OMI 2022 - Solución para *Transporte público*

SUBTAREA 1: $d_i = 1$. La red de transporte es una línea donde el nodo i es padre del nodo $i + 1$

Debido a la estructura del árbol (una línea) para cada pareja de nodos a_i, b_i de una pregunta se puede realizar un recorrido que vaya desde el nodo b_i hasta el nodo N y que pase por a_i. Dado que todos los recorridos tienen una distancia $d_i = 1$ entonces el largo de dicho recorrido será $largo = N - b_i$.

El siguiente código resuelve esta subtarea:

```
while(q--){
  cin >> a >> b;
  cout << n - b << "\n";
}
```

SUBTAREA 2: $d_i = 1$. El nodo 1 es padre de todos los demás nodos de la red.

Esta configuración de árbol se conoce como *estrella* ya que hay un nodo central al que todos los demás están conectados. Dado que en cada pregunta b_i tiene que ser igual a a_i o un ancestro de a_i solo existen dos opciones distintas:

- $b_i \neq 1$: En este caso $a_i = b_i$ y son una hoja de la estrella y no se puede hacer ningún recorrido, por lo que el resultado es 0.
- $b_i = 1$: En este caso b_i es la raíz y a_i puede ser la misma raíz o una hoja cualquiera. En cualquier caso se puede hacer un recorrido que vaya de una hoja a_i si es hoja a otra hoja a través de la raíz. Dicho recorrido siempre tendrá longitud 2.

El siguiente código resuelve esta subtarea:

```
while(q--){
  cin >> a >> b;
  if(b == 1) cout << "2\n";
  else cout << "0\n";
}
```

SUBTAREA 3: $N, Q \leq 1000$

Los límites de la subtarea te permiten resolver cada pregunta de manera independiente revisando el camino máximo que pasa por a_i.

Observa que un recorrido que pase por a_i debe *llegar* a a_i por una de las estaciones a las que a_i está conectada y debe *salir* posteriormente hacia otra estación distinta. Esto es para evitar pasar por el mismo lugar.

Si revisas todas las estaciones a las que a_i está conectada y para cada una ves cuál es el camino más largo entonces el camino más largo que pase por a_i será la suma de los dos caminos más largos de sus hijos.

ATENCIÓN: Recuerda que el nodo b_i define un límite del cual no se puede pasar. Por eso el padre de b_i debe marcarse como una estación a la que no se puede llegar.

La subtarea anterior se puede resolver usando un recorrido dfs como el que se muestra en el siguiente código:

```cpp
int n, q, a, b, d, m1, m2, t, padre[100003];
vector<pair<int, int> > hijos[100003];

void dfspadre(int nodo, int p){
    padre[nodo] = p; // MARCA EL PADRE DE CADA NODO
    for(auto h : hijos[nodo]){
        if (h.first == p) continue; // NO REVISES A SU PADRE
        dfspadre(h.first, nodo);
    }
}

int dfsmax(int nodo, int p, int prohibido){
    int tmp, res = 0;
    for(auto h : hijos[nodo]){
        // SI ES DE DONDE VIENES O EL NODO PROHIBIDO, NO VAYAS PARA ALLA
        if (h.first == p || h.first == prohibido) continue;
        tmp = dfsmax(h.first, nodo, prohibido) + h.second;
        res = max(res, tmp); // GUARDA EL MEJOR CAMINO
    }
    return res;
}

int main() {
  cin >> n >> q;
  for(int i = 1; i < n; ++i){
    cin >> a >> b >> d;
    // GUARDA LOS HIJOS CON SUS DISTANCIAS
    hijos[a].push_back({b, d});
    hijos[b].push_back({a, d});
  }

  // OBTEN EL PADRE DE CADA NODO
  dfspadre(1, 0);
  while(q--){
    cin >> a >> b;

    // OBTEN LAS DOS ESTACIONES CON MAYOR CAMINO DESDE a
    m1 = m2 = 0;
    for(auto h : hijos[a]){
        if (h.first == padre[b]) continue; // NO VAYAS AL PROHIBIDO
        // SUMA EL CAMINO MAS LARGO Y LA DISTANCIA DE a A ESA ESTACION
        t = dfsmax(h.first, a, padre[b]) + h.second;
        if (t > m1){ // ES EL MAXIMO HASTA EL MOMENTO
            m2 = m1;
            m1 = t;
        }
        // ES EL SEGUNDO MAS LARGO
        else if (t > m2) m2 = t;
    }
    cout << m1 + m2 << "\n";
  }

  return 0;
}
```

Para cada pregunta es posible que la DFS revise los N nodos, por lo tanto la complejidad de la solución es $O(NQ)$

SUBTAREA 4: $a_i = b_i$ para todas las preguntas. El *recorrido* debe pasar por la estación que define

la zona.

En la subtarea anterior, para contestar cada pregunta conviertes a a_i en la raíz de un árbol temporal y mediante una DFS buscas las dos ramas más *largas* desde la raíz. La necesidad de hacer un árbol temporal viene de que las dos ramas más *largas* pueden estar en el subárbol original de a pero también en los ancestros de a por lo que te ves forzado a convertir a a en la raíz y tratarlo cada vez como un nuevo árbol.

Pero en esta subtarea te aseguran que $a_i = b_i$ eso quiere decir que nunca se puede subir a los ancestros de a y por lo tanto el camino más largo siempre se forma entre dos hijos de a. Por lo tanto, para cada nodo basta con conocer sus dos hijos más *largos* para contestar cada pregunta. Esto se puede hacer para todos los nodos en una única dfs como la siguiente:

```
long long dfsmax(int nodo, int p){
  long long tmp;
  for(auto h : hijos[nodo]){
      if (h.first == p) continue; // NO REVISES A SU PADRE
      tmp = dfsmax(h.first, nodo) + h.second;

      // GUARDA LOS DOS HIJOS MAS LARGOS DE CADA NODO
      if (tmp > m[0][nodo]){
          m[1][nodo] = m[0][nodo];
          m[0][nodo] = tmp;
      }
      else if (tmp > m[1][nodo]) m[1][nodo] = tmp;
  }

  // DEVUELVE SU CAMINO MAS LARGO
  return m[0][nodo];
}

int main() {
  cin >> n >> q;
  for(int i = 1; i < n; ++i){
    cin >> a >> b >> d;
    // GUARDA LOS HIJOS CON SUS DISTANCIAS
    hijos[a].push_back({b, d});
    hijos[b].push_back({a, d});
  }

  // OBTEN LOS HIJOS MAS LARGOS DE CADA NODO
  dfsmax(1, 0);
  while(q--){
    cin >> a >> b;

    cout << m[0][a] + m[1][a] << "\n";
  }

  return 0;
}
```

Este código si cambias la salida a que sea `cout << m[0][b] + m[1][b]` en vez de usar a te da 35 puntos en vez de 15 ya que puede resolver además las primeras dos subareas. ¿Ubicas por qué?

El recorrido DFS inicial es de complejidad N, contestar cada pregunta se puede hacer en tiempo constante, por lo tanto la complejidad es $O(N)$.

SUBTAREAS 5 y 6.

Para las últimas dos subtareas requieres poder contestar de forma eficiente cuales son los 2 más *largos* de entre los dos hijos más *largos* de un nodo y el camino máximo que se pueda recorrer usando sus ancestros hasta un

cierto límite (en este caso el nodo b).

Saber cuáles son los dos hijos más *largos* se puede hacer utilizando la dfs de la subtarea anterior y almacenando para cada nodo los mejores dos candidatos entre sus hijos.

Analiza cuál es el recorrido más largo que se puede hacer hacia los ancestros del nodo a. Sea $C = c_0, c_1, \ldots, c_m$ el conjunto de nodos entre los nodos b y a tal que $c_0 = b$, $c_m = a$ y c_i es padre de c_{i+1}. Sea además $rama(c_i)$ la longitud del camino más largo en el sub árbol de c_i que no pasa por c_{i+1}, es decir, que desde c_i usa un hijo distinto a los del camino entre b y a. Por último, sea $dist(u, v)$ la distancia del recorrido entre los nodos u y v El máximo recorrido de a hacia sus ancentros es entonces:

$max(rama(c_i) + dist(c_i, a))$ para $0 \leq i < m$

Observa que si se tienen dos nodos del conjunto c_i y c_j con $i < j$ entonces las opciones de recorridos por ambos son:

Recorrido por c_i es $R(c_i) = rama(c_i) + dist(c_i, c_j) + dist(c_j, a)$

Recorrido por c_j es $R(c_j) = rama(c_j) + dist(c_j, a)$

Por lo tanto $R(c_j) > R(c_i)$ si $rama(c_j) > rama(c_i) + dist(c_i, c_j)$ **independientemente** del nodo a.

IMPORTANTE: Detente aquí hasta que la parte anterior te quede completamente clara. La desigualdad anterior te permite decidir que si un $rama(c_j)$ es mayor que los recorridos por sus ancestros, entonces cuando vengas de un descendiente de c_j nunca será necesario revisar más arriba de c_j. Y dicha observación es la que permite resolver el problema de manera eficiente.

Una vez que hagas la observación anterior, existen distintas maneras de contestar la pregunta. El código de la solución oficial utiliza un *monotone stack* y *sparse tables* (si no conoces dichas estructuras de datos, te recomiendo que investigues sobre ambas) para resolver el problema de la siguiente manera:

Monotone stack

El *monotone stack* es una estructura de datos que te permite contestar de forma eficiente en un arreglo cuál es el primer elemento a la izquierda que es mayor/menor que el elemento i. El *monotone stack* es una estructura muy sencilla de implementar, a veces en programación competitiva se le conoce como *stack trick*.

De la observación sobre los recorridos por c_i y c_j se concluyó que una vez que bajando por el árbol encuentras un candidato mejor, nunca más será necesario revisar ancestros arriba de él. Si el límite de b no existiera, bastaría con ir recordando el mejor candidato. Como el límite de b existe, recordar el mejor candidato no basta, ya que el mejor candidato puede estar en un nivel superior al de b y por tanto ser inaccesible para esa pregunta. Para eso utilizamos el *monotone stack* en el que cada nodo guarda el primer candidato hacia arriba que sea mejor que él (el *monotone stack* contesta eficientemente cuál es el primer elemento hacia arriba que es *mejor* que yo). De esa forma, cuando se quiere saber el mejor ancestro para el nodo a se puede ir recorriendo el *monotone stack* de mejor en mejor candidato siempre y cuando no pasemos el límite de b.

El simple *monotone stack* no es suficiente para resolver la subtarea 6 ya que es posible que en un árbol, cada hijo vaya siendo mejor candidato que su padre y cuando tengamos que contesar una pregunta sobre un nodo a con un nodo b muy cercano a la raíz, sea necesario ir recorriendo todos los nodos lo que nos llevaría de nuevo a una solución $O(NQ)$.

Para poder *avanzar* rápido sobre el *monotone stack* se utiliza la estructura conocida como *sparse table*.

Sparse table

Una *sparse table* es una tabla de tamaño $N \times logN$ que permite contestar una pregunta sobre un rango en tiempo logarítmico. A grandes rasgos su funcionamiento es el siguiente. Cada nodo tiene asociada una lista de $logN$ valores que representan la información necesaria para contestar la pregunta que se desee para un intervalo de largo 2^0, uno de 2^1, uno de 2^2, ..., 2^{logN}. De esa forma, supón que se desea contestar con la *sparse table* una pregunta para el intervalo, por ejemplo, $[3, 28]$. La distancia entre 28 y 3 es $28 - 3 = 25$ cuya representación binaria es 11001, es decir tiene encendidos los bits que corresponden a las potencias $[0, 3, 4]$. Usando el *sparse table* se contesta la pregunta realizando los siguientes pasos:

- Para el nodo 3 (inicio del intervalo) usa el valor asociado de 2^0, es decir, la respuesta para el intervalo $[3, 4)$.
- Desde el nodo 3 salta 2^0 lugares hasta 4.
- Para el nodo 4 usa el valor asociado a 2^3, es decir, la respuesta para el intervalo $[4, 12)$.
- Desde el nodo 4 salta 2^3 lugares hasta 12.
- Para el nodo 12 usa el valor asociado a 2^4, es decir, la respuesta para el intervalo $[12, 26)$.

- Une los intervalos para los que tienes respuesta $[3,4) \cup [4,12) \cup [12,26) = [3,25]$

Dado que a lo más usas un valor por bit encendido de la diferencia, nunca se revisarán más de $logN$ valores.

La solución oficial utiliza un *sparse table* para calcular de manera las distancias entre el nodo a y cualquiera de sus ancestros y además para poder saltar por los elementos del *monotone stack*.

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

#define debugsl(x) std::cout << #x << " = " << x << ", "
#define debug(x) debugsl(x) << "\n";

#define lli long long

#define MAX 100000
#define LOG 17

#define distancia first
#define hijo second
#define idancestro second

#define idpregunta first

lli n, q, a, b, d, id, padre[MAX + 2];
std::vector<std::pair<lli, int> > hijos[MAX + 2];
std::vector<std::pair<int, int> > preguntas[MAX + 2];
int nivel[MAX + 2], max_hijo[MAX + 2], dist_max_hijo[MAX + 2];
lli respuesta[MAX + 2];
std::pair<lli, int> dist_padre[LOG + 1][MAX + 2];
lli hijos_profundos[2][MAX + 2];

typedef struct monotone_node {
  lli largo_hijo;
  lli dist_acum;
  int ancestro;
};
monotone_node ms[MAX + 2];
int st_ancestros[LOG + 1][MAX + 2];

void dfs_init(lli nodo, lli pad, lli niv, lli distpadre) {
  nivel[nodo] = niv;
  padre[nodo] = pad;
  dist_padre[0][nodo] = {distpadre, pad};
  for (int bit = 1; bit <= LOG; ++bit) {
    dist_padre[bit][nodo].idancestro =
        dist_padre[bit - 1][dist_padre[bit - 1][nodo].idancestro].idancestro;
    dist_padre[bit][nodo].distancia =
        dist_padre[bit - 1][nodo].distancia +
        dist_padre[bit - 1][dist_padre[bit - 1][nodo].idancestro].distancia;
  }

  for (auto h : hijos[nodo]) {
    if (h.hijo == pad) continue;
    dfs_init(h.hijo, nodo, niv + 1, h.distancia);
    if (hijos_profundos[0][h.hijo] + h.distancia > hijos_profundos[0][nodo]) {
      max_hijo[nodo] = h.hijo;
      dist_max_hijo[nodo] = h.distancia;
      hijos_profundos[1][nodo] = hijos_profundos[0][nodo];
      hijos_profundos[0][nodo] = hijos_profundos[0][h.hijo] + h.distancia;
    } else if (hijos_profundos[0][h.hijo] + h.distancia >
               hijos_profundos[1][nodo]) {
      hijos_profundos[1][nodo] = hijos_profundos[0][h.hijo] + h.distancia;
    }
  }
}
```

```
lli distancia(lli u, lli v) {

  int nu = nivel[u];
  int nv = nivel[v];
  int dif = nu - nv;
  lli sum = 0;
  for (int bit = 0; bit <= LOG; ++bit) {
    if (dif & (1 << bit)) {
      sum += dist_padre[bit][u].distancia;
      u = dist_padre[bit][u].idancestro;
    }
  }
  return sum;
}

void actualiza_monotone_stack(lli nodo, lli disthijo, lli distpadre) {
  lli pos = padre[nodo];
  lli dist_acumulada = distpadre;
  while (pos && disthijo >= dist_acumulada + ms[pos].largo_hijo) {
    dist_acumulada += ms[pos].dist_acum;
    pos = ms[pos].ancestro;
  }
  ms[nodo].largo_hijo = disthijo;
  ms[nodo].dist_acum = dist_acumulada;
  ms[nodo].ancestro = pos;

  st_ancestros[0][nodo] = pos;
  for (int bit = 1; bit <= LOG; ++bit)
    st_ancestros[bit][nodo] =
        st_ancestros[bit - 1][st_ancestros[bit - 1][nodo]];
}

void dfs_solve(lli nodo, lli pad, lli distpadre) {
  // RESUELVE TODAS LAS PREGUNTAS QUE PASAN POR ESTE NODO
  for (auto p : preguntas[nodo]) {
    b = nivel[p.hijo];  // OBTEN EL NIVEL DEL ANCESTRO

    // HAZ UNA BUSQUEDA BINARIA SOBRE EL MONOTONE STACK PARA VER EL MAXIMO
    // DEBAJO DE b
    lli pos = pad;
    for (int bit = LOG; bit >= 0; --bit) {
      if (nivel[st_ancestros[bit][pos]] >= b) pos = st_ancestros[bit][pos];
    }

    lli dist_ancestro = 0;
    if (nivel[pos] >= b)
      dist_ancestro = distancia(nodo, pos) + ms[pos].largo_hijo;

    lli x = dist_ancestro;
    lli y = hijos_profundos[0][nodo];
    lli z = hijos_profundos[1][nodo];

    if (x < y) std::swap(x, y);
    if (y < z) std::swap(y, z);

    respuesta[p.idpregunta] = x + y;
  }

  // RESUELVE PARA EL HIJO MAXIMO
  if (max_hijo[nodo]) {
    actualiza_monotone_stack(nodo, hijos_profundos[1][nodo], distpadre);
    dfs_solve(max_hijo[nodo], nodo, dist_max_hijo[nodo]);
  }
```

```
  // RESUELVE PARA LOS DEMAS HIJOS
  if (hijos[nodo].size() > 2 || (hijos[nodo].size() == 2 && nodo == 1)) {
    actualiza_monotone_stack(nodo, hijos_profundos[0][nodo], distpadre);
    for (auto h : hijos[nodo]) {
      if (h.hijo == pad || h.hijo == max_hijo[nodo]) continue;
      dfs_solve(h.hijo, nodo, h.distancia);
    }
  }
}

int main() {
  // LEE LA ENTRADA
  std::cin >> n >> q;
  for (int i = 1; i < n; ++i) {
    std::cin >> a >> b >> d;
    hijos[a].push_back({d, b});
    hijos[b].push_back({d, a});
  }
  for (int i = 1; i <= q; ++i) {
    std::cin >> a >> b;
    preguntas[a].push_back({i, b});
  }

  dfs_init(1, 0, 1, 0);

  dfs_solve(1, 0, 0);

  for (int i = 1; i <= q; ++i) std::cout << respuesta[i] << "\n";

  return 0;
}
```