协议之条款下提供，附加条款亦可能应用