HUNAN UNIVERSITY

程序设计训练报告

学生姓名 符航康

学生学号 202408040228

专业班级 计算机科学与技术（人工智能班）2402 班

指导老师 屈卫兰

助　　教 江政良 、戴龙宝

2025 年 9 月 15 日

小学期训练5-目录

小学期训练 5-报告

Dijkstra? (ID: H5-01)

一、问题描述

【问题描述】

给定一个含权的无向图，顶点编号为 1～n，你的任务为找出顶点 1 到顶点 n 之间的最短路径。

【输入形式】

输入的第一行为两个整数 n 和 m （2 ≤ n ≤ 10^5， 0 ≤ m ≤ 10^5），其中 n 为顶点数，m 是边数。

接下来的 m 行包含用形式 ai、bi 和 wi （1≤ai、bi ≤n，1≤ wi ≤ 10^6），这里 ai、bi 是边的端点，而 wi 是边的长度。

该图可能包括环，或者一对顶点之间包含多条边。

【输出形式】

如果无路径，输出 -1，否则输出最短路径，如果有多个，则输出字典序最小的路径。

对于两个整数序列 A （a1、a2、...）和 B （b1、b2、...），称序列 A 字典序小于序列 B 当且仅当，存在 k ≥ 1，i <k 时，ai = bi，i =k 时，ai < bi 。

【样例输入】

代码块

```
5 6
1 2 2
2 5 5
2 3 4
1 4 1
4 3 3
3 5 1
```

【样例输出】

代码块

```
1 4 3 5
```

二、问题分析与边界条件

问题核心： 本题要求在一个含权的无向图中，找出从顶点 1 到顶点 n 的最短路径。当存在多条长度相同的最短路径时，需要输出节点序列字典序最小的那一条。

思路拆解： 直接在 Dijkstra 算法的松弛操作中加入字典序的判断是行不通的，因为局部的字典序最优选择无法保证全局路径的字典序最优。为了解决这个问题，我们可以将问题分解为以下三个步骤：

1. **计算最短路径长度**： 首先，我们需要确定从顶点 1 到顶点 n 的最短路径长度到底是多少。
2. **筛选有效路径**： 在确定了最短路径长度后，我们需要一种方法来判断图中的任意一条边（u， v）是否存在于某条最短路径上。
3. **构建字典序最小路径**： 在所有满足最短路径长度的边构成的子图中，从起点 1 开始，通过贪心策略构建出一条字典序最小的路径。

边界条件：

1. **路径不存在**： 顶点 1 与顶点 n 之间可能不连通。此时，无法找到路径，应输出 -1。代码中通过判断最短路径长度是否为无穷大（`INF`）来处理。
2. **图的特性**： 题目明确指出图可能包含环或重边。我们使用邻接表和优先队列实现的 Dijkstra 算法可以正确处理这些情况。
3. **数据规模**： 顶点数 `n` 和边数 `m` 可达 10^5，需要使用时间复杂度较低的算法，如堆优化的 Dijkstra（O（M log N））。同时，路径长度可能超过 32 位整型范围，需要使用 `long long` 类型来存储距离。

三、算法设计

核心思想： 本解决方案采用"**正向 Dijkstra + 反向 Dijkstra + 贪心构造**"的策略，将问题完美分解。

1. **正向 Dijkstra**: 从起点 `1` 出发，运行一次 Dijkstra 算法，计算出起点到所有其他节点的最短距离 `distFromSource`。这样我们就得到了 `1->n` 的最短路径总长度 `shortestPathLength`。
2. **反向 Dijkstra**: 从终点 `n` 出发，在反向图（对于无向图即原图）上运行一次 Dijkstra 算法，计算出所有节点到终点 `n` 的最短距离 `distToDest`。
3. **贪心路径构建**： 有了以上两个距离数组，我们可以精确地判断任意一条边 `(u, v)` 是否位于某条 `1->n` 的最短路径上。其充要条件是：`distFromSource[u] + weight(u, v) + distToDest[v] == shortestPathLength`。基于此，我们从起点 `1` 开始，每一步都贪心地选择满足该条件的、且节点编号最小的邻居作为路径的下一个节点，直到到达终点 `n`。这个贪心策略是正确的，因为它在每一步都保证了路径的局部字典序最小，同时又不偏离任何一条最短路径。

数据结构：

- `vector<vector<Edge>> adj`： 使用邻接表存储图。`Edge` 是一个 `pair<int, int>`，表示 `{邻居节点, 边权重}`。邻接表适合存储稀疏图。
- `priority_queue<State>`： 使用最小优先队列（小顶堆）来实现 Dijkstra 算法的核心部分，用于高效地获取当前距离最小的待处理节点。`State` 是一个 `pair<long long, int>`，表示 `{距离, 节点编号}`。
- `vector<long long> distFromSource`， `vector<long long> distToDest`： 两个长整型数组，分别存储正向和反向 Dijkstra 算法计算出的最短距离。

算法流程：

1. 读取 `n` 和 `m`，并构建邻接表 `adj` 来表示图。由于是无向图，反向图与原图相同。
2. 调用 `dijkstra(1, n, adj)`，从起点 1 开始计算，结果存入 `distFromSource`。
3. 检查 `distFromSource[n]`。如果为无穷大，说明 1 和 n 不连通，输出 -1 并结束程序。否则，将其值赋给 `shortestPathLength`。
4. 调用 `dijkstra(n, n, adj)`，从终点 n 开始计算，结果存入 `distToDest`。
5. 初始化路径 `path`，并将起点 1 加入。设置 `currentNode = 1`。
6. 进入循环，只要 `currentNode` 不等于终点 `n`： a。 遍历 `currentNode` 的所有邻居 `neighbor`。 b. 对于每个邻居，检查是否满足最短路径条件：`distFromSource[currentNode] + weight + distToDest[neighbor] == shortestPathLength`。 c. 在所有满足条件的邻居中，找到节点编号最小的一个，记为 `nextNode`。 d. 将 `nextNode` 加入 `path`，并更新 `currentNode = nextNode`。
7. 循环结束后， `path` 中存储的就是字典序最小的最短路径。遍历并输出 `path` 中的所有节点。

四、核心代码讲解

8. **Dijkstra 算法通用实现**

代码块

```
vector<long long> dijkstra(int startNode, int n, const vector<vector<Edge>>& a
    vector<long long> dist(n + 1, INF);
    priority_queue<State, vector<State>, greater<State>> pq;

    dist[startNode] = 0;
    pq.push({0, startNode});

    while (!pq.empty()) {
        long long currentDist = pq.top().first;
        int u = pq.top().second;
        pq.pop();

        if (currentDist > dist[u]) {
            continue;
```

```
        }

        for (const auto& edge : adj[u]) {
            int v = edge.first;
            int weight = edge.second;
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                pq.push({dist[v], v});
            }
        }
    }
    return dist;
}
```

- **功能**： 这是一个标准的堆优化 Dijkstra 算法模板。输入起点、节点总数和图的邻接表，返回一个包含起点到所有节点最短距离的数组。
- **`dist` 数组**： 初始化为无穷大 `INF`， `dist[i]` 记录起点到节点 `i` 的当前最短距离。
- **`priority_queue`**： `pq` 是一个小顶堆，存储 `{距离, 节点}` 对，距离小的元素会被优先置于堆顶。
- **`dist[startNode] = 0`**： 起点到自身的距离为 0，并将其入队作为算法的开端。
- **`while (!pq.empty())`**： 循环直到所有可达节点都被访问。
- **`if (currentDist > dist[u])`**： 一个重要的优化。如果从队列中取出的距离 `currentDist` 大于 `dist[u]` 中记录的距离，说明这是一个过时的、较长的路径信息，直接跳过。
- **松弛操作**： `if (dist[u] + weight < dist[v])` 是核心的松弛操作。如果通过当前节点 `u` 可以找到一条到达邻居 `v` 的更短路径，则更新 `dist[v]` 并把新的状态 `{dist[v], v}` 压入优先队列。

9. 贪心构建路径

代码块

```
vector<int> path;
int currentNode = 1;
path.push_back(currentNode);

while (currentNode != n) {
    int nextNode = -1;
    // 遍历当前节点的所有邻居，寻找最优的下一个节点
    for (const auto& edge : adj[currentNode]) {
        int neighbor = edge.first;
        int weight = edge.second;

```

```
            // 检查该邻居是否在某条最短路径上
            if (distFromSource[currentNode] + weight + distToDest[neighbor] == sho
                // 在所有有效的邻居中，选择编号最小的
                if (nextNode == -1 || neighbor < nextNode) {
                    nextNode = neighbor;
                }
            }
        }
        // 移动到选择的下一个节点
        currentNode = nextNode;
        path.push_back(currentNode);
    }
```

- **功能**： 这段代码实现了算法的第三步，即在确定了最短路径上的所有"合法"边之后，贪心地构建出字典序最小的路径。
- `while (currentNode != n)`： 循环从起点 1 开始，直到到达终点 n 为止。
- `nextNode = -1`： 在每一步迭代开始时，用于记录下一个节点的候选者，初始化为 -1。
- `if (distFromSource[...] == shortestPathLength)`： 这是路径构建的关键判断。它利用预先计算好的两个距离数组，确保选择的边 `(currentNode, neighbor)` 一定位于某条从 1 到 n 的最短路径上。
- `if (nextNode == -1 || neighbor < nextNode)`： 这个判断实现了贪心选择。在所有"合法"的邻居中，选择节点编号最小的一个作为 `nextNode`。
- `path.push_back(currentNode)`： 将选出的最优节点加入最终路径。

五、复杂度分析

时间复杂度： O（M log N）

- 算法的主体是两次独立的 Dijkstra 算法调用。
- 每一次堆优化的 Dijkstra 算法中，每个顶点最多入队一次，每条边最多被访问一次（对于无向图是两次）。
- 优先队列的操作（插入 `push` 和提取最小 `pop`）的时间复杂度为 `O(log V)`，其中 `V` 是队列中的元素数量，最多为 `N`。
- 因此，单次 Dijkstra 的时间复杂度为 `O(M log N)`。
- 两次 Dijkstra 的总时间复杂度为 `2 * O(M log N)`，即 `O(M log N)`。
- 最后构建路径的步骤，最多遍历所有边一次，时间复杂度为 `O(N + M)`。
- 综上，算法的瓶颈在于 Dijkstra，总时间复杂度为 `O(M log N)`。

空间复杂度： O（N + M）

- 存储图的邻接表 `adj` 和 `adjRev` 需要 `O(N + M)` 的空间。

- 两个距离数组 `distFromSource` 和 `distToDest` 分别需要 `O(N)` 的空间。
- Dijkstra 算法中的优先队列在最坏情况下可能存储 `N` 个节点，需要 `O(N)` 的空间。
- 存储最终路径的 `path` 向量最多存储 `N` 个节点，需要 `O(N)` 的空间。
- 因此，总的空间复杂度为 `O(N + M)`。

六、总结与反思

方法总结： 本题是经典 Dijkstra 算法的一个巧妙变种，它考察了如何在满足最短路径约束的同时，优化第二个维度（字典序）。核心的解决方法是"**预计算+贪心**"：通过两次 Dijkstra（正向和反向）预计算出全局信息（每个点到起点和终点的最短距离），然后利用这些信息指导后续的局部贪心选择，从而保证局部最优能够推导出全局最优。这种"双向搜索"或"meet-in-the-middle"的思想在解决许多图论路径问题时非常有效。

优化与拓展：

- 本题的解法在时间和空间上已经相当优越，对于给定的数据范围是完全足够的。
- 一个有趣的拓展是，如果问题要求输出所有最短路径中字典序第 `k` 小的路径，该如何解决？这会是一个更复杂的问题，可能需要结合搜索和剪枝技术。

学习收获：

1. **深化了对 Dijkstra 算法的理解**： 不仅仅是会背模板，而是能理解其原理，并将其作为工具解决更复杂的问题。
2. **学习了处理复合优化目标的策略**： 当问题有多个优化目标（如：长度最短、字典序最小）时，通常需要先满足最高优先级的目标（最短路径），然后在满足该目标的解空间内，再优化次要目标（字典序）。
3. **掌握了"双向 Dijkstra"的应用**： 通过从起点和终点同时运行最短路算法，可以高效地判断某点或某边是否在最短路径上，这是一个非常实用且强大的技巧。

踩点上课 （ID: H5-02）

一、问题描述

二、问题分析与边界条件

问题核心： 本题要求计算在一个带有特殊"传送门"规则的网格图中，从左上角 （1，1） 到右下角 （n，m） 的最短路径成本。

思路拆解： 这是一个图论中的最短路问题。路径由两种移动方式构成：

1. **普通移动**： 在相邻的、非障碍的单元格之间移动，每次移动成本固定为 `w`。

2. **传送门移动**： 从一个入口 `P_i` （值为正数的单元格）可以移动到另一个入口 `P_j` ，成本为 `grid[P_i] + grid[P_j]` 。

关键的洞察在于，任何一条最优路径**最多只会使用一次传送门移动。**因为多次传送（如 S -> P1 -> P2 -> P3 -> P4 -> D）的成本为 `(val(P1)+val(P2)) + (val(P3)+val(P4))` ，这总是大于或等于单次传送（如 S -> P1 -> P4 -> D）的成本 `val(P1)+val(P4)` 。

基于此，问题被分解为两种独立的可能性：

1. 完全不使用传送门，只通过普通移动从起点到达终点。
2. 使用一次传送门，路径形如：起点 -> （普通移动） -> 入口 A -> （传送） -> 入口 B -> （普通移动） -> 终点。

边界条件： 根据题目和代码逻辑，需要考虑以下边界情况：

- **路径不通**：
  - 起点和终点之间可能没有任何普通路径。
  - 传送网络可能无法从起点或终点访问，即所有入口都无法通过普通移动到达。
  - 最终可能无法到达终点，此时应输出 -1。
- **无传送门**： 地图中可能没有任何入口（值 > 0 的单元格），此时只能进行普通移动。
- **数值溢出**： 路径成本 `w` 和入口值都很大，路径总成本可能超过 32 位整型范围，因此代码中必须使用 `long long` 来存储距离。
- **起点/终点即为入口**： 起点或终点本身也可能是入口，算法需要能正确处理这种情况。

三、算法设计

核心思想： 算法的核心是分别计算"不使用传送门"和"使用一次传送门"两种策略下的最短路径，然后取两者的最小值作为最终答案。

1. **不使用传送门**： 这等价于在网格图上寻找从起点到终点的最短路，由于所有边的权重（普通移动成本 `w` ）都相同，可以使用**广度优先搜索 （BFS）** 高效求解。
2. **使用一次传送门**： 这条路径的成本可以分解为三部分：
   - `成本1` ： 从起点 `S` 普通移动到某个入口 `P_i` 的最短距离。
   - `成本2` ： `P_i` 和 `P_j` 之间的传送成本，即 `value(P_i) + value(P_j)` 。
   - `成本3` ： 从某个入口 `P_j` 普通移动到终点 `D` 的最短距离。
3. 为了找到使用传送门的最低总成本，我们需要找到 `min(成本1 + 成本2 + 成本3)` 。这可以进一步优化为 `min(成本1 + value(P_i)) + min(成本3 + value(P_j))` 。
   - `min(从S到入口Pi的最短路 + value(Pi))`
   - `min(从D到入口Pj的最短路 + value(Pj))`

4. 这两部分的计算都需要知道起点/终点到所有其他点的最短普通移动距离。因此，我们可以运行两次 BFS 来预计算这些距离。

数据结构:

- `std::vector<std::vector<int>> grid`： 存储输入的 n x m 网格。
- `std::vector<std::vector<long long>> dist`： 二维数组，用于存储 BFS 计算出的从某个源点到图中所有点的最短距离。
- `std::queue<std::pair<int, int>>`： BFS 的核心数据结构，用于按层次遍历网格节点。

算法流程:

1. **初始化**： 读取输入 `n`， `m`， `w` 和网格数据。定义一个足够大的数 `INF` 代表无穷大，用于表示不可达。
2. **第一次 BFS**: 以起点 `(0, 0)` 为源点，运行一次 BFS，计算出 `dist_from_S` 数组，其中 `dist_from_S[i][j]` 表示从起点到 `(i, j)` 的最短普通移动距离。
3. **第二次 BFS**: 以终点 `(n-1, m-1)` 为源点，运行一次 BFS，计算出 `dist_from_D` 数组，其中 `dist_from_D[i][j]` 表示从终点到 `(i, j)` 的最短普通移动距离。
4. **计算无传送门成本**： `cost_no_portal = dist_from_S[n-1][m-1]`。如果此值为 `INF`，则表示无法仅通过普通移动到达。
5. **计算使用传送门成本**:
   - 初始化 `min_S_to_portal = INF` 和 `min_D_to_portal = INF`。
   - 遍历整个网格，如果单元格 `(i, j)` 是一个入口 （ `grid[i][j] > 0` ）：
     - 如果该入口可以从起点到达 （`dist_from_S[i][j] != INF`），则更新 `min_S_to_portal = min(min_S_to_portal, dist_from_S[i][j] + grid[i][j])`。
     - 如果该入口可以从终点到达 （`dist_from_D[i][j] != INF`），则更新 `min_D_to_portal = min(min_D_to_portal, dist_from_D[i][j] + grid[i][j])`。
   - `cost_with_portal = min_S_to_portal + min_D_to_portal`.
6. **确定最终答案**： 最终结果为 `min(cost_no_portal, cost_with_portal)`。如果结果仍然是 `INF`，则表示无论如何都无法到达终点，输出 -1。否则，输出计算出的最小值。

四、核心代码讲解

7. BFS 广度优先搜索函数

代码块

```
vector<vector<ll>> bfs(pair<int, int> start, int n, int m, ll w, const vector<
    vector<vector<ll>> dist(n, vector<ll>(m, INF));
```

```
    queue<pair<int, int>> q;

    int start_r = start.first;
    int start_c = start.second;

    if (grid[start_r][start_c] == -1) {
        return dist;
    }

    dist[start_r][start_c] = 0;
    q.push({start_r, start_c});

    int dr[] = {-1, 1, 0, 0};
    int dc[] = {0, 0, -1, 1};

    while (!q.empty()) {
        pair<int, int> curr = q.front();
        q.pop();
        int r = curr.first;
        int c = curr.second;

        for (int i = 0; i < 4; ++i) {
            int nr = r + dr[i];
            int nc = c + dc[i];

            if (nr >= 0 && nr < n && nc >= 0 && nc < m && grid[nr][nc] != -1 &
                dist[nr][nc] = dist[r][c] + w;
                q.push({nr, nc});
            }
        }
    }
    return dist;
}
```

代码分析:

- 这个函数是计算网格图中单源最短路径的标准 BFS 实现。
- `dist` 数组初始化为 `INF`，用于标记未访问过的节点并存储最短距离。
- `q` 是一个队列，存储待访问的节点坐标。
- 起点首先入队，其距离设为 0。
- `dr` 和 `dc` 数组定义了上、下、左、右四个方向的移动。
- 主循环 `while (!q.empty())` 不断从队列中取出节点，并探索其所有未被访问过的、非障碍的邻居节点。

- 当找到一个合法的邻居 `(nr, nc)` 时，更新其距离 `dist[nr][nc] = dist[r][c] + w`，并将其入队。
- 函数返回 `dist` 数组，包含了从 `start` 点到所有可达点的最短距离。

8. 主逻辑：计算两种策略的成本

代码块

```
void solve() {
    // ... 输入读取 ...

    // 1. 从起点 (0,0) 开始运行 BFS
    vector<vector<ll>> dist_from_S = bfs({0, 0}, n, m, w, grid);

    // 2. 从终点 (n-1, m-1) 开始运行 BFS
    vector<vector<ll>> dist_from_D = bfs({n - 1, m - 1}, n, m, w, grid);

    // 情况A: 完全不使用传送门的路径成本
    ll cost_no_portal = dist_from_S[n - 1][m - 1];

    // 情况B: 使用一次传送门的路径成本
    ll min_S_to_portal = INF;
    ll min_D_to_portal = INF;

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            if (grid[i][j] > 0) { // 如果是入口
                if (dist_from_S[i][j] != INF) {
                    min_S_to_portal = min(min_S_to_portal, dist_from_S[i][j]
                }
                if (dist_from_D[i][j] != INF) {
                    min_D_to_portal = min(min_D_to_portal, dist_from_D[i][j]
                }
            }
        }
    }

    ll cost_with_portal = INF;
    if (min_S_to_portal != INF && min_D_to_portal != INF) {
        cost_with_portal = min_S_to_portal + min_D_to_portal;
    }

    // 4. 最终答案是两种情况成本的较小值
    ll final_ans = min(cost_no_portal, cost_with_portal);

    if (final_ans >= INF) {
        cout << -1 << endl;
    } else {
```

```
41            cout << final_ans << endl;
42        }
43    }
```

代码分析:

- 这段代码清晰地实现了算法设计的流程。
- 首先调用两次 `bfs` 函数，分别获取从起点和终点出发的距离图。
- `cost_no_portal` 直接从 `dist_from_S` 中查询终点位置的距离。
- 接着，通过一个双层循环遍历所有网格单元。当找到一个入口 （`grid[i][j] > 0`） 时，结合预计算的 `dist_from_S` 和 `dist_from_D`，计算并更新到达该入口并激活它的最低成本（`min_S_to_portal` 和 `min_D_to_portal`）。
- `cost_with_portal` 由 `min_S_to_portal` 和 `min_D_to_portal` 相加得到，前提是两者都不是 `INF`。
- 最后，比较 `cost_no_portal` 和 `cost_with_portal`，取较小值作为最终答案，并处理无法到达的情况。

五、复杂度分析

- **时间复杂度： O（N * M）**
  - **推导过程**： 算法主要由三次遍历网格的操作构成。第一次是 `bfs` from start，第二次是 `bfs` from end。每次 BFS 都会访问每个节点和每条边一次，在网格图中，节点数是 `N*M`，边数约是 `2*N*M`，所以单次 BFS 的时间复杂度是 `O(N*M)`。第三次是遍历所有单元格以计算传送门成本，其复杂度也是 `O(N*M)`。因此，总的时间复杂度由 BFS 主导，为 `O(N*M) + O(N*M) + O(N*M) = O(N*M)`。
- **空间复杂度： O（N * M）**
  - **推导过程**： 算法需要额外的空间来存储从起点和终点出发的距离数组。 `dist_from_S` 和 `dist_from_D` 都是 `N x M` 大小的二维数组。BFS 中使用的队列在最坏情况下也可能存储 `O(N*M)` 个节点。因此，主要的额外空间开销与网格大小成正比，空间复杂度为 `O(N*M)`。

六、总结与反思

方法总结： 本题是一个在特殊规则下的网格图最短路问题。解决此类问题的关键在于**问题分解**和**模型抽象**。通过分析发现最优路径最多只经过一次"传送"，成功地将一个复杂问题分解为两个更简单的子问题：纯粹的网格最短路和"起点-传送-终点"模式的最短路。对于边权统一的网格图，**广度优先搜索（BFS）** 是计算最短路径最直接、最高效的算法。

学习收获:

1. **复杂问题的简化能力**： 学会了通过分析问题特性（最优子结构、路径特点）来简化问题模型，这是解决复杂算法问题的核心能力。

2. **BFS 的灵活应用**： 本题中两次从不同源点（起点和终点）执行 BFS，以预计算所需的所有路径信息，体现了基础算法的灵活组合应用。
3. **代码实现细节**： 注意到了路径成本可能导致的整数溢出问题，并使用 `long long` 类型来保证计算的正确性。同时，通过定义 `INF` 来优雅地处理不可达状态，使代码逻辑更清晰。

数组跳远 （ID: H5-03）

一、问题描述

对于一个具有 n 个元素的数组 a， 执行以下操作：

首先，选择下标 i （1 ≤ i ≤n）—— 设置为数组的开始位置，放一个标记在 i 处（在值 ai 的地方）

当 i ≤ n 时，你的得分将增加 ai ，且将标记向右移动 ai 个位置，也就是说用 i +ai 替换 i ，继续这个过程

如果 i >n，则结束操作

例如， 如果 n = 5 且 a=[7， 3， 1， 2， 3]，则可以进行以下操作

选择 i = 1，操作过程为，最后得分为 a1 = 7

选择 i = 2，操作过程为， 最后得分为 a2 + a5 = 6

选择 i = 3，操作过程为， 最后得分为 a3 + a4 = 3

选择 i = 4，操作过程为， 最后得分为 a4 = 2

选择 i = 5，操作过程为， 最后得分为 a5 = 3

请选择合适的开始位置，使得经过上述操作后可获得最大的分数。

【输入形式】

输入的第一行为一个整数 t （1≤ t ≤ 104），表示测试用例的组数。

每个测试用例的第一行为一个整数 n （1≤ n ≤ 2×105），表示数组 a 的元素个数

接下来一行包含 n 个整数 a1、a2、...、an （1 ≤ ai ≤ 109），表示数组 a 的元素

【输出形式】

对于每个测试用例，输出独立一行，表示选择合适的开始位置后经过上述操作可以获得的最大分数。

【样例输入】

代码块

```
1   4
2   5
3   7 3 1 2 3
```

```
3
2 1 4
6
2 1000 2 3 995 1
5
1 1 1 1 1
```

【样例输出】

代码块

```
7
6
1000
5
```

二、问题分析与边界条件

问题核心

本题的核心要求是：对于一个给定的数组，我们需要从每一个可能的位置 `i` 出发，按照规则 `i_next = i + a[i]` 进行跳跃，并累加路径上所有位置的值作为得分。最终，我们需要找出所有可能出发点中能得到的最高总分。

思路拆解

1. **计算每个起点的得分**：对于数组中的每一个下标 `i`（从 `0` 到 `n-1`），都需要计算出从该点出发的完整跳跃路径所能获得的总分数。
2. **避免重复计算**：一个朴素的解法是为每个起点 `i` 都独立模拟一次跳跃过程。但这样做效率低下，因为不同的跳跃路径可能会在某个点 `k` 交汇。例如，从 `i` 跳一步到达 `k`，从 `j` 跳一步也可能到达 `k`。如果每次都重新计算，从 `k` 点开始的后续路径得分就会被重复计算。
3. **优化思路**：为了避免重复计算，我们可以存储已经计算过的路径得分。这正是动态规划（Dynamic Programming）思想的用武之地。我们可以定义 `dp[i]` 为从位置 `i` 出发能获得的总分。

边界条件

- **数据规模**：数组长度 `n` 最大可达 `2 * 10^5`。这意味着一个时间复杂度为 `O(N^2)` 的朴素算法（即对每个点都进行一次完整模拟）将会超时，必须寻求 `O(N)` 或 `O(N log N)` 的高效解法。
- **数值范围**：数组元素 `a[i]` 的值最大可达 `10^9`。由于得分是累加的，总分很容易超过标准 `int` 类型的最大值（约 `2 * 10^9`）。因此，所有与分数相关的变量（包括数组 `a` 和 `dp` 数组）都必须使用 `long long` 类型来存储，以防止整数溢出。

- **跳跃终止**：跳跃的终止条件是当前位置 `i` 超出数组上界（`i >= n`）。任何一次跳跃 `i + a[i]` 都有可能直接满足该条件。

三、算法设计

核心思想

本题采用**动态规划**的策略进行求解。传统的动态规划通常是自底向上、从前向后计算。但本题的特殊之处在于，一个位置 `i` 的状态依赖于它未来的位置 `j = i + a[i]`。如果我们从前往后计算，当计算 `dp[i]` 时，`dp[j]` 还是未知的。

因此，我们采用**从后向前**的计算顺序。我们定义 `dp[i]` 为从下标 `i` 出发能获得的总分数。状态转移方程如下：

- 首先，从 `i` 出发，必然会获得 `a[i]` 的分数。
- 然后，跳到新的位置 `j = i + a[i]`。
  - 如果 `j` 仍然在数组范围内（`j < n`），则从 `i` 出发的总分等于 `a[i]` 加上从 `j` 出发的后续总分 `dp[j]`。即 `dp[i] = a[i] + dp[j]`。
  - 如果 `j` 超出了数组范围（`j >= n`），则跳跃在此处结束，总分就是 `a[i]`。即 `dp[i] = a[i]`。

当我们从 `i = n-1` 向 `0` 遍历时，对于任意一个 `i`，其依赖的 `dp[j]`（其中 `j > i`）都已经被计算出来了，从而保证了状态转移的正确性。

数据结构

- `std::vector<long long> a`：用于存储输入的数组元素。使用 `long long` 是为了处理 `a[i]` 本身可能很大的情况，并为后续计算总分做准备。
- `std::vector<long long> dp`：动态规划数组，`dp[i]` 用于存储从下标 `i` 出发能获得的总分数。必须使用 `long long` 防止累加得分溢出。

算法流程

1. **输入与初始化**：读取测试用例数量 `t`。在每个测试用例中，读取数组大小 `n` 和数组 `a` 的所有元素。创建一个与 `a` 等大的 `dp` 数组。
2. **动态规划计算**：从数组的末尾开始，向前遍历数组。下标 `i` 从 `n-1` 递减到 `0`。
   a. 将当前位置的分数 `a[i]` 赋给 `dp[i]`。
   b. 计算下一步的落点 `jumpDestination = i + a[i]`。
   c. 判断落点是否在数组内。如果 `jumpDestination < n`，则将后续路径的分数 `dp[jumpDestination]` 累加到 `dp[i]` 上。
3. **寻找最大值**：当循环结束后，`dp` 数组中已经存储了从每个位置出发的最终得分。使用 `std::max_element` 函数（或手动遍历）找出 `dp` 数组中的最大值。

4. **输出结果**：输出找到的最大分数值。

四、核心代码讲解

算法的核心实现是**从后向前填充 DP 数组**的循环部分。

代码块

```
// dp[i] 表示从下标 i (0-indexed) 开始跳跃的总分
vector<long long> dp(n, 0);

// 从后往前填充DP数组
for (int i = n - 1; i >= 0; --i) {
    // 当前这一跳的分数
    dp[i] = a[i];

    // 计算下一个落点
    long long jumpDestination = i + a[i];

    // 如果落点在数组范围内，加上后续跳跃的分数
    if (jumpDestination < n) {
        dp[i] += dp[jumpDestination];
    }
}
```

逐行分析：

- `for (int i = n - 1; i >= 0; --i)`： 这是整个算法的关键。通过从 `n-1` 向 `0` 遍历，我们确保了当计算 `dp[i]` 时，所有可能依赖的 `dp` 值（即下标大于 `i` 的 `dp` 值）都已经被计算完毕。
- `dp[i] = a[i];`： 初始化 `dp[i]` 的值。无论后续是否能继续跳跃，从 `i` 出发的得分至少包含 `a[i]` 本身。
- `long long jumpDestination = i + a[i];`： 计算从当前位置 `i` 跳跃一次后的目标位置。这里使用 `long long` 是为了防止 `i + a[i]` 的中间结果溢出（虽然在本题 `i` 和 `a[i]` 的约束下 `int` 足够，但这是个好习惯）。
- `if (jumpDestination < n)`： 检查跳跃是否会超出数组边界。
- `dp[i] += dp[jumpDestination];`： 这是状态转移的核心。如果跳跃没有出界，就将目标位置 `jumpDestination` 已经计算好的总分累加到当前位置 `i` 的总分上，实现了子问题解的复用。

五、复杂度分析

时间复杂度： O（N）

- 算法的主体是一个从 `n-1` 到 `0` 的单层循环。循环内部的所有操作（赋值、加法、比较）都是常数时间 `O(1)` 的。因此，填充整个 `dp` 数组的时间复杂度为 `O(N)`。
- 之后寻找 `dp` 数组中的最大值也需要遍历一次数组，时间复杂度为 `O(N)`。
- 总的时间复杂度为 `O(N) + O(N) = O(N)`。

空间复杂度： O（N）

- 算法需要一个额外的 `dp` 数组来存储每个位置出发的总分。这个数组的大小与输入数组 `a` 的大小 `n` 相同。
- 因此，算法使用的额外空间与输入规模 `n` 成正比，空间复杂度为 `O(N)`。

六、总结与反思

方法总结

本题是动态规划思想的一个典型应用。其特殊之处在于状态依赖的"方向性"。当一个问题的状态 `dp[i]` 依赖于下标比 `i` 更大的状态 `dp[j]` 时，采用**从后向前**的迭代顺序是一种非常有效且简洁的解决方法。这种"逆向思维"在处理许多动态规划或图论问题时都非常关键。

优化与拓展

- **递归与记忆化**：本题同样可以使用"自顶向下"的递归配合记忆化（Memoization）来解决，这在逻辑上与从后向前的动态规划是等价的。但通常来说，迭代式的动态规划在实现上没有递归开销，性能会稍好一些。
- **空间优化**：在本题的设定下，`dp` 数组是必需的，因为最终需要比较所有起点的得分。但在某些类似问题中，如果只关心某个特定终点或全局状态，可能存在空间优化（例如使用滚动数组）的机会。

学习收获

1. **识别动态规划模式**：学会了识别问题中存在的"重叠子问题"（不同路径汇合到同一点）和"最优子结构"（一个点的最优解依赖于其后续点的最优解），这是应用动态规划的前提。
2. **确定正确的计算顺序**：深刻理解了根据状态依赖关系来确定 DP 计算顺序（从前向后 vs 从后向前）的重要性。
3. **注意数据类型**：再次强调了在算法竞赛中仔细分析数据范围、预判结果大小并选择合适数据类型（如 `long long`）以防止溢出的重要性。这是一个非常常见的陷阱。

有效的 BFS （ID: H5-04）

一、问题描述

> 【问题描述】

在图的 BFS（广度优先搜索）中，通常采用队列来保存当前顶点的邻接点，但对对应邻接点的存入顺序没有要求，因此对于一个图的 BFS 结果可以有多个，在本问题中，从顶点 1 开始，请验证一个给定的顶点序列是否为一个有效的 BFS 序列?

【输入形式】

输入的第一行为一个整数 n （1≤ n ≤ 2 ×10^5），表示树中节点的数量。

接下来 n-1 行描述了树的边，每行包含两个整数 x 和 y （1 ≤ x， y ≤ n），表示对应边的两个端点，输入保证给定的图构成一颗树。

最后一行为 n 个互不相同的整数 a1、a2、...、an （ 1 ≤ ai ≤ n），代表待检验的顶点序列。

【输出形式】

如果待检验的序列是一个正确的 BFS 序列，输出"Yes"，否则输出"No"。

【样例输入 1】

代码块

```
4
1 2
1 3
2 4
1 2 3 4
```

【样例输出 1】

代码块

```
Yes
```

【样例输入 2】

代码块

```
4
1 2
1 3
2 4
1 2 4 3
```

【样例输出 2】

代码块

```
No
```

二、问题分析与边界条件

问题核心

本题的核心要求是判断一个给定的节点序列，是否是给定树从节点 `1` 开始的某一种合法的广度优先搜索（BFS）序列。

思路拆解

要解决这个问题，我们不能简单地自己跑一遍 BFS 然后对比序列，因为合法的 BFS 序列可能有很多种。正确的思路是，利用 BFS 的核心性质来"模拟"和"验证"给定的序列。

1. **BFS 的层级性**：BFS 是按层遍历的。也就是说，所有在第 `k` 层的节点必须在所有第 `k+1` 层的节点之前被访问。
2. **父子节点的连续性**：对于任意一个父节点 `u`，它的所有子节点（在 BFS 树中与 `u` 相连且层数更深的邻居）在序列中必须以一个"连续的块"出现。但是，这些子节点在这个块内部的相对顺序是任意的。
3. **模拟验证**：基于以上性质，我们可以用一个标准的队列来驱动 BFS 的逐层逻辑，同时用一个指针来扫描给定的序列。在每个步骤中，我们从队列中取出一个父节点 `u`，然后检查序列中紧随其后的一段是否恰好是 `u` 的所有子节点。

边界条件

根据题目和代码分析，需要考虑以下边界情况：

- **起点检查**：给定的序列 `a` 的第一个元素必须是起点 `1`。
- **单节点树**：如果 `n=1`，那么唯一的合法序列就是 `[1]`。
- **序列与子节点数量不匹配**：在处理父节点 `u` 时，如果序列 `a` 剩下的节点数量，少于 `u` 的实际子节点数量，则序列非法。
- **节点集合不匹配**：序列中对应子节点"块"内的节点集合，必须与 `u` 的实际子节点集合完全一致。
- **输入保证**：题目保证输入的是一棵树，因此我们无需担心图不连通或存在环路的问题。

三、算法设计

核心思想

算法的核心是通过模拟 BFS 的执行过程来验证给定序列 `a` 的有效性。我们维护一个标准的 BFS 队列 `q` 和一个指向序列 `a` 当前位置的指针 `seq_idx`。我们按照 `a` 的顺序来"消费"节点，并检查这个消费过程是否符合 BFS 的规则。

具体来说，当从队列 `q` 中取出一个父节点 `u` 时，我们首先确定它在树中所有未被访问过的邻居（即它的"子节点"）。假设有 `k` 个子节点，那么根据 BFS 的性质，序列 `a` 中从 `seq_idx` 开始的 `k` 个节点必须恰好是这 `k` 个子节点。由于它们的顺序不固定，我们只需验证这两组节点的**集合**

是否完全相同。如果相同，我们就将这 `k` 个节点（按照它们在序列 `a` 中的顺序）加入到队列 `q` 中，并推进 `seq_idx` 指针，继续下一轮验证。若任何一步验证失败，则该序列不是有效的 BFS 序列。

数据结构

- `vector<unordered_set<int>> adj`： 使用邻接表来存储树的结构。选择 `unordered_set` 作为邻居的容器是本算法的关键。它使得我们可以在接近 O（1） 的平均时间内添加和查找邻居，并且可以直接比较两个集合是否相等，完美地解决了"子节点顺序不固定"的问题。
- `queue<int> q`： 标准队列，用于驱动 BFS 的模拟过程，确保按层处理节点。
- `vector<bool> visited`： 标记数组，用于在遍历树时避免回溯到父节点，并正确识别出当前节点的子节点。
- `vector<int> a`： 存储待检验的顶点序列。
- `int seq_idx`： 一个整型指针，用于追踪当前在序列 `a` 中已经验证到的位置。

算法流程

1. **初始化**：读取节点数 `n` 、 `n-1` 条边和待验证序列 `a` 。构建邻接表 `adj` 。
2. **基本校验**：检查 `a[0]` 是否为 `1` ，若不是则序列无效。处理 `n=1` 的平凡情况。
3. **启动模拟**：创建一个队列 `q` 并将起点 `1` 入队。创建一个 `visited` 数组，标记 `1` 已被访问。初始化序列指针 `seq_idx = 1` 。
4. **循环验证**：当队列 `q` 非空时，执行以下操作： a。 从 `q` 中取出一个父节点 `u` 。 b. 遍历 `u` 的所有邻居，将其中未被访问过的节点收集到一个 `unordered_set` 中，记为 `actual_children` 。 c. 根据 `actual_children` 的大小 `k` ，从序列 `a` 的 `seq_idx` 位置开始，提取 `k` 个节点，放入另一个 `unordered_set` 中，记为 `expected_children` 。 d. **核心比较**：判断 `actual_children` 和 `expected_children` 这两个集合是否完全相等。若不相等，则序列无效，输出 "No" 并终止程序。 e. **推进模拟**：如果集合相等，则验证通过。依次将这 `k` 个子节点（即 `a[seq_idx]` 到 `a[seq_idx + k - 1]` ）标记为已访问，并加入队列 `q` 。 f. 更新序列指针 `seq_idx = seq_idx + k` 。
5. **最终判断**：循环结束后，检查 `seq_idx` 是否等于 `n` 。如果等于 `n` ，说明序列中的所有节点都按照 BFS 规则被成功验证，序列有效，输出 "Yes"。否则，序列无效，输出 "No"。

四、核心代码讲解

算法最关键的部分在于主循环，它同步了 BFS 的模拟过程和对给定序列的消费与验证。

代码块

```
// 用于BFS模拟的队列
queue<int> q;
q.push(1);

```

```cpp
// 标记节点是否被访问过
vector<bool> visited(n + 1, false);
visited[1] = true;

// 指向给定序列 a 中，下一个期望出现的子节点的位置
int seq_idx = 1;

while (!q.empty() && seq_idx < n) {
    int u = q.front();
    q.pop();

    // 1. 找出节点 u 在树中的所有实际子节点（即未访问过的邻居）
    unordered_set<int> actual_children;
    for (int neighbor : adj[u]) {
        if (!visited[neighbor]) {
            actual_children.insert(neighbor);
        }
    }
    int children_count = actual_children.size();

    // 2. 从序列 a 中提取出期望的子节点块
    if (seq_idx + children_count > n) { // 检查序列长度是否足够
        cout << "No" << endl;
        return;
    }
    unordered_set<int> expected_children;
    for (int i = 0; i < children_count; ++i) {
        expected_children.insert(a[seq_idx + i]);
    }

    // 3. 比较实际子节点集和期望子节点集是否完全相同
    if (actual_children != expected_children) {
        cout << "No" << endl; // 节点集合不匹配
        return;
    }

    // 4. 验证通过，将被验证的子节点加入队列，并更新访问状态和序列指针
    for (int i = 0; i < children_count; ++i) {
        int child = a[seq_idx + i];
        visited[child] = true;
        q.push(child);
    }
    seq_idx += children_count;
}
```

代码分析:

1. `while (!q.empty() && seq_idx < n)`： 循环持续的条件是模拟队列中还有待处理的父节点，并且序列 `a` 还没有被完全消费。
2. `int u = q.front(); q.pop();`： 取出当前层的父节点 `u`。
3. `actual_children` **的构建**： 这一步通过遍历 `u` 的邻接表并检查 `visited` 数组，准确地找到了 `u` 在 BFS 树中的所有子节点。
4. `expected_children` **的构建**： 根据 `actual_children` 的数量，从序列 `a` 的当前位置 `seq_idx` 提取出同样数量的节点，作为"期望的"子节点集合。
5. `if (actual_children != expected_children)`： 这是算法的**核心判断**。`std::unordered_set` 的 `!=` 操作符会比较两个集合的元素是否完全一致，而忽略元素的顺序。这完美地契合了题目中"子节点存入顺序没有要求"的规则。如果集合内容不同，立即判定序列非法。
6. **推进状态**： 如果验证通过，代码会将 `expected_children` 中的节点（按它们在序列 `a` 中的顺序）加入队列，以供下一层处理，并移动 `seq_idx` 指针跳过这个刚刚被验证过的子节点块。

五、复杂度分析

时间复杂度： O（N）

- **建图**：读取 `n` 个节点和 `n-1` 条边并构建邻接表，时间复杂度为 O（N）。
- **BFS 模拟**：在 `while` 循环中，每个节点作为父节点 `u` 只会入队和出队一次。对于每个节点 `u`，我们会遍历其所有邻居来构建 `actual_children` 集合。由于图中所有节点的度数之和为 `2 * (N - 1)`，所以遍历所有邻居的总操作次数为 O（N）。
- **集合操作**：构建 `expected_children` 集合和比较两个集合的操作，分摊到每个节点和每条边上，其总时间复杂度也为 O（N）。
- 因此，整个算法的主要操作都与节点数和边数线性相关，总时间复杂度为 O（N）。

空间复杂度： O（N）

- **邻接表** `adj`： 存储了 `n-1` 条边，需要 O（N） 的空间。
- **序列** `a`： 存储输入的序列，需要 O（N） 的空间。
- **访问数组** `visited`： 需要 O（N） 的空间。
- **队列** `q`： 在最坏的情况下（例如星形图），队列中可能存储接近 `N-1` 个节点，需要 O（N） 的空间。
- **临时集合**： `actual_children` 和 `expected_children` 集合的大小不会超过 `N`，因此也占用 O（N） 级别的空间。
- 综上，算法所需的额外空间与输入规模 `N` 成正比，空间复杂度为 O（N）。

六、总结与反思

方法总结

本题是检验对广度优先搜索（BFS）核心性质理解的经典问题。其关键点在于，不能试图生成一个 BFS 序列去进行字符串匹配，而应该利用 BFS 的**层序遍历**和**父子节点关系**这两个内在属性，去验证一个给定的序列是否满足这些属性。

解决此类"验证型"算法问题的通用方法是：**模拟算法过程，并在关键决策点进行校验。**本题中，关键决策点就是"当一个父节点处理完毕后，接下来出现的一批新节点是否是它的所有子节点"。通过使用 `unordered_set` 这一恰当的数据结构，巧妙地处理了子节点顺序不确定的问题，使得验证逻辑变得简洁而高效。

优化与拓展

- **性能优化**：本解法的时间和空间复杂度均为 O（N），已经是该问题下的最优解，无需进一步优化。
- **问题拓展**：
  a. **验证 DFS 序列**：如果问题改为验证一个序列是否为有效的深度优先搜索（DFS）序列，算法思路会完全不同。需要使用栈来模拟 DFS 过程，并检查序列中的下一个节点是否是当前栈顶节点的合法子节点。
  b. **处理普通图**：如果输入的不是树，而是包含环的普通图，本算法的核心逻辑依然适用。`visited` 数组的存在可以确保我们不会因环路而无限循环，并能正确识别出在 BFS 过程中新发现的邻居节点。

学习收获

通过解决本题，可以有以下几点收获：

1. **深化对 BFS 的理解**：不仅要知道如何实现 BFS，更要深刻理解其"逐层扩展"和"所有子节点作为一个整体被发现"的核心性质。
2. **数据结构的选择**：学习到如何根据问题特性（如本题的"顺序无关性"）选择最合适的数据结构（`unordered_set`），从而简化问题逻辑，提高代码效率和可读性。
3. **模拟与验证的思维**：掌握了一种重要的算法设计思想，即通过模拟一个标准过程并在关键步骤插入校验逻辑，来解决验证类问题。

旅行的期望 （ID: H5-05）

一、问题描述

> **问题背景** 在一个由 N 个城市和 N-1 条双向道路构成的国家中，所有城市之间都可以通过道路互相到达，形成一个树状结构。一位旅行者从 1 号城市出发，开始一段随机的旅程。
>
> **旅行规则**
>
> 1. 当旅行者到达一个城市时，他会随机选择一条尚未走过的道路，前往下一个城市。
> 2. 他永远不会走回头路（即不会立刻返回刚刚来的城市）。

3. 旅程在到达一个“终点城市”时结束。一个城市被称为“终点城市”，是指当旅行者到达该城市后，除了他来时的那条路外，没有其他新的、未走过的道路可供选择。这些城市在图论中通常被称为“叶子节点”。

任务 计算旅行者从 1 号城市出发，到任意一个终点城市结束旅程时，所走过的总路程的数学期望值。

输入形式

- 第一行包含一个整数 N（2 ≤ N ≤ 100001），表示城市的数量。
- 接下来 N-1 行，每行包含两个整数 u 和 v（1 ≤ u， v ≤ N），表示城市 u 和城市 v 之间有一条道路相连。

输出形式

- 输出一个浮点数，表示旅行总路程的期望值。结果要求保留足够的小数位数，误差在 10⁻⁶ 以内。

样例输入

代码块

```
4
1 2
1 3
1 4
```

样例输出

代码块

```
1.0000000
```

样例解释 从城市 1 出发，有三条路可以选，分别通向 2， 3， 4。

- 走到 2 的概率是 1/3，路程是 1。城市 2 是终点。
- 走到 3 的概率是 1/3，路程是 1。城市 3 是终点。
- 走到 4 的概率是 1/3，路程是 1。城市 4 是终点。 期望路程 E = （1 * 1/3） + （1 * 1/3） + （1 * 1/3） = 1.0。

二、问题分析与边界条件

问题核心

本题的核心要求是计算在一个树形结构中，从根节点（1 号城市）出发，进行随机遍历，到达任意一个叶子节点时，所经过路径长度的数学期望。

思路拆解

根据数学期望的定义 `E(X) = Σ [X_i * P(X_i)]`，我们需要解决以下几个子问题：

1. **识别所有终点**：找出图中所有的叶子节点。
2. **计算路径长度**：对于每一个叶子节点，计算从起点（1 号城市）到它的路径长度（即经过的道路数量）。
3. **计算路径概率**：对于每一个叶子节点，计算从起点出发，通过随机选择路径最终到达它的概率。
4. **求和计算期望**：将每个叶子节点的（路径长度 × 到达概率）累加起来，得到最终的期望值。

边界条件

1. **图的结构**：题目保证 N 个城市和 N-1 条路构成一个连通图，这说明该图一定是一棵树，不存在环路。
2. **起点**：旅程固定从 1 号城市开始。
3. **终点**：终点是度为 1 的节点（叶子节点）。但有一个特殊情况：如果 1 号城市本身就是叶子节点（例如 N=2 的情况），它不能是终点，因为旅程必须开启。代码中通过 `nowcity != STARTCITY` 条件来处理了这种情况。
4. **数据规模**：城市数量 N 最大可达 100,001，这意味着 `O(N^2)` 的算法可能会超时，需要设计更高效的算法。

三、算法设计

核心思想

本题是典型的图（树）的遍历问题，结合了概率计算。**深度优先搜索 (DFS)** 是解决此类问题的理想方法。我们可以从起点（1 号城市）开始进行 DFS，同时在递归过程中维护两个关键信息：

1. `length`：从起点到当前节点的路径长度。
2. `rate`：从起点到达当前节点的概率。

当 DFS 遍历到一个叶子节点时，我们就得到了一条完整的旅行路径，此时记录下该路径的 `length` 和 `rate`。遍历完所有可能的路径后，将所有记录的 `length * rate` 求和，即可得到期望值。

数据结构

- **图的表示**：代码使用了一个结构体 `Arabmap` 内的 `pair<int, int> routine[MAXSIZE]` 数组来存储 N-1 条边。这是一种**边列表**的表示方法。
- **遍历辅助**：
  - `bool visited[MAXSIZE]`：一个布尔数组，用于在 DFS 过程中标记已访问过的城市，以避免在树上往回走。
  - `vector<int> neighbors`：在 DFS 函数中临时创建的向量，用于存放当前城市的邻居节点。

- **结果存储**：
  - `int leaves[MAXSIZE]`：用于存储到达每个叶子节点时的路径长度。
  - `double rates[MAXSIZE]`：用于存储到达每个叶子节点的概率。

算法流程

1. **初始化**：读取城市数量 N 和 N-1 条道路信息，存储到边列表中。
2. **启动 DFS**：从 `STARTCITY`（1 号城市）调用 `DFS` 函数，初始路径长度为 -1（进入函数后会变为 0），初始概率为 1.0。
3. **DFS 递归过程** `DFS(nowcity, length, rate)`： a。 将当前城市 `nowcity` 标记为已访问，并将路径长度 `length` 加 1。 b. 查找并获取 `nowcity` 的所有邻居节点。 c. **判断是否为叶子节点**：如果 `nowcity` 的邻居数量为 1，并且它不是起点，那么它就是一个终点（叶子节点）。将当前的 `length` 和 `rate` 分别存入 `leaves` 和 `rates` 数组中，然后结束当前路径的递归。 d. **处理非叶子节点**：如果当前节点不是叶子节点，计算其可选择的子节点数量 `childnum`。 - 如果 `nowcity` 是起点，`childnum` 等于其邻居总数。 - 如果 `nowcity` 是中间节点，`childnum` 等于其邻居总数减 1（因为不能走回头路）。 e. 遍历所有邻居节点，如果邻居未被访问过，则对其进行递归调用 `DFS`，并传入更新后的参数：路径长度为 `length`，概率为 `rate / childnum`。
4. **计算期望值**：DFS 结束后，遍历 `leaves` 和 `rates` 数组，累加 `leaves[i] * rates[i]`，得到最终的期望值 `E`。
5. **输出结果**：按格式要求打印期望值 `E`。

四、核心代码讲解

算法的核心在于 `DFS` 函数，它通过递归遍历了所有可能的路径，并正确地计算了每条路径的长度和概率。

代码块

```
void DFS(int nowcity, int length, double rate) {
    // 标记当前城市已访问，并将路径长度加1
    visited[nowcity] = 1;
    length++;

    // 查找当前城市的所有邻居
    vector<int> neighbors;
    FindNearCities(neighbors, nowcity);
    int size = (int)neighbors.size();

    // 基本情况：判断是否为叶子节点（终点）
    // 条件：邻居只有1个（来时的路），且不是起点
    if (size == 1 && nowcity != STARTCITY) {
        leaves[p] = length; // 记录路径长度
```

```
            rates[p] = rate;    // 记录到达概率
            p++;
            return; // 结束当前路径的探索
        }

        // 递归情况：当前为非叶子节点
        int childnum;
        for (int i = 0; i < size; i++) {
            // 计算分支数量，用于均分概率
            if (nowcity == STARTCITY) {
                childnum = size; // 起点有 size 个分支
            } else {
                childnum = size - 1; // 中间节点有 size-1 个分支（除去父节点）
            }

            // 对每个未访问过的邻居（子节点）进行递归
            if (!visited[neighbors[i]]) {
                DFS(neighbors[i], length, rate / childnum);
            }
        }
    }
```

逻辑分析：

- `visited[nowcity]=1; length++;`： 这是递归的入口操作。每深入一层，路径长度加 1。
- `if (size == 1 && nowcity != STARTCITY)`： 这是递归的**终止条件**。当旅行者到达一个只有一个邻居（即来时的城市）且不是起点的城市时，旅程结束。此时，我们将这条完整路径的长度和最终概率记录下来。
- `childnum = ...`： 这部分是概率计算的关键。在 `nowcity`，旅行者会等概率地选择一个子节点前进。因此，到达 `nowcity` 的概率 `rate` 需要被 `childnum` 均分，传递给下一层的递归。
- `if (!visited[neighbors[i]])`： 这个判断确保了 DFS 只会向"深处"探索（访问子节点），而不会返回父节点，这与题目的"不走回头路"规则一致。
- `DFS(neighbors[i], length, rate / childnum)`： 这是**递归步骤**。它将问题分解为规模更小的子问题：从下一个城市出发，计算后续路径的期望。

五、复杂度分析

时间复杂度： `O(N^2)`

- `DFS` 函数本身会访问每个城市一次，这是 `O(N)`。
- 然而，在 `DFS` 的每一次调用中，都会调用 `FindNearCities` 函数。`FindNearCities` 的实现是通过遍历整个 `routine` 边列表（包含 N-1 条边）来查找邻居。

- 因此，对于每个城市的访问，都需要 `O(N)` 的时间来查找其邻居。
- 总的时间复杂度为 `N` （次 DFS 调用） * `O(N)` （FindNearCities 的开销） = `O(N^2)` 。
- **注意**：对于 N 高达 100,001 的情况， `O(N^2)` 的复杂度在严格的测试数据下会超时。更优化的实现应该使用邻接表来存储图，使得查找邻居的平均时间复杂度接近 `O(1)` ，从而将总时间复杂度优化到 `O(N)` 。

空间复杂度： `O(N)`

- `routine` 数组存储了 N-1 条边，空间为 `O(N)` 。
- `visited` ， `leaves` ， `rates` 三个数组的大小都与城市数量 N 成正比，空间为 `O(N)` 。
- DFS 递归调用的最大深度在最坏情况下（链状图）为 N，因此系统栈空间也需要 `O(N)` 。
- 综合来看，总的空间复杂度为 `O(N)` 。

六、总结与反思

方法总结

本题通过一个经典的“随机游走求期望”问题，考察了对图的深度优先搜索（DFS）算法的掌握，以及在遍历过程中处理概率问题的能力。核心解法是将期望值的计算分解到每条从根到叶的路径上，利用 DFS 遍历所有此类路径，并在递归过程中动态维护和传递路径长度与概率，最终汇总结果。

优化与拓展

- **性能优化**： 如复杂度分析中所述，当前代码最主要的性能瓶颈在于 `FindNearCities` 函数。通过将图的存储结构从**边列表**改为**邻接表**（例如 `vector<int> adj[MAXSIZE]` ），可以极大地提升查找邻居的效率。优化后，DFS 遍历每个节点和每条边恰好一次，总时间复杂度将降至 `O(N)` ，能够轻松通过最大规模的数据。
- **问题拓展**：
  - 如果图中有环，应该如何处理？（需要更复杂的 DFS 状态标记，避免无限循环）。
  - 如果每条道路的长度（权重）不为 1，该如何修改？（在 DFS 中累加真实长度而非步数）。
  - 如果旅行者可以走回头路，但每次选择的概率不同，期望值又该如何计算？（可能需要动态规划或更复杂的概率模型）。

学习收获

1. **深刻理解 DFS**: 通过本题，我再次体会到 DFS 在处理树/图路径问题上的强大能力。它不仅能用于查找，还能在遍历过程中携带和计算状态（如本题的长度和概率）。
2. **数据结构的重要性**： 本题的代码实现明确地展示了数据结构选择对算法性能的决定性影响。一个低效的图表示法（边列表）导致了平方级别的时间复杂度，而改用邻接表即可达到线性时间。这提醒我在解决问题时，必须首先选择最合适的数据结构。

3. **概率与算法的结合**： 学会了如何将数学期望的计算融入到算法流程中，理解了在树的分支处概率是如何被均分的，这为解决类似的概率+图论问题打下了基础。

猫与餐厅的故事 （ID: H5-06）

一、问题描述

【问题描述】

公司今天发薪，阿迪想与朋友们去餐厅庆祝一下。

他住在一个非常神奇的公园里，这个公园是一个根在顶点 1，且由 n 个顶点组成的有根树，顶点 1 也就是他的住所。然而不幸的是，公园也有许多的猫，阿迪已经找出了所有包含猫的顶点。

公园的叶子顶点都有餐厅，阿迪想选择一家他可以去的餐厅，但很不幸，他非常害怕猫，因而如果从餐厅去往他家的路径上有连续包含猫的数量超过 m 时，他将不能去往这家餐厅。

你的任务是帮助他确认他能去的餐厅的数量。

【输入形式】

输入的第一行包含两个整数 n 和 m （2≤ n ≤ 10^5， 1≤ m ≤ n），分别表示树的顶点数以及对于阿迪来说可以忍受的最大的包含猫的连续顶点数。

第二行包含 n 个整数 a1、a2、...、an，这里的每个 ai 或者为 0（顶点 i 无猫），或者为 1（顶点 i 有猫）。

接下来的 n - 1 行包含用形式“xi yi”（1≤ xi， yi ≤ n，xi ≠ yi）表示的树的边，表示顶点 xi 和顶点 yi 之间有边相连。

【输出形式】

输出为一个整数，表示从阿迪家去往叶子顶点的路径上至多包含 m 只猫的叶子顶点的数量。

【样例输入 1】

代码块

```
4 1
1 1 0 0
1 2
1 3
1 4
```

【样例输出 1】

代码块

```
2
```

【样例输入 2】

代码块

```
7 1
1 0 1 1 0 0 0
1 2
1 3
2 4
2 5
3 6
3 7
```

【样例输出 2】

代码块

```
2
```

二、问题分析与边界条件

问题核心： 本题的核心任务是在一个给定的树结构中，找出从根节点（顶点 1）出发，到达任意一个叶子节点（餐厅）的所有“安全”路径的数量。一条路径被定义为“安全”，当且仅当该路径上连续出现带猫节点的数量不超过一个给定的阈值 `m`。

思路拆解： 要解决这个问题，我们需要：

1. **表示公园的结构**：公园是一个树形结构，可以使用图的数据结构（如邻接表）来表示。
2. **遍历所有路径**：需要一种方法从根节点（阿迪的家）出发，系统性地访问到所有的叶子节点（餐厅）。
3. **路径状态跟踪**：在遍历过程中，必须实时记录当前路径上末尾连续有多少个节点有猫。
4. **路径合法性判断**：在任何时刻，如果连续带猫节点的数量超过了 `m`，则当前路径及其所有延伸路径都视为不安全，应停止继续探索。
5. **统计结果**：当成功到达一个叶子节点且路径全程安全时，将结果计数器加一。

边界条件： 根据题目描述和代码实现，需要考虑以下边界情况：

- **根节点有猫**：从根节点开始，连续猫的数量就从 1 开始计数。
- **路径剪枝**：一旦某条路径上的连续猫数量超过 `m`，必须立即停止对该路径的进一步探索（剪枝），否则可能会错误地把不安全路径的子路径计入结果。
- **叶子节点的定义**：一个节点是叶子节点，当且仅当它的度为 1。但根节点（顶点 1）除外，即使它的度为 1（例如 n=2 的情况），它也不是餐厅。在遍历中，如果一个节点除了其父节点外没有其他邻居，那么它就是一个叶子节点。

三、算法设计

核心思想： 本问题是在树上进行带条件的路径搜索，深度优先搜索（Depth First Search， DFS）是解决此类问题的经典且自然的方法。DFS 可以模拟“从家（根节点）出发，沿着一条路走到黑，再回溯换另一条路”的过程，非常契合本题的求解逻辑。

数据结构：

1. **邻接表 （`vector<vector<int>> adj`）**：用于存储树的结构。 `adj[i]` 存储了与顶点 `i` 相连的所有顶点。这种方式便于遍历一个节点的所有邻居，空间效率也高。
2. **猫位置数组 （`vector<int> hasCat`）**：一个大小为 `n+1` 的数组， `hasCat[i]` 的值为 1 表示顶点 `i` 有猫，为 0 表示没有。用于 O（1）时间复杂度查询某个顶点是否有猫。

算法流程：

1. **初始化**：读入 `n` 和 `m`，初始化 `hasCat` 数组和邻接表 `adj`。
2. **建树**：通过循环读入 `n-1` 条边，构建邻接表来表示公园的树形结构。
3. **启动 DFS**：从根节点 1 开始，调用核心的 DFS 函数 `solve_dfs(1, 0, 0)`。初始调用时，当前节点为 1，其父节点设为一个不存在的节点 0，路径上初始的连续猫数量为 0。
4. **DFS 递归过程 `solve_dfs(u, parent, consecutiveCats)`**： a。 **更新连续猫数量**：进入函数后，首先根据当前节点 `u` 是否有猫来更新连续猫的数量 `currentConsecutiveCats`。如果 `hasCat[u]` 为 1，则 `currentConsecutiveCats = consecutiveCats + 1`；否则，连续性被打破， `currentConsecutiveCats = 0`。 b. **合法性检查（剪枝）**：判断 `currentConsecutiveCats` 是否大于 `m`。如果大于 `m`，说明当前路径已不安全，立即返回，不再继续向下探索。 c. **探索子节点**：遍历当前节点 `u` 的所有邻居 `v`。如果 `v` 不是 `parent` （防止走回头路），则递归调用 `solve_dfs(v, u, currentConsecutiveCats)`。 d. **判断并统计叶子节点**：在遍历完所有子节点后，判断当前节点 `u` 是否为叶子节点。一个简单的判断方法是：在递归过程中，如果一个节点没有任何可以向下探索的子节点，它就是叶子节点。若该叶子节点被成功访问（即路径安全），则全局计数器 `reachableRestaurants` 加 1。
5. **输出结果**：DFS 结束后，输出 `reachableRestaurants` 的值。

四、核心代码讲解

算法的核心在于深度优先搜索的递归函数 `solve_dfs`。

代码块

```
function<void(int, int, int)> solve_dfs =
    [&](int u, int parent, int consecutiveCats) {

    // 步骤 a: 更新当前路径的连续猫咪数
    int currentConsecutiveCats = hasCat[u] ? consecutiveCats + 1 : 0;

    // 步骤 b: 检查是否违反猫咪数量限制 (剪枝)
```

```
        if (currentConsecutiveCats > m) {
            return;
        }

        // 步骤 c & d: 探索子节点并判断是否为叶子
        bool is_leaf = true; // 假设当前节点是叶子
        for (int v : adj[u]) {
            if (v != parent) {
                is_leaf = false; // 只要有一个子节点，就不是叶子
                solve_dfs(v, u, currentConsecutiveCats);
            }
        }

        // 如果遍历完邻居后，is_leaf 仍为 true，说明它是一个安全的叶子节点
        if (is_leaf) {
            reachableRestaurants++;
        }
    };
```

代码分析:

1. `int currentConsecutiveCats = hasCat[u] ? consecutiveCats + 1 : 0;`： 这一行是状态更新的核心。它接收来自父节点的连续猫数量 `consecutiveCats`，并根据当前节点 `u` 的情况进行更新。三元运算符简洁地实现了逻辑：如果 `u` 有猫，则数量加一；如果没猫，则清零。
2. `if (currentConsecutiveCats > m) { return; }`： 这是算法的关键剪枝步骤。一旦路径不满足条件，就立刻终止对该分支的探索，极大地提高了效率，并保证了结果的正确性。
3. `bool is_leaf = true; ... for ... is_leaf = false;`： 这里巧妙地利用一个布尔变量来判断叶子节点。先假设当前节点是叶子，然后遍历它的邻居。只要能找到一个非父节点的邻居（即一个子节点），就将 `is_leaf` 置为 `false`。
4. `if (is_leaf) { reachableRestaurants++; }`： 在对所有子节点递归调用结束后，检查 `is_leaf` 标志。如果它仍然是 `true`，意味着当前节点 `u` 是一个叶子节点，并且由于程序能执行到这里，从根到 `u` 的路径一定是安全的。因此，将结果加一。

五、复杂度分析

时间复杂度： O（N）

- **推导过程**： 算法的核心是深度优先搜索（DFS）。在 DFS 过程中，每个顶点和每条边都只会被访问一次。图的初始化（读入数据、建邻接表）需要 O（N） 的时间。因此，总的时间复杂度与图的顶点数和边数成线性关系，即 O（N + （N-1））， 简化后为 O（N）。

空间复杂度： O（N）

- **推导过程**:

a. **邻接表 `adj`**： 存储了 `N-1` 条边，每条边存两次，所以总空间为 O（N）。

b. **`hasCat` 数组**： 需要 O（N） 的空间。

c. **递归调用栈**： DFS 的递归深度取决于树的高度。在最坏的情况下（树退化成一条链），树的高度为 N，因此递归栈需要 O（N） 的空间。

- 综合以上几点，算法所需的总空间与输入规模 N 成正比，因此空间复杂度为 O（N）。

六、总结与反思

方法总结： 本题是利用深度优先搜索（DFS）解决树上路径问题的典型应用。其核心在于通过递归函数的参数来传递和维护路径状态（本题中为连续猫的数量），并利用剪枝策略来提前终止不合法的搜索分支。这种"携带状态的遍历"是树/图算法中一种非常重要的思想。

优化与拓展：

- **BFS 实现**： 本题同样可以使用广度优先搜索（BFS）解决。BFS 的队列中需要存储一个状态对 `(currentNode, consecutiveCats)`。当从队列中取出一个节点时，对其所有子节点进行判断和状态更新，然后将合法的子节点状态对加入队列。当到达一个叶子节点时，进行计数。两种方法在思路上是等价的。
- **问题变体**： 如果问题改为"求所有安全路径中，路径长度最长/最短的是多少？"，我们可以在 DFS 参数中增加一个 `currentPathLength`，并在到达叶子节点时更新全局的最值。

学习收获：

1. **巩固了 DFS 在树遍历中的应用**：深刻理解了如何通过 DFS 的递归结构来探索树中的所有根到叶的路径。
2. **学习了状态传递与剪枝**： 掌握了通过函数参数在递归中传递状态信息，并根据该状态进行剪枝以优化搜索过程的关键技巧。
3. **理解了叶子节点的判断**： 学会在 DFS 遍历中，通过检查一个节点是否存在子节点来动态地判断其是否为叶子节点。

最昂贵的旅行 （ID: H5-07）

一、问题描述

> **问题描述**
>
> 这个国家有 n 个城市，编号从 0 ~ n-1，城市网络中没有任何环路，但可以从任意一个城市出发沿公路直接或间接到达其他城市。
>
> 有人住在编号为 0 的城市里，他希望去其他的一个城市旅行，但他不想付出更多的成本，所以他想知道去哪个城市的成本是最高的。
>
> **输入形式**

输入的第一行为一个整数 n （3≤ n ≤100），接下来的 n-1 行每行包括 3 个整数 u、v、c （0 ≤ u， v ≤ n-1，1≤ c ≤10^4），意为在城市 u 和 v 之间有公路直接相连，且旅行需要花费的成本为 c 。

输出形式

输出为一个整数，表示从城市 0 出发去到其他的某个城市，需要付出的最大成本。

样例输入 1

代码块

```
4
0 1 4
0 2 2
2 3 3
```

样例输出 1

代码块

```
5
```

样例输入 2

代码块

```
6
1 2 3
0 2 100
1 4 2
0 3 7
3 5 10
```

样例输出 2

代码块

```
105
```

二、问题分析与边界条件

问题核心： 题目的核心要求是计算一个带权图中，从指定的起点（0 号城市）出发，到所有其他城市的最长路径。由于题目明确指出“没有任何环路”且“可以从任意一个城市出发...到达其他城市”，这说

明给定的城市网络是一个**树形结构**。因此，问题转化为求解树中从根节点（0 号节点）到所有其他节点的最长路径长度。

思路拆解：

1. **图的表示**： 首先，需要一种方式来存储城市和道路的信息。一个包含 `n` 个节点和 `n-1` 条边的无向带权图是合适的数据结构。
2. **路径搜索**： 需要从 0 号城市出发，遍历所有能够到达的城市，即遍历整棵树。
3. **成本计算**： 在遍历的过程中，需要累加路径上的成本。
4. **最值更新**： 需要一个变量来记录并持续更新在遍历过程中遇到的最大路径成本。

边界条件： 根据题目描述和代码实现，需要注意以下边界条件：

- 城市数量 `n` 的范围是 `3 ⩽ n ⩽ 100`。
- 这是一个连通图，且有 `n` 个节点和 `n-1` 条边，保证了它是一个树。
- 起点固定为 0 号城市。
- 边的权重 `c` 是正整数。

三、算法设计

核心思想： 本题的本质是在一个树形结构中，寻找从根节点（0 号城市）出发的最长路径。深度优先搜索（DFS）是一种非常适合遍历树或图的算法，它可以系统地探索从起点出发的每一条路径直到终点。

我们将从 0 号城市开始进行深度优先搜索。对于每个访问到的城市，我们记录从起点到该城市的累计成本。在访问每个城市时，都用当前累计的成本去更新一个全局的最大成本记录。当 DFS 完成对所有可达城市的遍历后，这个全局记录的值就是最终的答案。

数据结构：

- **邻接表 （`vector<vector<pair<int, int>>> adj`）**：我们选择使用邻接表来存储这个树形结构。`adj` 是一个向量的向量，其中 `adj[i]` 存储了一个 `pair` 列表。每个 `pair<int, int>{v, c}` 表示城市 `i` 和城市 `v` 之间有一条成本为 `c` 的道路。相比邻接矩阵，邻接表更节省空间，尤其适用于边数远少于节点数平方的稀疏图（树就是一种典型的稀疏图）。

算法流程：

1. **初始化**： 读取城市数量 `n`，并初始化一个大小为 `n` 的邻接表 `adj` 和一个全局变量 `maxCost = 0` 用于记录最大成本。
2. **建图**： 循环读取 `n-1` 条道路信息（u， v， c）。由于道路是双向的，需要在 `adj[u]` 中添加 `{v, c}`，同时在 `adj[v]` 中添加 `{u, c}`，从而构建一个无向图的表示。
3. **开始搜索**： 从 0 号城市开始调用深度优先搜索函数 `dfs(0, -1, 0)`。
    - 第一个参数 `0` 是当前节点（起点）。

- 第二个参数 `-1` 是父节点。使用一个无效的节点编号（如 -1）来表示根节点没有父节点，这对于防止在遍历时走回头路至关重要。
- 第三个参数 `0` 是从起点到当前节点的累计成本，起点自身成本为 0。

4. **DFS 递归过程** `dfs(u, parent, currentCost)`： a。 **更新最值**： 进入函数后，首先用当前的累计成本 `currentCost` 更新全局最大成本： `maxCost = max(maxCost, currentCost)` 。 b. **遍历邻居**： 遍历当前城市 `u` 的所有邻居 `v` 。 c. **避免回溯**： 对于每个邻居 `v` ，检查它是否是来时的父节点 （ `v != parent` ）。如果不是，说明这是一条通往更深处的新路径。 d. **继续深入**： 对该邻居 `v` 进行递归调用 `dfs(v, u, currentCost + cost)` ，其中 `u` 成为新的父节点，并且路径成本更新为当前成本加上连接 `u` 和 `v` 的道路成本。
5. **输出结果**： 当所有递归调用都结束后， `maxCost` 中存储的即为从 0 号城市出发能到达的最昂贵的旅行成本，输出该值。

四、核心代码讲解

本算法的核心在于深度优先搜索的递归实现，以下是 `dfs` 函数的代码片段及其讲解。

代码块

```
// 深度优先搜索函数
void dfs(int u, int parent, int currentCost) {
    // 用当前到达 u 的成本更新全局最大成本
    maxCost = max(maxCost, currentCost);

    // 遍历 u 的所有邻居
    for (const auto& edge : adj[u]) {
        int v = edge.first;
        int cost = edge.second;

        // 如果邻居 v 不是我们来的父节点，就继续向下探索
        if (v != parent) {
            dfs(v, u, currentCost + cost);
        }
    }
}
```

逐行分析：

1. `void dfs(int u, int parent, int currentCost)` ：
   - `int u` ： 当前正在访问的城市（节点）的编号。
   - `int parent` ： 我们是从哪个城市（父节点）来到 `u` 的。这个参数是防止搜索在两个节点间来回往返的关键。

- `int currentCost`： 从起点 0 到达城市 `u` 所累积的总成本。

2. `maxCost = max(maxCost, currentCost);`：
   - 每当访问一个新城市 `u`，就意味着我们找到了一条从起点 0 到 `u` 的路径。
   - 我们将这条路径的成本 `currentCost` 与已记录的全局最大成本 `maxCost` 进行比较，并保留较大者。这样可以确保在遍历结束后，`maxCost` 存储的是所有路径中的最大成本。
3. `for (const auto& edge : adj[u])`：
   - 这是一个基于范围的 `for` 循环，用于遍历邻接表 `adj[u]` 中存储的所有与城市 `u` 直接相连的边。`edge` 是一个 `pair`，包含了邻居城市的编号和边的成本。
4. `if (v != parent)`：
   - 这是 DFS 遍历树（或用邻接表表示的无向图）时的核心逻辑之一。
   - 因为我们在构建邻接表时添加了双向边（`u` -> `v` 和 `v` -> `u`），如果不加判断，从 `u` 访问到 `v` 之后，会立即从 `v` 的邻居中找到 `u` 并访问回来，导致无限递归。
   - 通过检查邻居 `v` 是否是来时的 `parent` 节点，我们保证了搜索始终是向着"更深"或"未探索"的方向进行。
5. `dfs(v, u, currentCost + cost);`：
   - 如果邻居 `v` 是一个有效的、新的探索方向，就对其发起递归调用。
   - `v` 成为新的当前节点。
   - `u` 成为 `v` 的父节点，传递给下一层递归。
   - `currentCost + cost` 是新的累计成本，即到达 `u` 的成本加上从 `u` 到 `v` 的成本。

五、复杂度分析

时间复杂度： O（N）

- 该算法的本质是深度优先搜索。在一个由 N 个节点和 N-1 条边构成的树中，DFS 会访问每个节点（城市）一次，并遍历每条边（道路）两次（因为是无向图表示，`u` -> `v` 和 `v` -> `u` 各一次）。
- 因此，总的操作次数与节点数 N 和边数 M 成正比，即 O（N+M）。对于树来说，M = N-1，所以时间复杂度为 O（N + N - 1），简化为 **O（N）**。

空间复杂度： O（N）

- 空间消耗主要来自两个方面：
  a. **邻接表存储**： 邻接表需要存储 N-1 条边，每条边存储两次，所以总共是 `2*(N-1)` 个条目，空间复杂度为 O（N）。
  b. **递归调用栈**： DFS 是通过递归实现的，调用栈的深度取决于树的高度。在最坏的情况下，树可能退化成一条链，此时树的高度为 N-1，递归深度会达到 O（N）。

- 综合来看，算法所需的额外空间与输入规模 N 成正比，因此空间复杂度为 **O（N）**。

六、总结与反思

方法总结： 本题是一个典型的图论问题在树结构上的应用。通过分析题目"n 个城市、n-1 条路、无环路"的特性，我们能够迅速识别出其数据结构为"树"。一旦确定是树，许多图的遍历算法都可以被简化和应用。本题的核心解法是采用深度优先搜索（DFS）从根节点出发，遍历所有节点，并在过程中记录最大路径和。这展示了将实际问题抽象成经典数据结构模型，并利用基础算法高效求解的通用思路。

优化与拓展：

- **BFS 解法**： 虽然 DFS 非常直观，但本题同样可以使用广度优先搜索（BFS）解决。BFS 需要一个队列，并在队列中存储 `{节点，当前成本}` 的 `pair`。每次从队列中取出一个元素，对其所有未访问过的邻居，计算新成本并加入队列。同样，需要一个全局变量来更新最大成本。
- **寻找树的直径**： 本题求解的是从固定根节点出发的最长路径。一个相关但更复杂的问题是求解"树的直径"，即树中任意两个节点之间最长路径的长度。这通常可以通过两次 DFS 或 BFS 来解决：首先从任意节点 `s` 出发找到距离它最远的节点 `u`，然后再从节点 `u` 出发找到距离 `u` 最远的节点 `v`，`u` 和 `v` 之间的路径就是树的直径。

学习收获：

1. **问题抽象能力**： 学习到了如何从问题描述中提炼关键信息，将一个应用问题（城市旅行）转化为一个标准的算法模型（在树上求最长路径）。
2. **DFS 在树上的应用**： 深刻理解了深度优先搜索在树遍历中的工作原理，特别是通过传递 `parent` 节点参数来避免在无向图中走回头路的关键技巧。
3. **邻接表的应用**： 掌握了使用 `vector<vector<pair<int, int>>>` 作为邻接表来表示带权图的方法，这是图论问题中最常用和高效的数据结构之一。

良心树 （ID: H5-08）

一、问题描述

> **【问题描述】**
>
> 给定一颗有根树，顶点编号为 1～n，树是一个无环的连通图，有根树有一个特定的顶点，称为根。
>
> 顶点 i 的祖先是从根到顶点 i 的路径上除顶点 i 以外的所有顶点，顶点 i 的父母是 i 的祖先中最接近 i 的顶点，每个顶点都是它父母的孩子。在给定的树中，顶点 i 的父母是顶点 pi，对于根，pi 为 -1。
>
> 在树中，其中一些顶点不尊重其他一些顶点，实际上，如果 ci =1，表示顶点 i 不尊重它的所有祖先，而如果 ci =0，则表示它尊重它所有的祖先。
>
> 你需要一个一个地删除一些顶点，在每一步中，选择一个非根顶点，它不尊重它的父母并且它的所有孩子顶点也不尊重它。如果有几个这样的顶点，你需要选择具有最小编号的顶点。当你删除了这

样的一个顶点 v， 则 v 的所有子顶点与 v 的父母顶点相连。

直到树中无满足删除标准的顶点，则上述过程停止。按顺序输出你删除的所有顶点，注意这个顺序的唯一的。

【输入形式】

输入的第一行为一个整数 n （1≤ n ≤10^5），表示树的顶点数。

接下来的 n 行描述了整颗树：第 i 行包含两个整数 pi 和 ci （1≤ pi ≤ n， 0≤ ci ≤1），这里 pi 是顶点 i 的父母，若 ci=0，表示顶点 i 尊重它的父母，ci=1，表示顶点 i 不尊重它的父母，pi=-1 时，表示顶点 i 是树的根，同时 ci=0。

【输出形式】

如果树中至少有一个顶点被删除，则按照顺序输出顶点编号，否则输入 -1。

【样例输入 1】

代码块

```
5
3 1
1 1
-1 0
2 1
3 0
```

【样例输出 1】

代码块

```
1 2 4
```

【样例输入 2】

代码块

```
5
-1 0
1 1
1 1
2 0
3 0
```

【样例输出 2】

```
1   -1
```

二、问题分析与边界条件

问题核心： 本题要求我们根据一套特定的规则，在一棵有根树上迭代地删除节点，并按删除顺序输出这些节点的编号。删除过程一直持续到没有节点满足删除条件为止。

思路拆解： 初看此题，会认为是一个动态模拟过程：找到所有满足条件的节点，选择编号最小的删除，更新树的结构，然后重复此过程。然而，这种模拟非常复杂，因为删除一个节点会改变其父节点的子节点列表。我们需要分析删除操作是否会影响其他节点的可删除状态。

一个节点 `v` 可被删除的条件是：

1. `v` 不是根节点。
2. `v` 不尊重其父节点，即 `c[v] == 1`。
3. `v` 的所有孩子 `u` 都不尊重它，即对所有孩子 `u` 都有 `c[u] == 1`。

关键洞察： 让我们分析删除节点 `v` 对其父节点 `p` 的可删除状态的影响。 `p` 是否可删除，取决于它是否有"尊重"它的孩子（即 `c` 值为 0 的孩子）。

- 在删除 `v` 之前， `v` 是 `p` 的孩子。由于 `v` 要被删除， `c[v]` 必须为 1，所以 `v` 是一个"不尊重"的孩子。
- `v` 的所有孩子 `u` 也都必须满足 `c[u] == 1`，它们也都是"不尊重"的孩子。
- 当 `v` 被删除后， `p` 失去了孩子 `v`，同时获得了 `v` 的所有孩子 `u` 作为自己的新孩子。
- 在这个过程中， `p` 失去了一个不尊重的孩子，又获得了一批不尊重的孩子。因此， `p` 所拥有的"尊重"它的孩子的数量始终没有改变。
- 这个惊人的结论意味着：**删除一个节点不会改变任何其他节点的可删除状态。**一个节点是否可删除，完全由树的初始结构和所有节点的 `c` 值决定，它是一个静态属性。

因此，问题从一个复杂的动态模拟简化为了一个简单的静态检查。

边界条件：

- `1 ≤ n ≤ 10^5`：节点数量较大，算法必须高效，时间复杂度应为 O（N） 或 O（N log N）。
- 根节点：根节点的父节点为 -1，它永远不能被删除。
- 叶子节点：没有孩子的节点。对于叶子节点，条件"所有孩子都不尊重它"天然满足。因此，一个非根的叶子节点 `i` 只要满足 `c[i] == 1` 即可被删除。
- 无满足条件的节点：如果遍历完所有节点都找不到可删除的，应输出 -1。

三、算法设计

基于上述分析，我们无需模拟删除过程，只需在初始状态下找出所有满足删除条件的节点即可。

核心思想： 将动态的删除问题转化为静态的条件检查。一次性找出所有符合删除标准的节点，并按编号从小到大输出。

数据结构：

1. `vector<int> parents`： 大小为 `n+1`， `parents[i]` 存储节点 `i` 的父节点编号。
2. `vector<int> respects`： 大小为 `n+1`， `respects[i]` 存储节点 `i` 的 `c` 值。
3. `vector<vector<int>> children`： 大小为 `n+1` 的邻接表， `children[i]` 存储节点 `i` 的所有子节点的编号。这用于快速遍历一个节点的子节点。
4. `vector<int> respectful_children_count`： 大小为 `n+1`， `respectful_children_count[i]` 存储节点 `i` 的所有孩子中 `c` 值为 0（尊重）的孩子的数量。这是为了实现 O（1） 的条件检查。

算法流程：

1. **初始化**：读取 `n`，并初始化上述四个数据结构。
2. **建树与数据录入**：遍历输入，对于每个节点 `i`，读取其父节点 `p` 和 `c` 值。
   - `parents[i] = p`
   - `respects[i] = c`
   - 如果 `p != -1`，则将 `i` 添加到 `p` 的子节点列表 `children[p].push_back(i)`。
3. **预计算尊重子节点数**：创建一个大小为 `n+1` 的 `respectful_children_count` 数组并初始化为 0。遍历所有节点 `i` 从 1 到 `n`，再遍历 `i` 的每一个孩子 `child_node` （从 `children[i]` 中获取），如果 `respects[child_node] == 0`，则 `respectful_children_count[i]` 加一。
4. **筛选待删除节点**：创建一个空的 `vector<int> nodes_to_delete` 用于存放结果。从 1 到 `n` 遍历所有节点 `i`。对于每个节点 `i`，检查其是否满足所有删除条件：
   - `parents[i] != -1` （非根节点）
   - `respects[i] == 1` （不尊重父节点）
   - `respectful_children_count[i] == 0` （所有孩子都不尊重它） 如果全部满足，则将 `i` 添加到 `nodes_to_delete` 向量中。
5. **输出结果**：
   - 检查 `nodes_to_delete` 是否为空。如果为空，输出 -1。
   - 如果不为空，则依次输出 `nodes_to_delete` 中的所有元素。由于我们是按 `i` 从 1 到 `n` 的顺序进行检查和添加的，这个向量中的节点编号自然就是升序的，符合题目"选择具有最小编号的顶点"的要求。

四、核心代码讲解

代码片段 1：预计算"尊重"子节点的数量

```
代码块
1   // 核心步骤1: 计算每个节点拥有多少个“尊重”它的孩子 (c[child] == 0)
2   vector<int> respectful_children_count(n + 1, 0);
3   for (int i = 1; i <= n; ++i) {
4       // 遍历 i 的所有孩子，统计其中 c 值为 0 的个数
5       for (int child_node : children[i]) {
6           if (respects[child_node] == 0) {
7               respectful_children_count[i]++;
8           }
9       }
10  }
```

分析： 这是算法的关键预处理步骤。我们没有在每次判断时都去遍历子节点，而是提前为每个节点计算好它有多少个“尊重”它的孩子。

- `vector<int> respectful_children_count(n + 1, 0);`： 定义一个数组，用于存储每个节点的尊重子节点数，并全部初始化为 0。
- `for (int i = 1; i <= n; ++i)`： 遍历树中的每一个节点，作为潜在的父节点。
- `for (int child_node : children[i])`： 使用之前构建的邻接表 `children`，高效地遍历当前节点 `i` 的所有直接子节点。
- `if (respects[child_node] == 0)`： 检查该子节点的 `c` 值。如果为 0，说明这是一个“尊重”父节点的孩子。
- `respectful_children_count[i]++;`： 将父节点 `i` 的尊重子节点计数器加一。 通过这个双重循环，我们用 O（N） 的时间复杂度完成了对所有节点尊重子节点数的统计，为后续的 O（1） 判断奠定了基础。

代码片段 2：筛选并收集所有可删除的节点

```
代码块
1   // 核心步骤2: 找出所有在初始状态下就满足删除条件的节点
2   vector<int> nodes_to_delete;
3   for (int i = 1; i <= n; ++i) {
4       // 删除条件:
5       // 1. 非根节点 (parents[i] != -1)
6       // 2. 不尊重其父节点 (respects[i] == 1)
7       // 3. 所有孩子都不尊重它 (即，尊重它的孩子数量为 0)
8       if (parents[i] != -1 && respects[i] == 1 && respectful_children_count[i] =
9           nodes_to_delete.push_back(i);
10      }
11  }
```

分析： 这是算法的核心判断步骤。利用预计算的结果，我们可以快速筛选出所有符合条件的节点。

- `vector<int> nodes_to_delete;`： 创建一个动态数组，用于存储最终要输出的节点列表。
- `for (int i = 1; i <= n; ++i)`： 按节点编号从小到大遍历所有节点。这保证了最终输出的序列是按编号升序的，巧妙地满足了题目中"选择具有最小编号的顶点"的要求。
- `if (parents[i] != -1 && respects[i] == 1 && respectful_children_count[i] == 0)`： 这是一个逻辑与（AND）表达式，将三个删除条件合并在一个 `if` 语句中进行判断。
  - `parents[i] != -1`： 检查是否为根节点。
  - `respects[i] == 1`： 检查 `c` 值是否为 1。
  - `respectful_children_count[i] == 0`： 直接使用上一步预计算的结果，以 O（1）的复杂度判断是否所有孩子都不尊重它。
- `nodes_to_delete.push_back(i);`： 如果一个节点同时满足以上三个条件，就将其编号加入待删除列表。

五、复杂度分析

- **时间复杂度： O（N）**
  - **输入和建树**： 循环 `n` 次读取输入，构建 `parents` 、 `respects` 和 `children` 数组/邻接表。总共有 `n-1` 条边，所以填充邻接表的操作总共也是 O（N）。此阶段复杂度为 O（N）。
  - **预计算尊重子节点数**： 外层循环遍历 `n` 个节点，内层循环遍历每个节点的子节点。由于每个节点只会被其父节点访问一次，这等价于遍历了树的所有边，复杂度为 O（N）。
  - **筛选待删除节点**： 单层循环 `n` 次，每次循环内部的判断均为 O（1） 操作。此阶段复杂度为 O（N）。
  - **总时间复杂度**： O（N） + O（N） + O（N） = O（N）。算法非常高效。
- **空间复杂度： O（N）**
  - `parents` , `respects` , `respectful_children_count` 三个数组各需要 O（N） 的空间。
  - `children` 邻接表需要存储 `n-1` 条边，总空间消耗为 O（N）。
  - `nodes_to_delete` 向量在最坏情况下可能存储 `n-1` 个节点，需要 O（N） 的空间。
  - **总空间复杂度**： O（N） + O（N） + O（N） + O（N） + O（N） = O（N）。

六、总结与反思

方法总结： 本题的成功解决依赖于一个关键的洞察：看似复杂的动态模拟问题，其核心判定条件在操作中是**不变量**。通过深入分析删除操作对树结构和节点属性的影响，我们将问题转化为了一个简单的静态属性检查问题。这体现了在解决算法问题时，深入分析问题内在性质、寻找不变量的重要性，它往往能将复杂问题极大简化。

优化与拓展：

- 本题的解法已经是线性的，在时间和空间上都已达到最优。
- 可以思考一个拓展问题：如果删除一个节点 `v` （c[v]=1），会使其父节点 `p` 的一个“尊重”子节点 `u` （c[u]=0） 变为“不尊重”（c[u]=1），问题会如何变化？在这种情况下，删除操作会动态地改变其他节点的状态，原有的静态检查方法失效。届时，我们可能需要使用真正的数据结构（如优先队列或集合）来维护当前所有可删除的节点，每次取出最小的进行删除，并更新受影响节点的状态，再将其加入队列。

学习收获：

1. **分析不变量是简化问题的金钥匙**：面对模拟类问题时，不要急于动手写复杂的模拟逻辑。首先应该仔细分析操作的“副作用”，判断是否存在不变量。找到不变量，往往就能找到通往高效解法的捷径。
2. **预计算思想的应用**：通过 `respectful_children_count` 数组，我们将反复进行的查询操作（一个节点有多少尊重它的孩子）的时间复杂度从 O（度数） 降为了 O（1）。这种“以空间换时间”的预计算思想在算法设计中非常普遍和有效。
3. **图/树的邻接表表示法**：对于树或图的问题，使用邻接表（ `vector<vector<int>>` ）来存储其结构，是进行遍历和邻接点查询的标准且高效的做法。

[0-1 串] （ID: H5-09）

一、问题描述

> **【问题描述】**
>
> 对于一个包含 n 个整数元素的序列 a1、a2、...、an，每个元素的值或者是 0 或者是 1，选择两个下标 i 和 j（1≤i≤j≤n），对于所有的此范围内的元素 ak（i≤k≤j），执行操作 ak =1- ak。
>
> 选择合适的 i 和 j，执行上述操作一次之后，可以得到的新序列中包含 1 的个数最多是多少?
>
> **【输入形式】**
>
> 输入的第一行为一个整数 n（1≤ n ≤100），接来的一行为 n 个整数，每个整数或者是 0 或者是 1。
>
> **【输出形式】**
>
> 输出为一个整数，表示执行一次上述操作后可以获得的最大 1 的个数。
>
> **【样例输入 1】**
>
> 代码块
>
> ```
> 1   5
> 2   1 0 0 1 0
> ```

【样例输出 1】

```
代码块
1    4
```

【样例说明】

在第一个样例中，选择 i=2， j=5， 改变后的序列为[1 1 1 0 1]，包含 4 个 1，很显然无法改变为[1 1 1 1 1]。

二、问题分析与边界条件

问题核心： 本题的核心要求是，在一个 0-1 序列中，通过**翻转一个连续子段**（0 变 1，1 变 0），使得最终序列中 1 的数量达到最大。

思路拆解：

1. **最终结果的构成**： 最终 1 的总数 = `原始序列中1的个数` + `翻转操作带来的1的个数变化量`。
2. **分析变化量**： 当我们翻转一个子段时，每一个 0 会变成 1（使 1 的总数 +1），每一个 1 会变成 0（使 1 的总数 -1）。因此，变化量 = `子段中0的个数` - `子段中1的个数`。
3. **核心子问题**： 为了让最终 1 的总数最大，我们需要找到一个子段 [i， j]，使得 `(子段中0的个数 - 子段中1的个数)` 这个差值最大。

边界条件： 根据题目描述和代码实现，需要考虑以下边界情况：

- **序列长度**： `n` 的范围是 1 到 100。代码中也处理了 `n=0` 的极端情况。
- **全为 1 的序列**： 如果输入序列所有元素都是 1，如 `1 1 1 1`。翻转任何子段都会导致 1 的个数减少。最优策略是翻转长度为 1 的子段，最终结果为 `n-1`。
- **全为 0 的序列**： 如果输入序列所有元素都是 0，如 `0 0 0 0`。翻转整个序列可以得到最多的 1，最终结果为 `n`。

三、算法设计

核心思想： 本题的巧妙之处在于将问题转化为一个经典的算法模型：**最大子数组和 （Maximum Subarray Sum）**。

- **问题转化**： 我们要寻找的 `max(子段中0的个数 - 子段中1的个数)`，可以看作是寻找一个“收益”最大的子段。
- 定义“收益”：
  - 对于原始序列中的 `0`，翻转它会使 1 的数量增加 1，我们视其**收益为 `+1`**。
  - 对于原始序列中的 `1`，翻转它会使 1 的数量减少 1，我们视其**收益为 `-1`**。

- **新问题**： 基于上述定义，我们可以创建一个新的“收益序列”。原问题就等价于：**在这个新的收益序列中，找到一个连续子段，使其和最大。**

数据结构：

1. `std::vector<int> a(n)`： 用于存储用户输入的原始 0-1 序列。
2. `std::vector<int> gain(n)`： 用于存储根据原始序列转化而来的收益序列。 `a[i]` 为 0 时， `gain[i]` 为 1； `a[i]` 为 1 时， `gain[i]` 为 -1。

算法流程：

1. **初始化**： 读入序列长度 `n` ，创建一个大小为 `n` 的 `vector a` 。定义一个变量 `initialOnes` 用于统计原始序列中 1 的个数。
2. **数据读取与统计**： 遍历输入，将数据存入 `a` 中，并同时计算 `initialOnes` 的值。
3. **构建收益序列**： 创建一个大小为 `n` 的 `vector gain` 。遍历 `a` ，根据 `a[i]` 的值（0 或 1）来填充 `gain` 数组（1 或 -1）。
4. **求解最大子数组和 （Kadane's Algorithm）**：
   - 初始化两个变量： `maxGain` （记录全局最大子段和） 和 `currentGain` （记录以当前元素结尾的最大子段和），并将它们都初始化为收益序列的第一个元素 `gain[0]` 。
   - 从收益序列的第二个元素开始遍历。在每一步中： a。 更新 `currentGain` ： `currentGain = max(当前收益值, currentGain + 当前收益值)` 。这一步的含义是，对于当前元素，我们要么将它自己作为一个新的子段的开始，要么将它加入到之前的子段中。 b. 更新 `maxGain` ： `maxGain = max(maxGain, currentGain)` 。用刚更新的 `currentGain` 来挑战全局最大值。
5. **计算最终结果**： 遍历结束后， `maxGain` 就代表了翻转操作能带来的最大收益。最终结果为 `initialOnes + maxGain` 。

四、核心代码讲解

6. 构建收益序列

代码块

```
// 步骤2: 构建收益序列
vector<int> gain(n);
for (int i = 0; i < n; ++i) {
    if (a[i] == 0) {
        gain[i] = 1; // 翻转0，收益为+1
    } else {
        gain[i] = -1; // 翻转1，收益为-1
    }
}
```

代码分析： 这段代码是算法设计的核心转化步骤。它创建了一个与原数组 `a` 等长的 `gain` 数组。循环遍历原数组 `a`，如果 `a[i]` 是 0，意味着翻转这个位置可以增加一个 1，因此在 `gain` 数组的对应位置记为 `1`。反之，如果 `a[i]` 是 1，翻转会减少一个 1，则记为 `-1`。这个 `gain` 数组完美地将原问题转化为了求最大子数组和的问题。

7. Kadane's Algorithm 实现

代码块

```
// 步骤3: 使用卡丹算法求解最大子数组和
int maxGain = gain[0];
int currentGain = gain[0];

for (int i = 1; i < n; ++i) {
    // 更新以当前元素结尾的最大子段和
    currentGain = max(gain[i], currentGain + gain[i]);
    // 更新全局最大子段和
    maxGain = max(maxGain, currentGain);
}
```

代码分析： 这是求解最大子数组和的经典实现——卡丹算法。

- `currentGain`：这个变量非常关键，它维护了**以当前元素 `gain[i]` 结尾的**连续子段的最大和。在每一步迭代中，它决定是延续之前的子段（`currentGain + gain[i]`）还是从当前元素 `gain[i]` 重新开始一个新的子段，取决于哪个选择能得到更大的和。
- `maxGain`：这个变量则负责在整个遍历过程中，持续记录 `currentGain` 曾达到过的最大值，从而确保在循环结束后，它存储的就是整个 `gain` 数组的全局最大子段和。

五、复杂度分析

- **时间复杂度： O（N）**
  - **推导过程**： 整个算法主要由三个独立的循环构成：
    i. 读取输入并统计初始 1 的个数，遍历一次数组，复杂度为 O（N）。
    ii. 构建收益序列 `gain`，遍历一次数组，复杂度为 O（N）。
    iii. 使用卡丹算法求解最大子数组和，遍历一次 `gain` 数组，复杂度为 O（N）。
  - 总的时间复杂度为 O（N） + O（N） + O（N） = O（N），与输入规模 `n` 呈线性关系。
- **空间复杂度： O（N）**
  - **推导过程**： 算法使用了两个与输入规模 `n` 相关的 `vector` 来存储数据：
    i. `vector<int> a` 存储原始序列，空间为 O（N）。
    ii. `vector<int> gain` 存储收益序列，空间为 O（N）。

- 因此，总的额外空间复杂度为 O（N）。（注：此算法可以被优化到 O（1）空间复杂度，即在输入的同时计算收益并运行卡丹算法，但 O（N）的实现思路更清晰）。

六、总结与反思

方法总结： 本题的解决思路是"**问题转化**"这一重要算法思想的绝佳体现。它将一个看起来需要暴力枚举所有子段（复杂度为 O（N^2））的翻转问题，通过巧妙地定义"收益"，成功转化为了一个可以在线性时间 O（N） 内解决的经典问题——最大子数组和。

优化与拓展:

- **空间优化**： 如复杂度分析中所述，我们无需完整存储 `gain` 数组。可以在读取输入 `a[i]` 后，立即计算出其对应的 `gain` 值，并直接在卡丹算法的循环中进行处理，从而将空间复杂度从 O（N） 优化到 O（1）。
- **问题拓展**： 如果题目改为"最多可以执行 K 次不重叠的翻转操作"，问题将变得更加复杂，可能需要使用动态规划来求解。

学习收获:

1. **识别问题模型**： 学会了如何从问题的描述中抽象出其数学或算法模型。当遇到求解"连续子段的最优值"这类问题时，应优先考虑是否能转化为最大子数组和或其变种问题。
2. **Kadane's Algorithm 的应用**： 深刻理解了卡丹算法的动态规划思想——即以当前位置为结尾的最优解，可以由前一位置的最优解推导出来。这不仅是解决最大子数组和的标准方法，其思想也适用于许多其他动态规划问题。

树的优化 （ID: H5-10）

一、问题描述

> 在一个原始森林里，有人发现了一颗根编号为 1 的神奇树，它的每个顶点以及每条边上都标有一个数字。
>
> 然而，他发现这颗树上有些顶点有瑕疵，也称为瑕疵点。一个顶点 `v` 被称为瑕疵点是指在它的子树中存在点 `u`，使得 `dist(v, u) > a[u]`，这里 `a[u]` 是标注在顶点 `u` 上的数字，而 `dist(v, u)` 是所有标注在从顶点 `v` 到顶点 `u` 的路径上边的数字之和。
>
> 这人决定删除一些叶子节点，直到整颗树不存在任何瑕疵点。那么，需要删除的叶子节点的最少数是多少?
>
> **【输入形式】**
>
> 输入的第一行为一个整数 `n` （1 ≤ n ≤ 10^5）。
>
> 接下来一行为 `n` 个整数 `a[1]`， `a[2]`， …， `a[n]` （1 ≤ a[i] ≤ 10^9），这里 `a[i]` 是标注在顶点 `i` 上的数字。

接下来的 `n-1` 行描述了树中边的情况，第 `i` 行有两个整数 `p[i]` 和 `c[i]` （1 ≤ `p[i]` ≤ n， -10^9 ≤ `c[i]` ≤ 10^9），这意味着在顶点 `i+1` 和 `p[i]` 之间有边相连，其上标有数字 `c[i]`。

【输出形式】

输出一个整数，表示需要删除的最少节点数。

【样例输入】

代码块

```
9
88 22 83 14 95 91 98 53 11
3 24
7 -8
1 67
1 64
9 65
5 12
6 -80
3 8
```

【样例输出】

代码块

```
5
```

二、问题分析与边界条件

问题核心

本题的核心要求是，通过删除最少数目的叶子节点，来修剪一棵带权树，使得修剪后保留的树中不存在任何“瑕疵点”。最终需要输出被删除的节点总数。

思路拆解

1. **理解“瑕疵点”**：一个节点 `v` 之所以成为瑕疵点，是因为它的子树里某个后代节点 `u` 不满足 `dist(v, u) <= a[u]` 的条件。
2. **反向思考**：“删除最少的节点”等价于“保留最多的节点”。我们的目标是找到一个最大的子树，其中所有节点都不会让其祖先成为瑕疵点。
3. **推导保留条件**：一个节点 `u` 能够被保留，当且仅当对于它的**所有**祖先节点 `v`，都必须满足 `dist(v, u) <= a[u]`。如果一个节点 `u` 使得它的某个祖先 `v` 成为瑕疵点，那么无论我们

怎么删除 `u` 的后代，`u` 本身对 `v` 的影响是无法消除的。因此，唯一的办法就是将 `u` 及其整个子树都删除。

4. **数学化简**：引入一个辅助定义 `d(x)`，表示从根节点 `1` 到节点 `x` 的路径权值和。那么 `dist(v, u)` 可以表示为 `d(u) - d(v)`。保留条件 `dist(v, u) <= a[u]` 就转化为 `d(u) - d(v) <= a[u]`，进一步整理为 `d(u) - a[u] <= d(v)`。由于这个不等式必须对 `u` 的所有祖先 `v` 成立，所以它等价于一个更强的条件：`d(u) - a[u]` 必须小于等于其所有祖先路径和中的最小值。

边界条件

1. **数据规模**：`n` 最大为 10^5，这意味着算法的时间复杂度必须是 `O(N)` 或 `O(N log N)` 级别。
2. **数值范围**：节点上的数字 `a[i]` 和边权 `c[i]` 的绝对值都可能很大（10^9），路径和的计算可能会超出 32 位整型的范围，因此必须使用 `long long` 类型来存储路径和。
3. **负权边**：边权 `c[i]` 可以是负数，这意味着从根节点出发，路径和并非单调递增，可能会减小。这在推导保留条件时需要特别注意，但最终的数学化简结果已经完美地处理了这种情况。

三、算法设计

核心思想

本题采用"条件检查与剪枝"的思路，通过两次深度优先搜索（DFS）来解决。核心思想是：先将复杂的"瑕疵点"定义转化为一个对每个节点都适用的、本地化的"保留条件"，然后遍历树进行检查，一旦发现某个节点不满足该条件，就将其整个子树"剪枝"（即计入删除总数），不再继续深入。

数据结构

1. **邻接表** `vector<vector<pair<int, int>>> adj`：用于存储树的结构。`adj[u]` 存储一个列表，其中每个元素 `{v, w}` 代表 `u` 的一个子节点 `v` 以及连接它们的边权 `w`。这是表示树或图的高效方式。
2. **子树大小数组** `vector<int> subtree_size`：`subtree_size[u]` 存储以节点 `u` 为根的子树中的节点总数（包括 `u` 自身）。这个数组用于优化剪枝操作：当我们决定删除节点 `u` 时，可以直接将被删除的节点数增加 `subtree_size[u]`，而无需再遍历其子树。
3. **节点值数组** `vector<ll> a`：存储每个节点 `i` 上的数字 `a[i]`。使用 `long long`（`ll`）类型以匹配路径和的类型。

算法流程

1. **输入与建树**：读取输入数据，构建邻接表来表示这棵树。
2. **第一次 DFS（计算子树大小）**：实现 `dfs_size(u)` 函数。从根节点 `1` 开始进行一次后序遍历。对于每个节点 `u`，其子树大小等于其所有子节点的子树大小之和，再加上 1（节点 `u` 自身）。

3. **第二次 DFS （检查与剪枝）**：实现 `dfs_check(u, path_sum, min_ancestor_sum)` 函数，从根节点 `1` 开始进行一次前序遍历。
   - `u`： 当前遍历到的节点。
   - `path_sum`： 从根 `1` 到 `u` 的路径和，即 `d(u)`。
   - `min_ancestor_sum`： `u` 的所有祖先 `v` 的路径和 `d(v)` 中的最小值。
4. **检查逻辑**： 在 `dfs_check` 函数中，对于节点 `u`，检查其是否满足保留条件： `path_sum - a[u] <= min_ancestor_sum`。
   - **若不满足**： 说明节点 `u` 必须被删除，而删除它意味着它的整个子树都无法保留。此时，将 `subtree_size[u]` 累加到总删除计数器 `total_deleted_count` 中，并立即返回，不再递归其子节点（实现剪枝）。
   - **若满足**： 说明节点 `u` 本身可以被保留。继续对其所有子节点 `v` 进行递归调用 `dfs_check`。在调用时，更新参数：新的路径和为 `path_sum + w` （w 为边 uv 的权值），新的最小祖先路径和为 `min(min_ancestor_sum, path_sum)`。
5. **初始化调用**： 主程序中，对根节点 `1` 调用 `dfs_check(1, 0, LLONG_MAX)`。 `path_sum` 初始为 0， `min_ancestor_sum` 初始为一个极大的值，以确保根节点自身不会因为这个检查而被错误地删除。
6. **输出结果**： 第二次 DFS 结束后， `total_deleted_count` 的值即为最终答案。

四、核心代码讲解

7. 计算子树大小 （`dfs_size`）

代码块

```
/**
 * @brief 通过后序遍历计算以每个节点为根的子树大小。
 * @param u 当前节点。
 */
void dfs_size(int u) {
    subtree_size[u] = 1; // 初始化为1，包含节点u自身
    for (auto const& [v, w] : adj[u]) { // 遍历所有子节点v
        dfs_size(v); // 递归计算子节点的子树大小
        subtree_size[u] += subtree_size[v]; // 累加到父节点的子树大小中
    }
}
```

讲解： 这是一个标准的后序遍历应用。函数首先递归地调用自身，处理完当前节点 `u` 的所有子树。当递归返回时，每个子节点 `v` 的 `subtree_size[v]` 已经计算完毕。然后，将这些子树的大小累加到 `subtree_size[u]` 上。 `subtree_size[u]` 初始化为 1，代表了节点 `u` 本身。这样，

当 `dfs_size(u)` 执行完毕后，`subtree_size[u]` 就正确地存储了以 `u` 为根的整个子树的节点数量。

8. 检查与剪枝 （`dfs_check`）

代码块

```
/**
 * @brief 主DFS函数，检查节点是否有效，并累加需要删除的节点数。
 * @param u 当前正在处理的节点。
 * @param path_sum 从根到节点u的路径上边权之和。
 * @param min_ancestor_sum 节点u的所有祖先v的path_sum的最小值。
 */
void dfs_check(int u, ll path_sum, ll min_ancestor_sum) {
    // 核心检查：path_sum(u) - a[u] <= min(path_sum(v)) 对所有祖先v成立
    if (path_sum - a[u] > min_ancestor_sum) {
        // 如果条件被违反，则 u 和它的整个子树都必须被删除。
        total_deleted_count += subtree_size[u];
        // 剪枝：不再向下遍历这个分支
        return;
    }

    // 更新其子节点将要继承的 "min_ancestor_sum"
    ll next_min_ancestor_sum = min(min_ancestor_sum, path_sum);

    // 对所有子节点进行递归检查
    for (auto const& [v, w] : adj[u]) {
        dfs_check(v, path_sum + w, next_min_ancestor_sum);
    }
}
```

讲解： 这是算法的核心实现。

- **参数传递**： 函数通过参数维护了遍历过程中的关键状态：`path_sum` （即 `d(u)`） 和 `min_ancestor_sum`。
- **核心判断**： `if (path_sum - a[u] > min_ancestor_sum)` 这一行代码直接实现了我们在问题分析中推导出的保留条件的**否定形式**。如果这个条件成立，意味着节点 `u` 无法被保留。
- **剪枝操作**： 如果节点 `u` 无法保留，我们利用预先计算好的 `subtree_size[u]`，一次性将整个子树的节点数加入 `total_deleted_count`。`return` 语句则起到了剪枝的效果，避免了对一个已知要被完全删除的子树进行不必要的遍历。
- **状态更新**： 如果节点 `u` 被保留，它将成为其子孙的一个新祖先。因此，在向下递归到子节点 `v` 之前，需要更新 `min_ancestor_sum`。新的最小值是当前 `min_ancestor_sum` 和 `u` 自身的 `path_sum` 中的较小者。同时，子节点 `v` 的路径和也更新为 `path_sum + w`。

五、复杂度分析

- **时间复杂度： O（N） 推导过程**： 整个算法由两次独立的深度优先搜索（DFS）构成。第一次 `dfs_size` 遍历树中的每个节点和每条边恰好一次，以计算子树大小，其复杂度为 O（N）。第二次 `dfs_check` 同样遍历全树，对于每个节点，要么进行一次判断并继续递归，要么判断后剪枝。在整个过程中，每个节点最多被访问一次。因此，总的时间复杂度是 O（N） + O（N） = O（N）。
- **空间复杂度： O（N） 推导过程**： 算法需要额外空间来存储树的结构和辅助信息。
  a. 邻接表 `adj` 需要 O（N） 的空间来存储 N-1 条边。
  b. 数组 `a` 和 `subtree_size` 都需要 O（N） 的空间。
  c. DFS 递归调用的栈深度在最坏情况下（树退化成一条链）可能达到 O（N）。 因此，总的空间复杂度由这些部分决定，为 O（N）。

六、总结与反思

方法总结

本题是一个在树上进行剪枝优化的典型问题。其解决思路展现了几个重要的算法思想：

1. **逆向思维**： 将"最小化删除"问题转化为"最大化保留"问题，从而更容易找到问题的约束条件。
2. **数学建模**： 关键在于将题目中对"瑕疵点"的描述性定义，通过引入路径和的概念，转化为一个清晰、普适的数学不等式 `d(u) - a[u] <= min(d(v))`，这个转化是解题的突破口。
3. **两遍 DFS 模式**： 这是一种在树形问题中常用的高效模式。第一遍 DFS（通常是后序遍历）用于收集子树的统计信息（如本题的子树大小、或其它问题中的深度、节点属性等）。第二遍 DFS（通常是前序遍历）则利用这些全局或半全局的信息，自顶向下地进行决策和计算。

学习收获

通过解决此题，可以深刻理解深度优先搜索在处理树形问题时的灵活性和强大功能。特别是学会了如何将一个看似全局的、依赖祖先与后代关系的复杂约束，转化为一个可以在 DFS 过程中通过传递参数来高效检查的本地化条件。此外，使用 `long long` 来预先防止整数溢出，是处理大数据范围问题时必须养成的良好编程习惯。预计算子树大小以实现高效剪枝的技巧，也为优化其他树相关算法提供了有益的参考。