








Grafos II : LCA

Contenido

1. Conceptos básicos	
2. Solución trivial	
3. Binary Lifting	
4. RMQ	
5. Aplicación	

Contenido

1. Conceptos básicos	
2. Solución trivial	
3. Binary Lifting	
4. RMQ	
5. Aplicación	

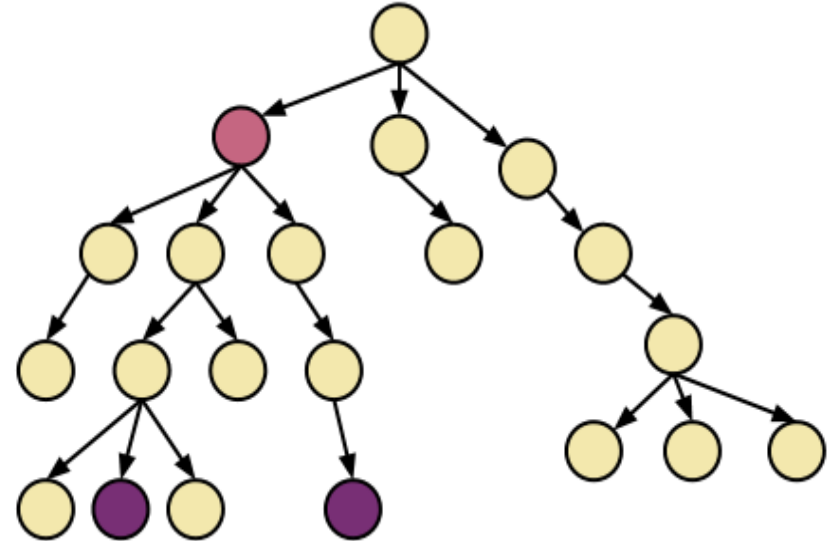
Conceptos básicos de árboles

Dado un árbol $G(V,E)$ con raíz $root$, definimos los siguientes términos:

- ❑ **Ancestro:** Los ancestros de un nodo u son todos los nodos que están en el camino que une $root$ y u
- ❑ **Descendientes:** Los descendientes de un nodo u son todos los nodos x tal que u es ancestro de x



Nota: Cualquier nodo es ancestro y descendiente de sí mismo

- ❑ **Subárbol:** Un subárbol de G está conformado por algún nodo u de G y todos sus descendientes.
- ❑ **Lowest common ancestor:** Se define al lowest common ancestor (lca) de dos nodos u, v , como el ancestro común de u y v que esté más alejado de la raíz; es decir, que esté a una profundidad mayor. Se denotará como $lca(u, v)$.



Fuente : <http://stoimen.com/2012/08/24/computer-algorithms-finding-the-lowest-common-ancestor/>

Contenido

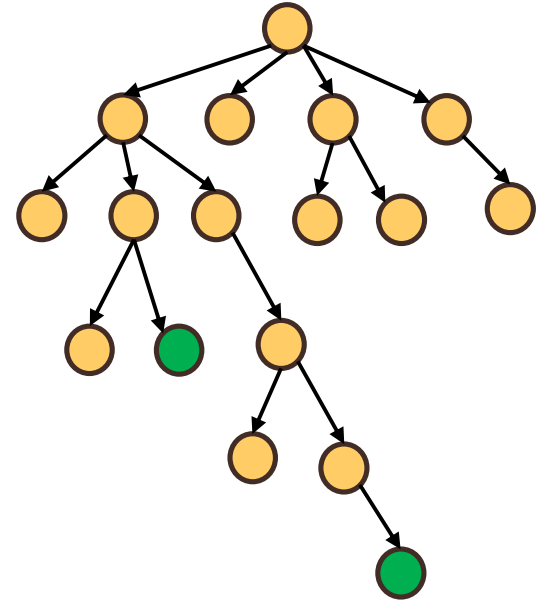
1. Conceptos básicos	
2. Solución trivial	
3. Binary Lifting	
4. RMQ	
5. Aplicación	

Solución Trivial

Para hallar el $lca(u, v)$, podemos usar el siguiente algoritmo:

1. Si u y v están en distintos niveles (con distinta profundidad), igualo los niveles haciendo que el nodo que se encuentra a una mayor profundidad vaya “subiendo” a través de sus padres ($u = \text{parent}[u]$).
2. Como u y v están en el mismo nivel, empiezo a “subirlos” a la vez hasta que ambos nodos sean iguales. Una vez sean iguales, el lca será ese nodo al que llegaron.

Como el *lca* siempre existe (porque la raíz es ancestro de cualquier nodo), el algoritmo siempre termina.

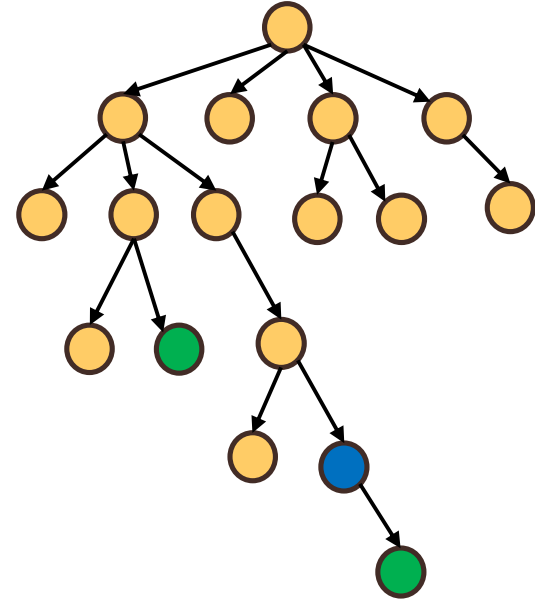


Solución Trivial

Para hallar el $lca(u, v)$, podemos usar el siguiente algoritmo:

1. Si u y v están en distintos niveles (con distinta profundidad), igualo los niveles haciendo que el nodo que se encuentra a una mayor profundidad vaya “subiendo” a través de sus padres ($u = parent[u]$)..
2. Como u y v están en el mismo nivel, empiezo a “subirlos” a la vez hasta que ambos nodos sean iguales. Una vez sean iguales, el lca será ese nodo al que llegaron.

Como el lca siempre existe (porque la raíz es ancestro de cualquier nodo), el algoritmo siempre termina.

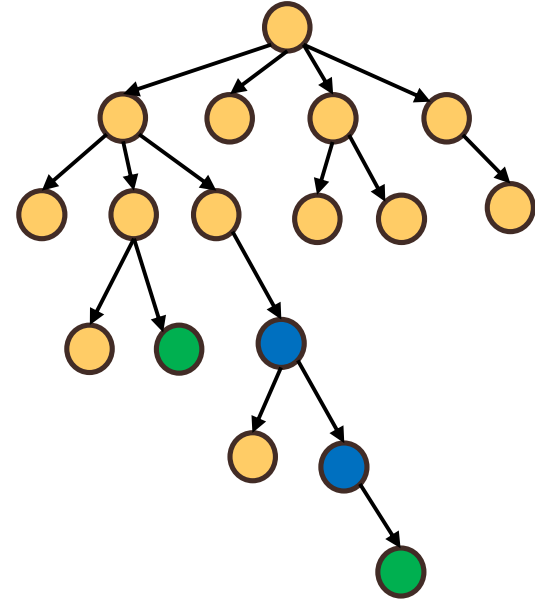


Solución Trivial

Para hallar el $lca(u, v)$, podemos usar el siguiente algoritmo:

1. Si u y v están en distintos niveles (con distinta profundidad), igualo los niveles haciendo que el nodo que se encuentra a una mayor profundidad vaya “subiendo” a través de sus padres ($u = \text{parent}[u]$).
2. Como u y v están en el mismo nivel, empiezo a “subirlos” a la vez hasta que ambos nodos sean iguales. Una vez sean iguales, el lca será ese nodo al que llegaron.

Como el lca siempre existe (porque la raíz es ancestro de cualquier nodo), el algoritmo siempre termina.

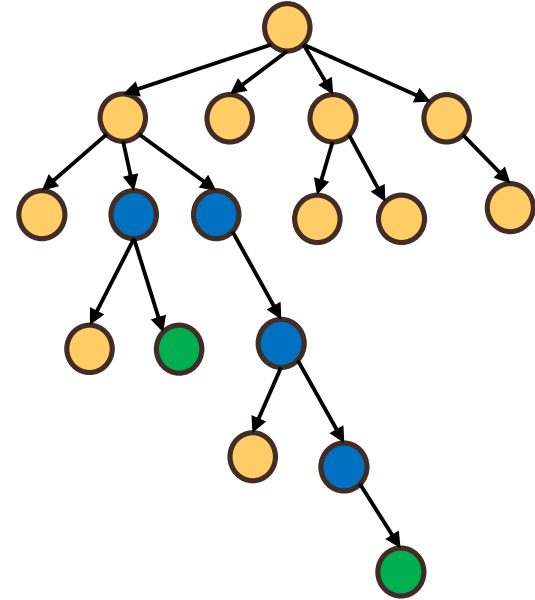


Solución Trivial

Para hallar el $lca(u, v)$, podemos usar el siguiente algoritmo:

1. Si u y v están en distintos niveles (con distinta profundidad), igualo los niveles haciendo que el nodo que se encuentra a una mayor profundidad vaya “subiendo” a través de sus padres ($u = \text{parent}[u]$).
2. **Como u y v están en el mismo nivel, empiezo a “subirlos” a la vez hasta que ambos nodos sean iguales. Una vez sean iguales, el lca será ese nodo al que llegaron.**

Como el lca siempre existe (porque la raíz es ancestro de cualquier nodo), el algoritmo siempre termina.



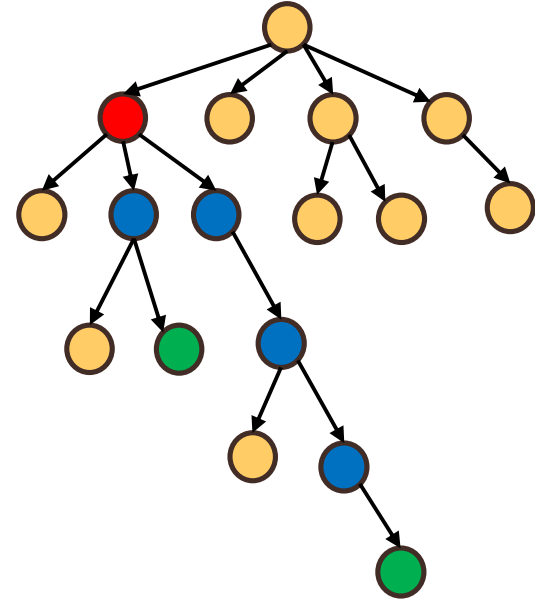
Solución Trivial

Para hallar el $lca(u, v)$, podemos usar el siguiente algoritmo:

1. Si u y v están en distintos niveles (con distinta profundidad), igualo los niveles haciendo que el nodo que se encuentra a una mayor profundidad vaya “subiendo” a través de sus padres ($u = parent[u]$).
2. **Como u y v están en el mismo nivel, empiezo a “subirlos” a la vez hasta que ambos nodos sean iguales. Una vez sean iguales, el lca será ese nodo al que llegaron.**

Como el lca siempre existe (porque la raíz es ancestro de cualquier nodo), el algoritmo siempre termina.

La complejidad es $O(n)$ por query. Pero también se necesita un precálculo de $O(n)$ para correr un dfs que halle los padres y las profundidades.



Código – Solución Trivial

```
vector<int> adj[MX];
int parent[MX];
int depth[MX];

void addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

void dfs(int u) {
    for (int v : adj[u]) {
        if (v != parent[u]) {
            parent[v] = u;
            depth[v] = depth[u] + 1;
            dfs(v);
        }
    }
}
```

```
void precalculate(int root = 0) { //O(n)
    parent[root] = -1;
    depth[root] = 0;
    dfs(root);
}

int lca(int u, int v) { //O(n)
    if (depth[u] < depth[v]) {
        swap(u, v);
    }
    while (depth[u] > depth[v]) {
        u = parent[u];
    }
    while (u != v) {
        u = parent[u];
        v = parent[v];
    }
    return u;
}
```

Código — Solución Trivial

En el algoritmo anterior en el 2do paso tuvimos que subir ambos nodos a la vez. Pero es posible hacer que solo uno de los nodos itere, de la siguiente forma:

$lca(u, v)$

1. *while* (u is not ancestor of v)
 $u := parent[u]$
2. *return* u

Sin embargo, para poder hacer esto y mantener la misma complejidad, necesitamos una función que nos diga en $O(1)$ si el nodo u es ancestro de v . ¿Cómo podemos hacer eso?

Función isAncestor

Definamos los siguientes términos:

- **Tiempo de entrada:** Es el momento en el que el nodo entra por primera vez en el dfs. Denotemos el tiempo de entrada de un nodo u como $tIn[u]$
- **Tiempo de salida:** Es el momento donde al nodo le devuelven la llamada recursiva y sale del dfs. Denotemos el tiempo de salida de un nodo u como $tOut[u]$

Este “tiempo” no está medido en segundos sino que es simplemente un contador.

Propiedad: Un nodo u es ancestro de un nodo v si se cumple que

$$tIn[u] \leq tIn[v] \ \&\& \ tOut[u] \geq tOut[v]$$

Código – Solución Trivial

```
vector<int> adj[MX];
int parent[MX];
int depth[MX];
int tIn[MX];
int tOut[MX];
int timer;

void addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}






void dfs(int u) {
    tIn[u] = timer++;
    for (int v : adj[u]) {
        if (v != parent[u]) {
            parent[v] = u;
            depth[v] = depth[u] + 1;
            dfs(v);
        }
    }
    tOut[u] = timer++;
}
```

```
void precalculate(int root = 0) { //O(n)
    parent[root] = -1;
    depth[root] = 0;
    timer = 0;
    dfs(root);
}

bool isAncestor(int u, int v) { //is u ancestor of v ?
    return tIn[u] <= tIn[v] && tOut[u] >= tOut[v];
}

int lca(int u, int v) { //O(n)
    /*if (depth[u] > depth[v]) {
        swap(u, v);
    }*/
    while (!isAncestor(u, v)) {
        u = parent[u];
    }
    return u;
}
```

Contenido

1. Conceptos básicos	
2. Solución trivial	
3. Binary Lifting	
4. RMQ	
5. Aplicación	

Binary Lifting

Para reducir la complejidad de una query de lca podemos usar el hecho de que cualquier número entero positivo puede ser expresado como una suma de **potencias de 2**.

En vez de ir avanzando uno por uno (padre por padre) para encontrar el lca, podemos dar saltos de potencias de 2.

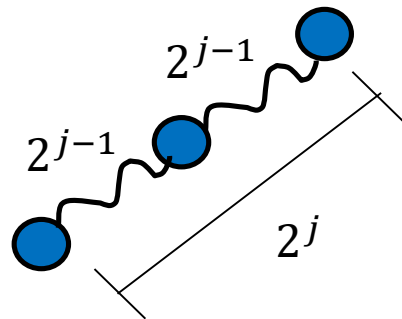
Para poder lograr esto, primero debemos hacer un precálculo de los ancestros de cada nodo que están a una distancia igual a una potencia de 2.

Definamos a la matriz $anc[i][j]$ como el ancestro del nodo i a una distancia 2^j , $\forall i \in V, 2^j \leq depth[i]$

$anc[i][0] = parent[i]$

$anc[i][j] = anc[anc[i][j-1]][j-1]$, $\forall j > 0, 2^j \leq depth[i]$

Complejidad $O(n \log n)$



Binary Lifting

Una vez hecho el precálculo, podemos responder las consultas $lca(u, v)$. Podemos seguir el siguiente procedimiento:

1. **Caso 1:** u es ancestro de v . Entonces la respuesta es u
2. **Caso 2:** u no es ancestro de v

En cada iteración trataremos de hacer que el nodo u dar el salto potencia de 2 más grande **sin pasarnos** del lca . ¿Cómo lo hacemos?

Podemos empezar desde la potencia de 2 más grande permitida e ir disminuyendo. Si al hacer el salto con esa potencia, voy a un nodo que **NO** es ancestro de v , entonces podemos saltar y estar seguros que no nos paremos. Al finalizar el bucle, no llegaremos al lca pero llegaremos a su hijo.

Como los saltos son potencia de 2, la complejidad sería $O(\log n)$ por query

Binary Lifting

Nota: $32 - \text{__builtin_clz}(n) = \lceil \log n \rceil$

```
vector<int> adj[MX];
int depth[MX];
int tIn[MX];
int tOut[MX];
int timer;
int anc[MX][32 - __builtin_clz(MX)];

void addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}
```

```
void dfs(int u) {
    tIn[u] = timer++;
    for (int v : adj[u]) {
        if (v != anc[u][0]) {
            anc[v][0] = u;
            depth[v] = depth[u] + 1;
            dfs(v);
        }
    }
    tOut[u] = timer++;
}






void precalculate(int n, int root = 0) { //O(n log n)
    anc[root][0] = -1;
    depth[root] = 0;
    timer = 0;
    dfs(root);
    for (int j = 1; (1 << j) < n; j++) {
        for (int i = 0; i < n; i++) {
            if (anc[i][j-1] != -1) {
                anc[i][j] = anc[anc[i][j-1]][j-1];
            } else {
                anc[i][j] = -1;
            }
        }
    }
}
```

Binary Lifting

```
bool isAncestor(int u, int v){ //is u ancestor of v ?
    return tIn[u] <= tIn[v] && tOut[u] >= tOut[v];
}

int lca(int u, int v) { //O(log n)
    /*if (depth[u] > depth[v]) {
        swap(u , v);
    }*/
    if (isAncestor(u , v)) {
        return u;
    }
    int bits = 31 - __builtin_clz(depth[u]);
    for (int i = bits; i >= 0; i--) {
        if (anc[u][i] != -1 && !isAncestor(anc[u][i] , v)) {
            u = anc[u][i];
        }
    }
    return anc[u][0];
}
```

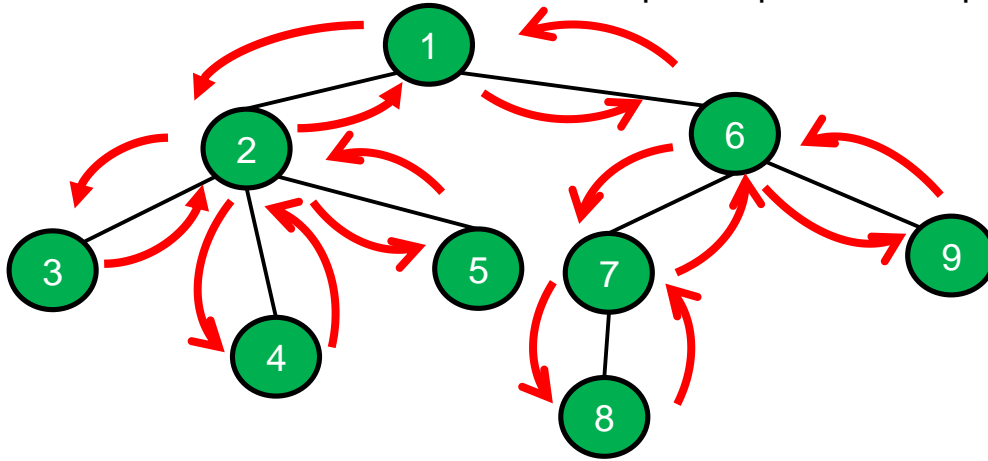
Contenido

1. Conceptos básicos	
2. Solución trivial	
3. Binary Lifting	
4. RMQ	
5. Aplicación	

Euler Tour de un árbol

El **Euler tour** de un rooted tree es la secuencia de los nodos visitados al recorrer el árbol en el orden del **dfs**. Se considerarán a las aristas como bidireccionales por lo que un nodo puede aparecer más de 1 vez.

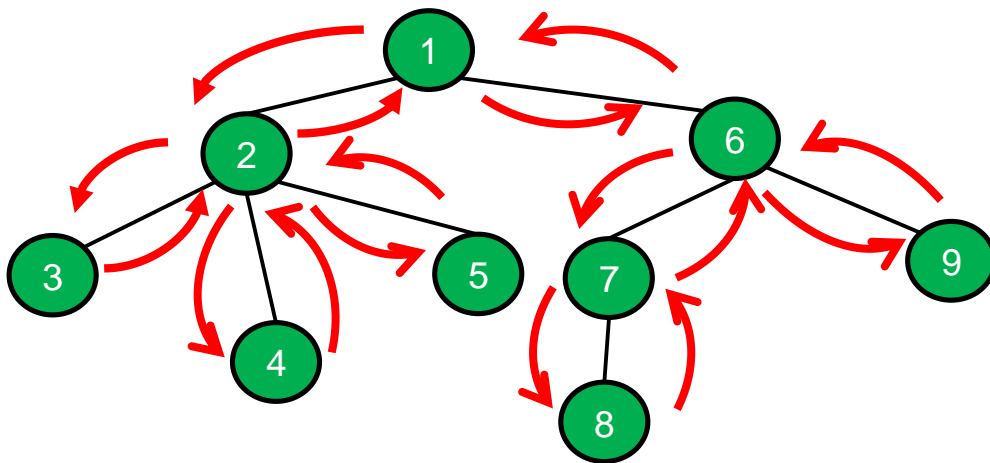
Ejemplo:



Orden	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Euler Tour	1	2	3	2	4	2	5	2	1	6	7	8	7	6	9	6	1

Euler Tour de un árbol

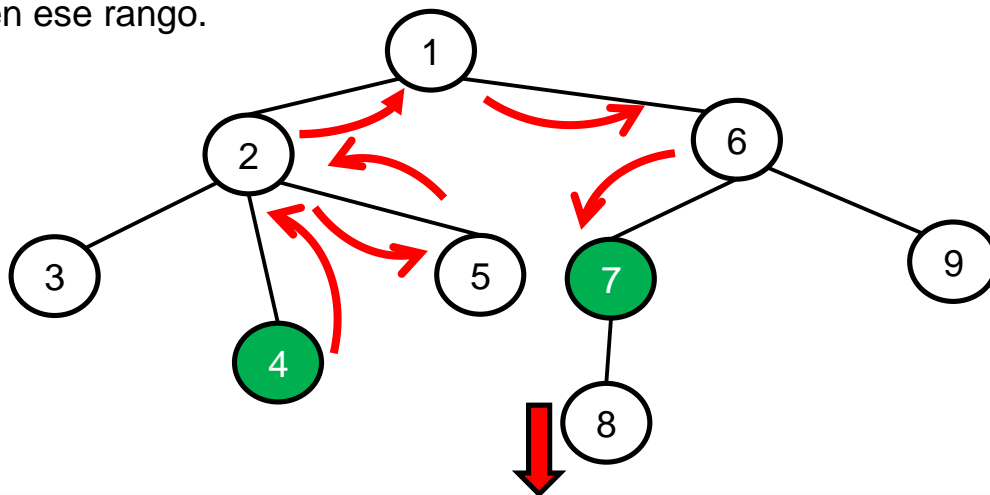
Observación 1 : En la secuencia del Euler tour, cada nodo u aparecerá $degree[u]$ veces. Excepto la raíz que aparecerá $degree[u] + 1$ veces. Es decir cada arista contribuirá con 2 nodos en el Euler tour y habrá 1 aparición extra de la raíz, dando un total de $2(n - 1) + 1 = 2n - 1$ elementos.



Orden	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Euler Tour	1	2	3	2	4	2	5	2	1	6	7	8	7	6	9	6	1

Euler Tour de un árbol

Observación 2 : Denotemos la primera aparición de un nodo u en el Euler tour como $first[u]$. Si se tiene un par de nodos (u, v) , entonces el $lca(u, v)$ se ubica entre $first[u]$ y $first[v]$ y es el nodo con menor profundidad en ese rango.



Orden	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Euler Tour	1	2	3	2	4	2	5	2	1	6	7	8	7	6	9	6	1
Profundidad	0	1	2	1	2	1	2	1	0	1	2	3	2	1	2	1	0

LCA \rightarrow Range Minimum Query

Hemos transformado el problema de hallar el lca en el problema del RMQ

El problema del RMQ es el siguiente :

Dado un arreglo $A[]$ de n elementos en donde está definida la operación \leq para cualquier par de elementos, se pide:

- $query(l, r)$: Devolver el menor elemento en el rango $[l, r]$

¿Cómo podríamos usar esto para calcular el LCA? Podemos notar que lo que se quiere es encontrar el nodo con menor profundidad en un rango



Orden	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Euler Tour	1	2	3	2	4	2	5	2	1	6	7	8	7	6	9	6	1
Profundidad	0	1	2	1	2	1	2	1	0	1	2	3	2	1	2	1	0

LCA -> Range Minimum Query



Orden	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Euler Tour	1	2	3	2	4	2	5	2	1	6	7	8	7	6	9	6	1
Profundidad	0	1	2	1	2	1	2	1	0	1	2	3	2	1	2	1	0

Podemos transformar esos 2 arreglos (Euler tour y profundidad) en un solo arreglo de parejas, y podemos usar el operador \leq de las parejas (comparar primero la primera componente, en caso sean iguales comparar la segunda)



Orden	0	1	2	3	4	5	6	7	8	9	10	11	12	...
<Euler, Prof>	1, 0	2, 1	3, 2	2, 1	4, 2	2, 1	5, 2	2, 1	1, 0	6, 1	7, 2	8, 3	7, 2	...

LCA -> Range Minimum Query

Nosotros ya hemos resuelto el RMQ con las estructuras de **segment tree** y **sparse table**. Comparemos ambos y elijamos el mejor:

Data Structure	Pre-procesamiento	Query	Update
Segment Tree	$O(n)$	$O(\log n)$	$O(\log n)$
Sparse Table	$O(n \log n)$	$O(1)$	No hay

Considerando que el árbol es estático, entonces no serán necesarios updates.

Lo más crítico suelen ser las queries porque puede haber un número muy grande de estas, así que en la mayoría de los casos lo más eficiente sería usar un **Sparse Table** para obtener el *lca* en tiempo constante.

Cabe resaltar que también existe un algoritmo con preprocesamiento $O(n)$ y query en $O(1)$, pero es más difícil de implementar y no suele ser tan crítico para las competencias.

LCA -> Range Minimum Query

Advertencia: Al implementarlo, no olvide que el tamaño del Euler tour será $2n - 1$, por lo que conviene multiplicar la memoria x2.

```
const int MX = 1e5;
const int loga = 32 - __builtin_clz(MX);

struct SparseTable{
    pair<int ,int> st[2 * MX][loga + 1]; //<min depth , node>

    pair<int,int> f(pair<int,int> a, pair<int,int> b){
        return min(a , b);
    }

    void build(vector<pair<int,int>> &A){ // O(n log n)
        //...
    }

    int query(int L, int R){ //O(1)
        int T = R - L + 1;
        int lg = 31 - (__builtin_clz(T));
        return f(st[L][lg], st[R- (1LL << lg) + 1][lg]).second;
    }
}st;
```

LCA -> Range Minimum Query

```
vector<int> adj[MX];
vector<pair<int,int>> euler; //{depth, node}
int first[MX];
int depth[MX];
int tIn[MX];
int tOut[MX];
int timer;






void addEdge(int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

void dfs(int u, int p = -1) { //O(n)
    tIn[u] = timer++;
    first[u] = euler.size();
    euler.push_back({depth[u], u});
    for (int v : adj[u]) {
        if (v != p) {
            depth[v] = depth[u] + 1;
            dfs(v, u);
            euler.push_back({depth[u], u});
        }
    }
    tOut[u] = timer++;
}
```

```
void precalculate(int root = 0) { //O(n logn)
    depth[root] = 0;
    timer = 0;
    euler.clear();
    dfs(root);
    st.build(euler);
}

int lca(int u, int v) { //O(1)
    int l = min(first[u], first[v]);
    int r = max(first[u], first[v]);
    return st.query(l, r);
}
```

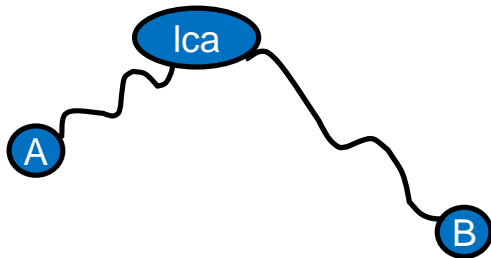
Contenido

1. Conceptos básicos	
2. Solución trivial	
3. Binary Lifting	
4. RMQ	
5. Aplicación	

On Path

Uno de los usos del *lca* puede ser para responder queries preguntando si un nodo *C* está en el camino que une a los nodos *A*, *B*

onPath(A, B, C) : Retorna **true** si *C* está en el camino *AB*, caso contrario retorna **false**



Para responder la query, podemos preguntarnos si *C* está en entre *A* y el *lca* o si *C* está entre *B* y el *lca*. Lo cual podemos responderlo usando la función **isAncestor**.

```
bool onPath(int A, int B, int C) { //is C on AB path ?
    int x = lca(A, B);
    if(isAncestor(x, C) && isAncestor(C, A)) {
        return true;
    }
    if(isAncestor(x, C) && isAncestor(C, B)) {
        return true;
    }
    return false;
}
```

Query en Path

Otro uso común del *lca* es de hallar una query en un path, normalmente que depende del **peso de las aristas** del path.

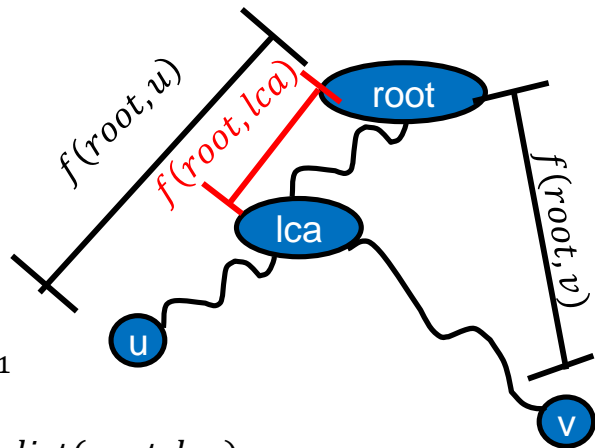
Supongamos que tienes pesos en las aristas del árbol y te interesa responder queries preguntando por un **operador o función asociativa y conmutativa (con inversa definida)** en el path entre u y v . Por ejemplo la suma/multiplicación/xor de los pesos en el camino de u , v . En el caso de la suma, estaríamos hallando la distancia entre dos nodos del árbol.

Para lograr esto vamos a hacer una especie de prefix sum en el árbol.

Para una mejor visualización, supongamos que tenemos el diagrama de la derecha y el operador ★ y definamos a $f(u, v)$ como el operador ★ aplicado en el camino entre u y v .

$$\Rightarrow f(u, v) = f(\text{root}, u) \star f(\text{root}, v) \star (f(\text{root}, \text{lca}))^{-1} \star (f(\text{root}, \text{lca}))^{-1}$$

Por ejemplo para la suma sería: $\text{dist}(u, v) = \text{dist}(\text{root}, u) + \text{dist}(\text{root}, v) - 2 \text{dist}(\text{root}, \text{lca})$



Query en Path – Binary Lifting

Con la técnica de **binary lifting**, es posible guardar información adicional aparte de los ancestros.

Supongamos que tienes pesos en las aristas del árbol y te interesa responder queries preguntando por una **función asociativa y conmutativa (no necesariamente con inversa definida)** en el path entre u y v . Por ejemplo el mínimo/máximo/gcd de los pesos en el camino de u , v .

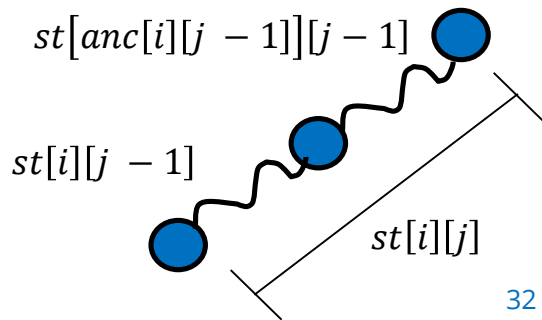
Entonces podemos hacer un precálculo similar a un sparse table.

Definamos a la matriz $st[i][j]$ como el resultado de aplicar la función en todo el rango de aristas que hay entre el nodo i y el ancestro $anc[i][j]$.

$st[i][0] = \text{arista entre } i \text{ y su padre}$

$st[i][j] = f(st[i][j-1], st[anc[i][j-1]][j-1])$, $\forall j > 0, 2^j \leq depth[i]$

Complejidad $O(n \log n)$



Query en Path — Binary Lifting

```
vector<pair<int,int>> adj[MX]; //node , weight
int depth[MX];
int tIn[MX];
int tOut[MX];
int timer;
int anc[MX][32 - __builtin_clz(MX)];
int st[MX][32 - __builtin_clz(MX)];
int neutral = INF;
```

```
void addEdge(int u, int v, int w) {
    adj[u].push_back({v, w});
    adj[v].push_back({u, w});
}
```

```
void dfs(int u) {
    tIn[u] = timer++;
    for (auto endpoint : adj[u]) {
        int v = endpoint.first;
        int w = endpoint.second;
        if (v != anc[u][0]) {
            anc[v][0] = u;
            depth[v] = depth[u] + 1;
            st[v][0] = w;
            dfs(v);
        }
    }
    tOut[u] = timer++;
}
```

```
int f(int u, int v) {
    return min(u, v);
}
```

```
void precalculate(int n, int root = 0) { //O(n log n)
    anc[root][0] = -1;
    depth[root] = 0;
    timer = 0;
    dfs(root);
    for (int j = 1; (1 << j) < n; j++) {
        for (int i = 0; i < n; i++) {
            if (anc[i][j - 1] != -1) {
                anc[i][j] = anc[anc[i][j - 1]][j - 1];
                st[i][j] = f(st[i][j - 1], st[anc[i][j - 1]][j - 1]);
            } else {
                anc[i][j] = -1;
            }
        }
    }
}
```

Query en Path – Binary Lifting

```
bool isAncestor(int u, int v){ //is u ancestor of v ?
    return tIn[u] <= tIn[v] && tOut[u] >= tOut[v];
}

int lift(int u, int v) { //O(log n)
    int ans = neutral;
    int bits = 31 - __builtin_clz(depth[u]);
    if (!isAncestor(u, v)) {
        for (int i = bits; i >= 0; i--) {
            if (anc[u][i] != -1 && !isAncestor(anc[u][i], v)) {
                ans = f(ans, st[u][i]);
                u = anc[u][i];
            }
        }
        ans = f(ans, st[u][0]);
        u = anc[u][0];
    }
    return ans;
}

int query(int u, int v) { //O(log n)
    return f(lift(u, v), lift(v, u));
}
```

Referencias

- ❑ CP-algorithms: https://cp-algorithms.com/graph/lca_binary_lifting.html
- ❑ CP-algorithms: <https://cp-algorithms.com/graph/lca.html>