Задача H

1) Алгоритм

Для решения задачи использовался метод бинарного подъема, который позволяет эффективно находить наименьшего общего предка двух вершин. Зная LCA двух узлов, можно легко вычислить расстояние между ними, что позволяет определить количество необходимой энергии для перехода между двумя временными точками.

Основные этапы алгоритма:

Метод `buildTimelineTree` используется для построения дерева связей начиная с корневой вершины. В процессе построения происходит вычисление входного и выходного времени (`entryTime`, `exitTime`) каждой вершины, что помогает в дальнейшем проверять, является ли одна вершина предком другой.

Заполняется массив `binaryLift`, который позволяет находить предков вершины на различных уровнях, что полезно для быстрого поиска LCA.

Поиск наименьшего общего предка:

Метод `findConvergencePoint` находит LCA двух временных точек. В этом методе используются результаты бинарного подъема (`binaryLift`) для эффективного подъема от текущих вершин к их предкам, пока не будет найден наименьший общий предок.

Вычисление расстояния между двумя точками:

Метод `calculateTimeDistance` использует найденный LCA для вычисления расстояния между двумя временными точками. Расстояние между двумя узлами вычисляется как сумма глубин (`timelineDepth`) двух точек минус удвоенная глубина их общего предка.

2) Сложность решения

Построение дерева - проход по всем вершинам и ребрам для построения дерева занимает O(N), где N — количество вершин. Сложность поиска наименьшего общего предка составляет O(logN) за счет использования бинарного подъема. Каждый запрос обрабатывается за O(logN) из-за необходимости вычисления LCA и расстояния между двумя точками.

Таким образом, суммарная временная сложность составляет O((N+Q)·logN), где Q - количество запросов.

Памятная сложность - используемые структуры данных, такие как `timeConnections`, `binaryLift`, `timelineDepth`, `entryTime`, и `exitTime`, требуют O(NlogN) памяти.

3) Распечатка кода

```
#include <cmath>
#include <iostream>
#include <vector>

const int MAX_BINARY_LIFT = 17;

class TimelineManager {
private:
  std::vector<std::vector<int>> timeConnections;
  std::vector<std::vector<int>> binaryLift;
  std::vector<int> timelineDepth;
  std::vector<int> entryTime;
  std::vector<int> exitTime;
  int timeCounter;

public:
  TimelineManager(int size) : timeCounter(0) {
    timeConnections.resize(size);
    binaryLift.resize(size,
std::vector<int>(MAX_BINARY_LIFT, -1));
    timelineDepth.resize(size, 0);
    entryTime.resize(size, 0);
    exitTime.resize(size, 0);
  }

  void addConnection(int from, int to) {
    timeConnections[from].push_back(to);
    timeConnections[to].push_back(from);
  }

  void buildTimelineTree(int currentNode, int parentNode) {
    entryTime[currentNode] = ++timeCounter;
    binaryLift[currentNode][0] = parentNode;

    for (int level = 1; level < MAX_BINARY_LIFT; ++level) {
      if (binaryLift[currentNode][level - 1] != -1)
```

```
        binaryLift[currentNode][level] =
binaryLift[binaryLift[currentNode][level - 1]][level - 1];
      else
        binaryLift[currentNode][level] = -1;
    }

    for (int nextNode : timeConnections[currentNode]) {
      if (nextNode != parentNode) {
        timelineDepth[nextNode] = timelineDepth[currentNode]
+ 1;
        buildTimelineTree(nextNode, currentNode);
      }
    }
    exitTime[currentNode] = ++timeCounter;
  }

  bool isTimelineAncestor(int ancestor, int descendant) {
    return entryTime[ancestor] <= entryTime[descendant] &&
           exitTime[ancestor] >= exitTime[descendant];
  }

  int findConvergencePoint(int timeline1, int timeline2) {
    if (isTimelineAncestor(timeline1, timeline2))
      return timeline1;
    if (isTimelineAncestor(timeline2, timeline1))
      return timeline2;

    for (int i = MAX_BINARY_LIFT - 1; i >= 0; --i) {
      if (binaryLift[timeline1][i] != -1 &&
          !isTimelineAncestor(binaryLift[timeline1][i],
timeline2))
        timeline1 = binaryLift[timeline1][i];
    }
    return binaryLift[timeline1][0];
  }

  int calculateTimeDistance(int point1, int point2) {
    int convergence = findConvergencePoint(point1, point2);
```

```
    return timelineDepth[point1] + timelineDepth[point2] - 2
* timelineDepth[convergence];
  }
};

int main() {
  std::ios_base::sync_with_stdio(false);
  std::cin.tie(nullptr);

  int vertexCount;
  std::cin >> vertexCount;

  TimelineManager timeline(vertexCount);

  for (int i = 0; i < vertexCount - 1; ++i) {
    int from, to;
    std::cin >> from >> to;
    timeline.addConnection(from, to);
  }

  timeline.buildTimelineTree(0, -1);

  int queryCount;
  std::cin >> queryCount;
  while (queryCount--) {
    int start, end, energy;
    std::cin >> start >> end >> energy;
    int requiredEnergy =
timeline.calculateTimeDistance(start, end);
    std::cout << (requiredEnergy <= energy ? "Yes\n" :
"No\n");
  }

  return 0;
}
```