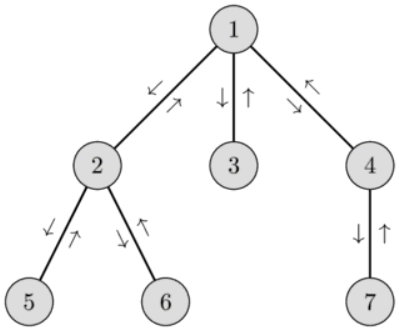
GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: EL ANCESTRO COMÚN MÁS BAJO (*LOWEST COMMON ANCESTOR* (LCA))

---

1. Introducción

El ancestro común más bajo (*Lowest Common Ancestor* (LCA)) es un concepto dentro de la Teoría de grafos y Ciencias de la computación. Sea T un árbol con raíz y n nodos. El ancestro común más bajo entre dos nodos v y w se define como el nodo más bajo en T que tiene a v y w como descendientes (donde se permite a un nodo ser descendiente de él mismo).



Para el caso de la imagen anterior el $LCA(5,6) = 2$, $LCA(3,6) = 1$ y $LCA(2,6) = 2$.

2. Conocimientos previos

2.1. Recorrido de Euler

El recorrido de Euler se define como una forma de atravesar el árbol de modo que cada vértice se agrega al recorrido cuando lo visitamos (ya sea moviéndose hacia abajo desde el vértice principal o regresando desde el vértice secundario). Comenzamos desde la raíz y volvemos a la raíz después de visitar todos los vértices. Requiere exactamente $2N - 1$ vértices para almacenar el recorrido de Euler.

2.2. Árbol de Rango (*Range Tree*)

Un árbol de rangos es una estructura de datos que almacena información sobre intervalos de un arreglo como un árbol. Esto permite responder consultas de rango sobre un arreglo de manera eficiente, mientras sigue siendo lo suficientemente flexible como para permitir una modificación rápida del arreglo. Esto incluye encontrar la suma de elementos del arreglo consecutivos $a[l \dots r]$, o encontrar el elemento mínimo en tal rango en $O(\log n)$ tiempo. Entre las respuestas a tales consultas, el árbol de rango permite modificar el arreglo reemplazando un elemento, o incluso cambiando los elementos de un subrango completo (por ejemplo, asignando todos los elementos $a[l \dots r]$ a cualquier valor, o agregando un valor a todos los elementos en el subrango).

2.3. Descomposición Sqrt (*Sqrt Decomposition*)

Descomposición Sqrt es un método (o una estructura de datos) que le permite realizar algunas operaciones comunes (encontrar la suma de los elementos del subarreglo, encontrar el elemento

mínimo/máximo, etc.) en $O(\sqrt{n})$ operaciones, que es mucho más rápido que $O(n)$ para el algoritmo trivial.

2.4. Tabla dispersa (*Sparse Table*)

Tabla dispersa es una estructura de datos que permite responder consultas de rango. Puede responder a la mayoría de las consultas de rango en $O(\log n)$, pero su verdadero poder es responder consultas de rango mínimo (o consultas de rango máximo equivalente). Para esas consultas, puede calcular la respuesta en $O(1)$ tiempo.

2.5. Unión de conjuntos disjuntos (*Disjoint Set Union o DSU*)

Esta estructura de datos proporciona las siguientes capacidades. Se nos dan varios elementos, cada uno de los cuales es un conjunto separado. Una DSU tendrá una operación para combinar dos conjuntos y podrá decir en qué conjunto se encuentra un elemento específico. La versión clásica también introduce una tercera operación, puede crear un conjunto a partir de un nuevo elemento.

3. Desarrollo

Antes de responder a las consultas, debemos preprocesar el árbol. Hacemos un recorrido DFS comenzando en la raíz y construimos una lista que almacena el orden de los vértices que visitamos (se agrega un vértice a la lista cuando lo visitamos por primera vez, y después del retorno de los recorridos DFS a sus hijos)). Esto también se llama un recorrido de Euler del árbol. Está claro que el tamaño de esta lista será $O(N)$. También debemos crear un arreglo $first[0..N-1]$ que almacene para cada vértice i su primera aparición en euler. Es decir, la primera posición en euler tal que $euler[first[i]] = i$. También utilizando el DFS podemos encontrar la altura de cada nodo (distancia desde la raíz a ella) y almacenarla en el arreglo $height[0..N-1]$.

Entonces, ¿cómo podemos responder a las consultas utilizando el recorrido de Euler y los dos arreglos adicionales? Supongamos que la consulta es un par de $v1$ y $v2$. Considere los vértices que visitamos en el recorrido de Euler entre la primera visita de $v1$ y la primera visita de $v2$. Es fácil ver que el $LCA(v1, v2)$ es el vértice con la altura más baja en este camino. Ya notamos que el LCA tiene que ser parte de la ruta más corta entre $v1$ y $v2$. Claramente también tiene que ser el vértice con la altura más pequeña. Y en la gira de Euler, esencialmente utilizamos el camino más corto, excepto que además visitamos todos los subárboles que encontramos en el camino. Pero todos los vértices en estos subárboles son más bajos en el árbol que el LCA y, por lo tanto, tienen una altura mayor. Por lo tanto, el LCA ($v1$, $v2$) se puede determinar únicamente al encontrar el vértice con la altura más pequeña en el recorrido de Euler entre la primera ($v1$) y la primera ($v2$).

Vamos a ilustrar esta idea. Considere la anterior imagen y el recorrido de Euler con las alturas correspondientes:

Vertices:	1	2	5	2	6	2	1	3	1	4	7	4	1
Alturas:	1	2	3	2	3	2	1	2	1	2	3	2	1

En el recorrido que comienza en el vértice 6 y termina en 4, visitamos los vértices [6,2,1,3,1,4]. Entre esos vértices, el vértice 1 tiene la altura más baja, por lo tanto, $LCA(6, 4) = 1$.

Para responder a una consulta, solo necesitamos encontrar el vértice con la altura más pequeña en el array euler en el rango desde el primer [v1] al primero [v2]. Por lo tanto, el problema de LCA se reduce al problema de RMQ (encontrar el mínimo en un problema de rango).

3.1. LCA por elevación binaria

Para cada nodo precomputaremos su antepasado por encima de él, su antepasado dos nodos por encima, su antepasado cuatro por encima, etc. Almacenémoslos en la matriz, es decir, up es up[i][j] el $2^j - simo$ antepasado por encima del nodo i con $i = 1 \dots N, j = 0 \dots ceil(\log N)$. Podemos calcular esta matriz utilizando un recorrido DFS del árbol.

Para cada nodo, también recordaremos la hora de la primera visita de este nodo (es decir, la hora en que el DFS descubre el nodo) y la hora en que lo dejamos (es decir, después de que visitamos a todos los hijos y salimos de la función DFS). Podemos usar esta información para determinar en tiempo constante si un nodo es antepasado de otro nodo.

Supongamos ahora que recibimos una consulta (u, v). Podemos comprobar inmediatamente si un nodo es el antepasado del otro. En este caso este nodo ya es el LCA. Si u es el ancestro de v, v no es el ancestro de u, escalamos los ancestros de u hasta encontrar el nodo más alto (es decir, el más cercano a la raíz), que no es un ancestro de v (es decir, un nodo x, tal que x no es un ancestro de v, pero up[x][0] lo es). Podemos encontrar este nodo x en $O(\log N)$ tiempo usando la matriz up.

Describiremos este proceso con más detalle. Deja que L = ceil($\log N$). Supongamos primero que $i = L$. Si $up[u][i]$ no es antepasado de v, entonces podemos asignar $u = up[u][i]$ y decrementar i. Si $up[u][i]$ es un ancestro, entonces simplemente decrementamos i. Claramente, después de hacer esto para todos los no negativos i del nodo u no será el nodo deseado, es decir, u todavía no es un ancestro de v, pero $up[u][0]$ lo es.

Ahora, obviamente, la respuesta a LCA será $up[u][0]$, es decir, el nodo más pequeño entre los ancestros del nodo u, que también es un ancestro de v. Por lo tanto, responder a una consulta LCA iterará i de ceil($\log N$) a 0 y verificará en cada iteración si un nodo es el ancestro del otro.

3.2. Algoritmo de Tarjan's offline para LCA

El algoritmo lleva el nombre de Robert Tarjan, quien lo descubrió en 1979 y también hizo muchas otras contribuciones a la estructura de datos Disjoint Set Union , que se utilizará mucho en este algoritmo.

El algoritmo responde a todas las consultas con un recorrido DFS del árbol. Es decir, una consulta (u, v) se responde en el nodo u, si nodo v ya ha sido visitado anteriormente, o viceversa. Así que supongamos que estamos actualmente en el nodo v, ya hemos realizado llamadas DFS recursivas, y también visitamos el segundo nodo u de la consulta (u, v). Aprendamos cómo encontrar el LCA de estos dos nodos.

Tenga en cuenta que LCA(u, v) es el nodo v o uno de sus antepasados. Entonces necesitamos encontrar el nodo más bajo entre los ancestros de v (incluido v), para el cual el nodo u es descendiente. También tenga en cuenta que para un fijo v los nodos visitados del árbol se dividen en un conjunto de conjuntos disjuntos. cada antepasado p de nodo v tiene su propio conjunto que contiene este nodo y todos los subárboles con raíces en los de sus hijos que no son parte del camino desde v a la raíz del árbol. El conjunto que contiene el nodo u determina el LCA(u, v) : el LCA es el representante del conjunto, es decir, el nodo en se encuentra en el camino entre v y la raíz del árbol.

Solo necesitamos aprender a mantener de manera eficiente todos estos conjuntos. Para ello aplicamos la estructura de datos DSU. Para poder aplicar Unión por rango, almacenamos el representante real (el valor en el camino entre v y la raíz del árbol) de cada conjunto en la matriz ancestor.

Analicemos la implementación del DFS. Supongamos que actualmente estamos visitando el nodo v. Colocamos el nodo en un nuevo conjunto en la DSU, ancestor[v] = v. Como de costumbre, procesamos a todos los niños de v. Para esto primero debemos llamar recursivamente a DFS desde ese nodo, y luego agregar este nodo con todo su subárbol al conjunto de v . Esto se puede hacer con la función union_sets y la siguiente asignación ancestor[find_set(v)] = v(esto es necesario, porque union_sets podría cambiar el representante del conjunto).

Finalmente, después de procesar a todos los hijos, podemos responder todas las consultas del (u, v) para cual u ya ha sido visitado. La respuesta a la consulta, es decir, el LCA de u y v, será el nodo ancestor[find_set(u)]. Es fácil ver que una consulta solo se responderá una vez.

3.3. Algoritmo Farach-Colton y Bender para LCA

Usamos la reducción clásica del problema LCA al problema RMQ. Atravesamos todos los nodos del árbol con DFS y mantenemos una matriz con todos los nodos visitados y las alturas de estos nodos. El LCA de dos nodos u y v es el nodo entre las ocurrencias de u y v en el recorrido, que tiene la menor altura.

Tenga en cuenta que el problema de RMQ reducido es muy específico: dos elementos adyacentes cualquiera en la matriz difieren exactamente en uno (dado que los elementos de la matriz no son más que las alturas de los nodos visitados en orden de recorrido, y vamos a un descendiente , en cuyo caso el siguiente elemento es uno más grande, o volver al ancestro, en cuyo caso el siguiente elemento es uno más bajo). El algoritmo de Farach-Colton y Bender describe una solución exactamente para este problema especializado de RMQ.

Denotemos con A el arreglo en el que queremos realizar las consultas de rango mínimo. Y N será del tamaño de A.

Hay una estructura de datos fácil que podemos usar para resolver el problema de RMQ con $O(N \log N)$ preprocesamiento y $O(1)$ para cada consulta: la tabla dispersa. Creamos una tabla T donde cada elemento $T[i][j]$ es igual al mínimo de A en el intervalo $[i, i + 2^j - 1]$. Obviamente $0 \leq j \leq \lceil \log N \rceil$, y por lo tanto el tamaño de la Tabla Dispersa será $O(N \log N)$. Puedes construir la tabla fácilmente en $O(N \log N)$ al notar que $T[i][j] = \min(T[i][j-1], T[i + 2^{j-1}][j-1])$.

¿Cómo podemos responder una consulta RMQ en $O(1)$ utilizando esta estructura de datos? Que la consulta recibida sea $[l, r]$, entonces la respuesta es $\min(T[l][\text{sz}], T[r - 2^{\text{sz}} + 1][\text{sz}])$, dónde sz es el mayor exponente tal que 2^{sz} no es mayor que la longitud del rango $r - l + 1$. De hecho, podemos tomar el rango $[l, r]$ y cubrirlo dos rangos de longitud 2^{sz}-1 que comienza en l y el otro termina en r. Estos rangos se superponen, pero esto no interfiere con nuestro cálculo. Para lograr realmente la complejidad temporal de $O(1)$ por consulta, necesitamos saber los valores de sz para todas las longitudes posibles desde 1 a N. Pero esto se puede precalcular fácilmente.

Ahora queremos mejorar la complejidad del preprocesamiento hasta $O(N)$.

Dividimos el arreglo A en bloques de tamaño $K = 0{,}5 \log N$ con $\log$ siendo el logaritmo en base 2. Para cada bloque calculamos el elemento mínimo y los almacenamos en un arreglo B. B tiene el tamaño $\frac{N}{K}$. Construimos una tabla dispersa a partir del arreglo B. El tamaño y la complejidad temporal del mismo será:

$$\frac{N}{K} \log\left(\frac{N}{K}\right) = \frac{2N}{\log(N)} \log\left(\frac{2N}{\log(N)}\right) =$$

$$= \frac{2N}{\log(N)} \left(1 + \log\left(\frac{N}{\log(N)}\right)\right) \leq \frac{2N}{\log(N)} + 2N = O(N)$$

Ahora solo tenemos que aprender a responder rápidamente consultas de rango mínimo dentro de cada bloque. De hecho, si la consulta mínima del rango recibido es $[l, r]$ y l y r están en diferentes bloques, entonces la respuesta es el mínimo de los siguientes tres valores: el mínimo del sufijo de bloque de l a partir de l, el mínimo del prefijo de bloque de r terminando en r, y el mínimo de los bloques entre ellos. El mínimo de los bloques intermedios se puede responder en $O(1)$ utilizando la tabla dispersa. Así que esto nos deja solo el rango mínimo de consultas dentro de los bloques.

Aquí explotaremos la propiedad del arreglo. Recuerde que los valores en el arreglo, que son solo valores de altura en el árbol, siempre diferirán en uno. Si eliminamos el primer elemento de un bloque y lo restamos de todos los demás elementos del bloque, cada bloque se puede identificar por una secuencia de longitud $K-1$ que consiste en el número $+1$ y -1. Debido a que estos bloques son tan pequeños, solo pueden ocurrir unas pocas secuencias diferentes. El número de secuencias posibles es:

$$2^{K-1} = 2^{0{,}5 \log(N) - 1} = 0{,}5 \left(2^{\log(N)}\right)^{0{,}5} = 0{,}5\sqrt{N}$$

Por lo tanto, el número de bloques diferentes es $O(\sqrt{N})$, y por lo tanto podemos precalcular los resultados de las consultas mínimas de rango dentro de todos los bloques diferentes en $O(\sqrt{N}K^2) = O(\sqrt{N}\log^2(N)) = O(N)$ tiempo. Para la implementación podemos caracterizar un bloque por una máscara de bits de longitud $K - 1$ (que cabrá en un int estándar) y almacenará el índice del mínimo en una matriz block[mask][l][r] de tamaño $O(\sqrt{N}\log^2(N))$.

Así que aprendimos a precalcular consultas mínimas de rango dentro de cada bloque, así como consultas mínimas de rango sobre un rango de bloques, todo en $O(N)$. Con estos precálculos podemos responder cada consulta en $O(1)$, utilizando como máximo cuatro valores precalculados: el mínimo del bloque que contiene l, el mínimo del bloque que contiene ry los dos mínimos de los segmentos superpuestos de los bloques entre ellos.

4. Implementación

4.1. C++

4.1.1. LCA usando arbol de rango

```
struct LCA {
   vector<int> height, euler, first, rtree;
   vector<bool> visited;
   int n;

   LCA(vector<vector<int>> &adj, int root = 0) {
      n = adj.size(); height.resize(n);
      first.resize(n); euler.reserve(n * 2);
      visited.assign(n, false); dfs(adj, root);
      int m = euler.size(); rtree.resize(m * 4);
      build(1, 0, m - 1);
   }

   void dfs(vector<vector<int>> &adj, int node, int h = 0) {
      visited[node] = true; height[node] = h;
      first[node] = euler.size();
      euler.push_back(node);
      for (auto to : adj[node]) {
         if(!visited[to]) {
            dfs(adj, to, h + 1); euler.push_back(node);
         }
      }
   }

   void build(int node, int b, int e) {
      if(b==e){ segtree[node] = euler[b];}
      else {
         int mid = (b + e) / 2;
         build(node << 1, b, mid);
         build(node << 1 | 1, mid + 1, e);
         int l = rtree[node << 1], r = rtree[node << 1 | 1];
         rtree[node] = (height[l] < height[r]) ? l : r;
      }
   }

   int query(int node, int b, int e, int L, int R) {
```

```
        if (b > R || e < L) return -1;
        if (b >= L && e <= R) return rtree[node];
        int mid = (b + e) >> 1;

        int left = query(node << 1, b, mid, L, R);
        int right = query(node << 1 | 1, mid + 1, e, L, R);
        if (left == -1) return right;
        if (right == -1) return left;
        return height[left] < height[right] ? left : right;
    }

    int lca(int u, int v) {
        int left = first[u], right = first[v],tmp;
        if (left > right){ tmp =left;left=right;right=tmp};
        return query(1, 0, euler.size() - 1, left, right);
    }
};
```

4.1.2. LCA usando elevación binaria

```
int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p){
    tin[v] = ++timer; up[v][0] = p;
    for (int i = 1; i <= l; ++i)
        up[v][i] = up[up[v][i-1]][i-1];
    for (int u : adj[v]) {
        if (u != p) dfs(u, v);
    }
    tout[v] = ++timer;
}

bool is_ancestor(int u, int v){
    return tin[u] <= tin[v] && tout[u] >= tout[v];
}

int lca(int u, int v) {
    if (is_ancestor(u, v)) return u;
    if (is_ancestor(v, u)) return v;
    for (int i = l; i >= 0; --i) {
        if (!is_ancestor(up[u][i], v)) u = up[u][i];
    }
    return up[u][0];
```

```
}

void preprocess(int root) {
   tin.resize(n); tout.resize(n);
   timer = 0; l = ceil(log2(n));
   up.assign(n, vector<int>(l + 1));
   dfs(root, root);
}
```

4.1.3. LCA usando algoritmo Tarjan offline

No se ha incluido la implementación de DSU, ya que se puede utilizar la clásica sin modificaciones.

```
vector<vector<int>> adj, queries;
vector<int> ancestor;
vector<bool> visited;

void dfs(int v){
   visited[v] = true; ancestor[v] = v;
   for (int u : adj[v]) {
      if (!visited[u]) {
         dfs(u); union_sets(v, u);
         ancestor[find_set(v)] = v;
      }
   }
   for (int other_node : queries[v]) {
      if (visited[other_node])
         cout << "LCA de " << v << " y " << other_node
         << " es " << ancestor[find_set(other_node)] << ".\n";
   }
}

void compute_LCAs() {
   // initialize n, adj and DSU
   // for (each query (u, v)) {
   //    queries[u].push_back(v);
   //    queries[v].push_back(u);
   // }
   ancestor.resize(n); visited.assign(n, false); dfs(0);
}
```

4.1.4. LCA usando algoritmo Farach-Colton y Bender

```
int n;

vector<vector<int>> adj;
```

```
int block_size, block_cnt;
vector<int> first_visit;
vector<int> euler_tour;
vector<int> height;
vector<int> log_2;
vector<vector<int>> st;
vector<vector<vector<int>>> blocks;
vector<int> block_mask;

void dfs(int v, int p, int h) {
   first_visit[v] = euler_tour.size(); euler_tour.push_back(v);
   height[v] = h;
   for (int u : adj[v]) {
      if (u == p) continue;
      dfs(u, v, h + 1);
      euler_tour.push_back(v);
   }
}

int min_by_h(int i, int j) {
   return height[euler_tour[i]] < height[euler_tour[j]] ? i : j;
}

void precompute_lca(int root) {
   // obtener el recorrido de euler y
   // los indices de las primeras ocurrencias
   first_visit.assign(n, -1); height.assign(n, 0);
   euler_tour.reserve(2 * n); dfs(root, -1, 0);

   // precalcular todos los valores de log
   int m = euler_tour.size(); log_2.reserve(m + 1);
   log_2.push_back(-1);
   for (int i = 1; i <= m; i++)
      log_2.push_back(log_2[i / 2] + 1);

   block_size = max(1, log_2[m] / 2);
   block_cnt = (m + block_size - 1) / block_size;

   // precalcula el minimo de cada bloque y
   // construya una tabla dispersa
   st.assign(block_cnt, vector<int>(log_2[block_cnt] + 1));
   for (int i = 0, j = 0, b = 0; i < m; i++, j++) {
      if (j == block_size) j = 0, b++;
      if (j == 0 || min_by_h(i, st[b][0]) == i) st[b][0] = i;
   }
   for (int l = 1; l <= log_2[block_cnt]; l++) {
      for (int i = 0; i < block_cnt; i++) {
         int ni = i + (1 << (l - 1));
         if (ni >= block_cnt) st[i][l] = st[i][l-1];
```

```
            else st[i][l]=min_by_h(st[i][l-1],st[ni][l-1]);
        }
    }
    // mascara de calculo previo para cada bloque
    block_mask.assign(block_cnt, 0);
    for (int i = 0, j = 0, b = 0; i < m; i++, j++) {
        if (j == block_size) j = 0, b++;
        if (j>0 && (i>=m || min_by_h(i-1,i)==i-1))block_mask[b]+=1<<(j-1);
    }
    // precalcular RMQ para cada bloque unico
    int possibilities = 1 << (block_size - 1);
    blocks.resize(possibilities);
    for (int b = 0; b < block_cnt; b++) {
        int mask = block_mask[b];
        if (!blocks[mask].empty()) continue;
        blocks[mask].assign(block_size, vector<int>(block_size));
        for (int l = 0; l < block_size; l++) {
            blocks[mask][l][l] = l;
            for (int r = l + 1; r < block_size; r++) {
                blocks[mask][l][r] = blocks[mask][l][r - 1];
                if (b * block_size + r < m)
                    blocks[mask][l][r]=min_by_h(b*block_size + blocks[mask][l][r],
                                                b*block_size +r)-b*block_size;
            }
        }
    }
}

int lca_in_block(int b, int l, int r) {
    return blocks[block_mask[b]][l][r] + b * block_size;
}

int lca(int v, int u) {
    int l = first_visit[v]; int r = first_visit[u];
    if (l > r){int tmp = l;l=r;r=tmp;}
    int bl = l / block_size;
    int br = r / block_size;
    if(bl==br)return euler_tour[lca_in_block(bl,l%block_size,r%block_size)];
    int ans1 = lca_in_block(bl,l%block_size,block_size-1);
    int ans2 = lca_in_block(br,0,r%block_size);
    int ans = min_by_h(ans1, ans2);
    if (bl + 1 < br) {
      int l = log_2[br - bl - 1]; int ans3 = st[bl+1][l];
      int ans4 = st[br - (1 << l)][l];
      ans = min_by_h(ans, min_by_h(ans3, ans4));
    }
    return euler_tour[ans];
}
```

4.2. Java

4.2.1. LCA usando arbol de rango

```
private class LCA {
   private int[] depth, dfs_order,first,minPos;
   private int cnt,n;

   private void dfs(List<Integer>[] tree, int u, int d) {
      depth[u] = d; dfs_order[cnt++] = u;
      for (int v : tree[u])
         if (depth[v] == -1) {
            dfs(tree, v, d + 1); dfs_order[cnt++] = u;
      }
    }

   private void buildTree(int node, int left, int right) {
      if(left==right){ minPos[node]=dfs_order[left]; return;}
      int mid = (left + right) >> 1;
      buildTree(2 * node + 1, left, mid);
      buildTree(2 * node + 2, mid + 1, right);
      minPos[node]=depth[minPos[2*node +1]]<depth[minPos[2*node+2]]? minPos[2*
          node+1]:minPos[2*node+2];
   }

   public LCA(List<Integer>[] tree, int root) {
      int nodes = tree.length; depth = new int[nodes];
      Arrays.fill(depth, -1);
      n = 2 * nodes - 1; dfs_order = new int[n];
      cnt = 0; dfs(tree, root, 0);
      minPos = new int[4 * n];
      buildTree(0, 0, n - 1);
      first = new int[nodes]; Arrays.fill(first, -1);
      for (int i = 0; i < dfs_order.length; i++)
         if (first[dfs_order[i]] == -1) first[dfs_order[i]] = i;
   }

   public int lca(int a, int b) {
      return minPos(Math.min(first[a],first[b]),Math.max(first[a],first[b])
          ,0,0,n-1);
   }

   private int minPos(int a, int b, int node, int left, int right) {
      if(a == left && right == b) return minPos[node];
      int mid = (left + right) >> 1;
      if (a <= mid && b > mid){
         int p1 = minPos(a,Math.min(b,mid),2*node+1,left,mid);
         int p2 = minPos(Math.max(a,mid+1),b,2*node+2,mid+1,right);
         return depth[p1] < depth[p2] ? p1 : p2;
      } else if (a <= mid) {
```

```
            return minPos(a,Math.min(b,mid),2*node+1,left,mid);
        } else if (b > mid) {
            return minPos(Math.max(a,mid+1),b,2*node+2,mid+1,right);
        } else { throw new RuntimeException(); }
    }
}
```

5. Complejidad

Usando *Sqrt-Descomposición*, es posible obtener una solución respondiendo a cada consulta en O($\sqrt{N}$) con preprocesamiento en tiempo O(N).

Usando un Árbol de Rango (*Range Tree*), puede responder a cada consulta en $O(\log N)$ con preprocesamiento en tiempo $O(N)$.

Usando elevación binaria necesitará $O(N \log N)$ para preprocesar el árbol, y luego $O(\log N)$ para cada consulta LCA.

En el caso del algoritmo Tarjan offline la complejidad temporal de este algoritmo. en primer lugar tenemos $O(n)$ debido a la DFS. En segundo lugar, tenemos las llamadas a funciones union_sets que suceden n veces, resultando también en $O(n)$. Y en tercer lugar tenemos las llamadas de find_set para cada consulta, lo que da $O(m)$. Entonces, en total, la complejidad del tiempo es $O(n + m)$, lo que significa que para valores suficientemente grandes m esto corresponde a $O(1)$ por responder una consulta.

El algoritmo de Farach-Colton y Bender es capaz de resolver las consultas mínimas de rango dado en $O(1)$ tiempo, sin dejar de tomar $O(N)$ tiempo de preprocesamiento.

6. Aplicaciones

El cómputo del ancestro común más bajo puede ser útil, por ejemplo, como parte de un procedimiento para determinar la distancia entre pares de nodos en un árbol: la distancia de v a w puede ser calculada como la distancia desde la raíz hasta v, sumada con la distancia desde la raíz hasta w, menos dos veces la distancia desde la raíz hasta su ancestro común más bajo.

7. Ejercicios propuestos

A continuación una lista de ejercicios que se puede resolver aplicamdo este algoritmo:

- DMOJ - Política Vacuna
- DMOJ - Los barcos más anchos
- SPOJ - LCA - Lowest Common Ancestor
- SPOJ - DISQUERY - Distance Query

- SPOJ - LCASQ - Lowest Common Ancestor
- TIMUS - 1471. Distance in the Tree