次短路径(pal,1s,256MB)

【算法分析】

由题意可知：1 号点到每个点的最短路径是唯一确定的。

用堆优化 Dijkstra 做 1 号点的单源最短路，求出到 i 号点的最短路径长度 dis[i]，对于无向图中的每一条边(a,b,c)，若 $dis[a]+c=dis[b]$ 或 $dis[b]+c=dis[a]$，就在新图中连边，得到最短路图。由于最短路径都是唯一的，1 号点所在的连通块构成一棵树，我们只需要考虑这棵树上的点，其余与 1 号点不连通的点输出-1。

以 1 号点作为该树的根，对于树上任意的另外一个点 i，由于它到父结点的边不能走，一条从点 i 出发的最短合法路径一定是以下的形式：从点 i 沿树边走到以 i 为根的子树内一点 j（j 可以与 i 相同），然后从点 j 出发沿一条（若走了多条，一定不是最短的路径）不在最短路图中的边（以下简称为非树边）走到以点 i 为根的子树外一点 k，最后再从点 k 沿树边走到 1 号点。

若我们确定了所走的非树边和起点 i，就能确定路径长度。考虑对于每个点 i，维护边的两端点恰好分别在以 i 为根的子树内和子树外的所有非树边。那么每条非树边需要在遍历到边的任意一个端点时被加入，在遍历到两个端点的 LCA 时被删除，且我们每次需要把点 i 子节点维护的非树边合并，每次询问对于点 i 所有非树边的最小路径长度。

容易想到用左偏树来维护，考虑怎样维护每条非树边的路径长度。若我们直接维护答案，每次从子节点合并上来时，需要让子节点的左偏树中所有非树边的路径长度加上子节点到点 i 的边权，这可以在左偏树上打标记来实现，但实际上有更简单的方法。对每条非树边(a,b,c)，我们在左偏树中以 $dis[a]+dis[b]+c$ 作为键值 key，则对于点 i 所求的答案就是 key-dis[i]，并且这样不会影响我们所取答案的最优性。

时间复杂度 $O(n\log n)$。

【参考程序】

```
//陈贤
#include <bits/stdc++.h>

template <class T>
inline void read(T &res)
{
    char ch;
    while (ch = getchar(), !isdigit(ch));
    res = ch ^ 48;
    while (ch = getchar(), isdigit(ch))
        res = res * 10 + ch - 48;
}

template <class T>
inline void put(T x)
{
    if (x > 9) put(x / 10);
    putchar(x % 10 + 48);
}
```

```
using std::vector;
using std::priority_queue;
const int N = 1e5 + 5;
const int M = 4e5 + 5;
const int Maxn = 0x3f3f3f3f;
int n, m, E;
int dep[N], anc[N][20], rt[N], ans[N];
int key[M], dist[M], dis[N];
int lc[M], rc[M], pa[M], pb[M], pc[M];
bool vis[M];
vector<int> v[N], add[N], del[N];

struct point
{
    int s, t;

    point() {}
    point(int S, int T):
        s(S), t(T) {}

    inline bool operator < (const point &a) const
    {
        return s > a.s;
    }
};
priority_queue<point> que;

struct Edge
{
    int to, cst; Edge *nxt;
}p[M], *lst[N], *P = p;

inline void Link(int x, int y, int z)
{
    (++P)->nxt = lst[x]; lst[x] = P; P->to = y; P->cst = z;
    (++P)->nxt = lst[y]; lst[y] = P; P->to = x; P->cst = z;
}

inline int Merge(int x, int y)
{
    if (!x || !y)
        return x + y;
    if (key[x] > key[y])
```

```
        std::swap(x, y);
    rc[x] = Merge(rc[x], y);
    if (dist[lc[x]] < dist[rc[x]])
        std::swap(lc[x], rc[x]);
    dist[x] = dist[rc[x]] + 1;
    return x;
}

inline void Dijkstra() //堆优化 Dijkstra 求最短路
{
    for (int i = 1; i <= n; ++i)
        dis[i] = Maxn;
    que.push(point(dis[1] = 0, 1));
    for (int i = 1, x, y; i <= n; ++i)
    {
        while (!que.empty() && vis[que.top().t])
            que.pop();
        if (que.empty())
            break;
        else
        {
            vis[x = que.top().t] = true;
            que.pop();
        }
        for (Edge *e = lst[x]; e; e = e->nxt)
            if (y = e->to, dis[y] > dis[x] + e->cst)
                que.push(point(dis[y] = dis[x] + e->cst, y));
    }
}

inline int query_LCA(int x, int y) //倍增求 LCA
{
    if (x == y)
        return x;
    if (dep[x] < dep[y])
        std::swap(x, y);
    for (int i = 17; i >= 0; --i)
    {
        if (dep[anc[x][i]] >= dep[y])
            x = anc[x][i];
        if (x == y)
            return x;
    }
    for (int i = 17; i >= 0; --i)
```

```
        if (anc[x][i] != anc[y][i])
            x = anc[x][i], y = anc[y][i];
    return anc[x][0];
}

inline void init_LCA(int x)
{
    dep[x] = dep[anc[x][0]] + 1;
    for (int i = 0; anc[x][i]; ++i)
        anc[x][i + 1] = anc[anc[x][i]][i];
    for (int i = 0, im = v[x].size(); i < im; ++i)
    {
        int y = v[x][i];
        if (y == anc[x][0])
            continue;
        anc[y][0] = x;
        init_LCA(y);
    }
}

inline void Dfs(int x)
{
    int u = 0;
    for (int i = 0, im = v[x].size(); i < im; ++i)
    {
        int y = v[x][i];
        if (y == anc[x][0])
            continue;
        Dfs(y);
        u = Merge(rt[y], u);
    }
    for (int i = 0, im = add[x].size(); i < im; ++i)
        u = Merge(u, add[x][i]);
    for (int i = 0, im = del[x].size(); i < im; ++i)
        vis[del[x][i]] = true;
    while (u && vis[u])
        u = Merge(lc[u], rc[u]);
    rt[x] = u;
    ans[x] = !u ? -1 : key[u] - dis[x];
}

int main()
{
    freopen("pal.in", "r", stdin);
```

```
freopen("pal.out", "w", stdout);

read(n); read(m);
dist[0] = -1;
for (int i = 1; i <= m; ++i)
{
    read(pa[i]);
    read(pb[i]);
    read(pc[i]);
    Link(pa[i], pb[i], pc[i]);
}

Dijkstra();
for (int i = 1; i <= m; ++i) //建最短路径树
{
    if (dis[pa[i]] == Maxn && dis[pb[i]] == Maxn)
        continue;
    if (dis[pa[i]] + pc[i] == dis[pb[i]])
        v[pa[i]].push_back(pb[i]);
    else if (dis[pb[i]] + pc[i] == dis[pa[i]])
        v[pb[i]].push_back(pa[i]);
}

memset(vis, false, sizeof(vis));
memset(ans, 255, sizeof(ans));
init_LCA(1);

for (int i = 1; i <= m; ++i)
{
    if (dis[pa[i]] == Maxn && dis[pb[i]] == Maxn)
        continue;
    if (dis[pa[i]] + pc[i] != dis[pb[i]]
    && dis[pb[i]] + pc[i] != dis[pa[i]])
    {
        int x = pa[i],
            y = pb[i],
            z = query_LCA(x, y);

        add[x].push_back(++E);
        key[E] = dis[x] + dis[y] + pc[i];
        del[z].push_back(E);

        add[y].push_back(++E);
        key[E] = dis[x] + dis[y] + pc[i];
```

```
            del[z].push_back(E);
        }
    }

    Dfs(1);
    for (int i = 2; i <= n; ++i)
        if (ans[i] == -1)
            puts("-1");
        else
            put(ans[i]), putchar('\n');

    fclose(stdin); fclose(stdout);
    return 0;
}
```

排列游戏(fiend,3s,233MB)

【算法分析】

根据方阵行列式的定义：

$$\det A = \sum_{p是1到n的排列} (-1)^{p的逆序对数} \prod_{i=1}^{n} a_{i,p_i}$$

回到题目，我们很容易发现令矩阵A：

$$\mathrm{A_{i,j}} = [L_i \le j \le R_i]$$

那么我们要求的就是矩阵A的行列式的正负：如果行列式为正则输出 Y，为负则输出 F，为0则输出 D。

矩阵的行列式具有以下性质：

（1） 交换两行后，行列式的值变成原来的相反数。

（2） 将一行加上另一行的k倍（k为任意实数），行列式的值不变。

这些性质让我们可以通过高斯消元求解行列式，单组数据复杂度$\mathrm{O(n^3)}$。

注意这个矩阵的特殊性质：仅由0和1组成，每行的1是连续的一段。

按照高斯消元的策略，我们的做法是：枚举i从1到n，选一个$\mathrm{j \in [i, n]}$满足第j行的1出现的区间左端点为i（如果找不到这样的j则行列式为0），如果$\mathrm{j \ne i}$则令$\mathrm{cnt++}$并让第i行和第j行互换，然后用第i行去消第i行之后所有1出现的左端点为i的行即可。

如果能够完成消元，则行列式的值为$(-1)^{cnt}$。

当然，我们需要保证每次消元之后，这个矩阵一直保持它的特殊性质。故我们每次选择的j，需要满足第j行的1出现的区间左端点为i且右端点最小。易得这样可以使得矩阵一直保持特殊性质：仅由0和1组成，每行的1是连续的一段。

如何实现消元？按照上文的规则，假设我们当前选出的行连续的1所占的区间为$\mathrm{[L, R]}$，那么我们需要进行的操作就是对于所有1所占的区间左端点为L的行，用选出的行消去这一行，即这一行$\mathrm{[L, R]}$内的 1 全部变成0，相当于把所有左端点为L的区间左端点改成$\mathrm{R + 1}$。

这个操作可以用左偏树优化：我们开 n 棵左偏树，第i棵左偏树储存所有左端点为i的区间（显然可能有些左偏树为空），左偏树中以区间右端点为关键字排序。因为我们当前选出

的区间$[L,R]$满足$[L,R]$是第 L 棵左偏树中关键字最小的，所以删掉$[L,R]$即为弹出第L个堆的堆顶，把所有左端点为L的区间左端点改成$R+1$即为把第L个和第$R+1$个堆合并之后的堆作为第$R+1$个堆。可以使用基本操作实现。

单组数据复杂度$O(n\log n)$。

【参考程序】

```
// 陈栉旷
#include <bits/stdc++.h>

template <class T>
inline void read(T &res)
{
    res = 0; bool bo = 0; char c;
    while (((c = getchar()) < '0' || c > '9') && c != '-');
    if (c == '-') bo = 1; else res = c - 48;
    while ((c = getchar()) >= '0' && c <= '9')
        res = (res << 3) + (res << 1) + (c - 48);
    if (bo) res = ~res + 1;
}

template <class T>
inline void Swap(T &a, T &b) {T t = a; a = b; b = t;}

const int N = 1e5 + 5;

int n, rt[N], val[N], dis[N], lc[N], rc[N], pos[N], id[N];

int mer(int x, int y)
{
    if (!x || !y) return x ^ y;
    if (val[x] >val[y]) Swap(x, y);
    rc[x] = mer(rc[x], y);
    if (dis[lc[x]] < dis[rc[x]]) Swap(lc[x], rc[x]);
    dis[x] = rc[x] ? dis[rc[x]] + 1 : 0;
    return x;
}

void work()
{
    int x, y;
    bool res = 1;
    read(n);
    for (int i = 1; i<= n; i++) rt[i] = 0;
    for (int i = 1; i<= n; i++)
    {
```

```
        read(x); read(y);
        dis[i] = lc[i] = rc[i] = 0; val[i] = y;
        rt[x] = mer(rt[x], i); pos[i] = id[i] = i;
    }
    for (int i = 1; i<= n; i++)
    {
        if (!rt[i] || val[rt[i]] <i) return (void) puts("D");
        int r = val[rt[i]], firs = pos[i];
        if (firs != rt[i]) res ^= 1, Swap(pos[i], pos[id[rt[i]]]),
            Swap(id[firs], id[rt[i]]);
        rt[i] = mer(lc[rt[i]], rc[rt[i]]);
        if (r < n) rt[r + 1] = mer(rt[r + 1], rt[i]);
    }
    puts(res ? "Y" : "F");
}

int main()
{
    #ifdef pyztxdy
    #else
        freopen("fiend.in", "r", stdin);
        freopen("fiend.out", "w", stdout);
    #endif

    int T; read(T);
    while (T--) work();
    return 0;
}
```

逃跑计划（escape，1.5s，512MB）

【算法分析】

我们将树看做以1 为根的有根树。

对于一个最优方案，我们一定可以将其划分成若干个连通块，然后按一定顺序访问这些连通块。给每个连通块定义两个值p, g，表示进入这个连通块要先消耗p的体力值后，才能获得g的体力值。

只有$p \le g$的时候才有必要访问一个连通块。并且，我们一定能找到一个最优方案，满足访问的连通块的p是递增的，如果不是递增的，我们可以通过调整成递增的，使得答案不会变劣。

对于以结点u为根的子树，维护出以它为根的一个序列，序列中的每个元素是一个连通块，并且序列中的连通块我们可以按顺序访问，表示从u走到u的子树中，最优的连通块访问序列。显然，这个序列每个元素都要满足$p \le g$，并且p递增。

从叶子结点往上求每个子树对应的序列。

假设现在要求u对应的最优访问序列，并且对于u的每个儿子v，它的最优访问序列已经

求出。

由于儿子之间的连通块的访问顺序可以任意调整，可以先将所有儿子的序列合并，并保持p的递增性质。现在我们先考虑把u单独看成一个连通块放在序列的开头，当且仅当出现两种情况之一，我们要不断合并开头的两个连通块u, v：

1. $p_u > g_u$，由于我们要访问u这个子树必须要先访问u对应的连通块，所以我们要通过合并，来使访问u对应的连通块的收益是非负的。
2. $p_u > p_v$，由于访问u子树中除u以外的其他连通块，必须先访问结点u。为了同时维护序列中p的递增性和序列中可以按顺序访问的性质，我们应当合并u, v两个连通块。

如果合并完仍有$p_u > g_u$则舍弃这个子树。

注意合并两个连通块时，要维护好p, g两个值的实际含义。

1. 若$g_u \le p_v$，因为我们要按$u \to v$的顺序访问连通块，所以就相当于，必须先消耗$p_u + p_v - g_u$的体力值，才能获得g_v的收益。
2. 若$g_u > p_v$，我们可以先在u中取得g_u的收益，然后在收益中扣掉p_v的代价，等价于消耗p_u的体力值，获得$g_u - p_v + g_v$的收益。

为了知道能否到达结点t，我们可以在结点t的儿子中加一个$g = +\infty$的虚拟节点。

最后得到结点1的序列后，算一下最后的总体力值，如果这个值接近$+\infty$，说明能够到达添加的虚拟结点，那也说明能够到达结点t。

那么我们只要一个数据结构，支持查询、删除最小值以及合并。

使用左偏树可以达到$O(n \log n)$的时间复杂度。

【参考程序】

//陈煜翔

```
#include <bits/stdc++.h>

template <class T>
inline void read(T &x)
{
    static char ch;
    static bool opt;
    while (!isdigit(ch = getchar()) &&ch != '-');
    x = (opt = ch == '-') ?0 :ch - '0';
    while (isdigit(ch = getchar()))
        x = x * 10 + ch - '0';
    if (opt)
        x = ~x + 1;
}

#define trav(u) for (int e = adj[u], v; v = to[e], e; e = nxt[e])

typedef long long s64;

const s64 INF = 1LL << 60;
constintMaxNV = 2e5 + 5;
constintMaxNE = MaxNV<< 1;
```

```
int n, des;
s64 p[MaxNV], g[MaxNV];
intnE, adj[MaxNV], to[MaxNE], nxt[MaxNE];

intans[MaxNV];
int dis[MaxNV], lc[MaxNV], rc[MaxNV];

inline void addEdge(int u, int v)
{
    nxt[++nE] = adj[u];
    adj[u] = nE;
    to[nE] = v;
}

inline int merge(int x, int y) //合并 x,y 两个左偏树
{
    if (!x || !y)
        return x + y;
    if (p[x] > p[y])
        std::swap(x, y);

    rc[x] = merge(rc[x], y);
    if (dis[rc[x]] > dis[lc[x]])
        std::swap(lc[x], rc[x]);
    dis[x] = dis[rc[x]] + 1;

    return x;
}

inline intdel_min(int&x) //删去并返回左偏树 x 的最小结点
{
    int res = x;
    x = merge(lc[x], rc[x]);
    return res;
}

inline void dfs(int u, int pre)
{
    int son = 0;

    ans[u] = u;
    trav(u)
        if (v != pre)
        {
```

```
            dfs(v, u);
            son = merge(son, ans[v]); //先合并 u 的所有儿子
        }

    while (son && (p[u] > g[u] ||p[u] > p[son]))
    {//当 p[u]>g[u]或 p[u]>p[son]时，需要合并 u 和儿子的左偏树
        int t = del_min(son);
        if (p[t] > g[u])//合并时分两类讨论
        {
            p[u] += p[t] - g[u];
            g[u] = g[t];
        }
        else
            g[u] += g[t] - p[t];
    }

    if (p[u] > g[u])
        ans[u] = 0;
    else
        ans[u] = merge(u, son); //ans[u]表示 u 合并结束后的左偏树
}

intmain()
{
    freopen("escape.in", "r", stdin);
    freopen("escape.out", "w", stdout);

    dis[0] = -1;

    int T;
    read(T);
    while (T--)
    {
        read(n), read(des);

        nE = 0;
        for (inti = 1; i<= n + 1; ++i)
        {
            adj[i] = 0;
            g[i] = p[i] = 0;
            lc[i] = rc[i] = dis[i] = 0;
        }

        for (inti = 1; i<= n; ++i)
```

```
        {
            read(g[i]);
            if (g[i] < 0)
            {
                p[i] = -g[i];
                g[i] = 0;
            }
        }
        for (inti = 1; i< n; ++i)
        {
            int u, v;
            read(u), read(v);
            addEdge(u, v);
            addEdge(v, u);
        }

        addEdge(des, ++n);
        g[n] = INF; //新建一个收益无穷大的虚拟结点作为des的儿子

        dfs(1, 0);

        s64 now = 0;
        while (ans[1] && now >= p[ans[1]])
        {
            int t = del_min(ans[1]);
            now += g[t] - p[t]; //计算最后的总体力
        }

        puts(now > INF / 2 ? "escaped" : "trapped");
    }

    return 0;
}
```