LCA y Binary Lifting

Marcelo Lemus

marcelo.lemus@progcomp.cl

Repaso de Grafos

Definicion de grafo

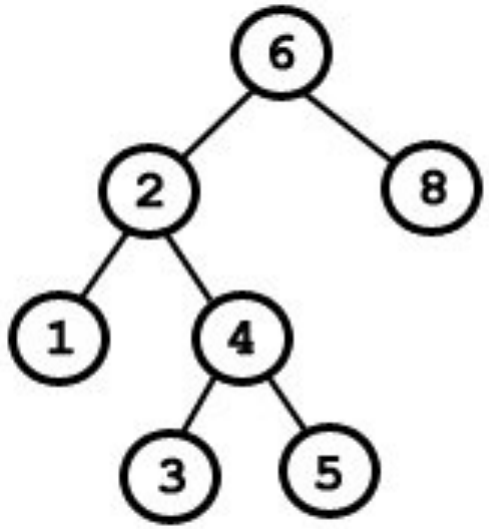
Un **grafo** $G = (V, E)$ consiste en:

- V: Conjunto de **vértices** (nodos)
- E: Conjunto de **aristas** (pares de vértices)

Las aristas implican una relación entre los pares de vertices que conecta

Definicion de arbol

Un **arbol** es un grafo que es conexo, sin ciclos y para cada par de vertices, existe solo 1 camino simple que conecta estos vertices



Matriz de adyacencia

La matriz de adyacencia, es una matriz de n x n en la cual n es la cantidad de nodos que posee el grafo. En la posición (i, j) posee un 1 (o true) si hay una arista desde el nodo i hacia el nodo j, y un 0 (o false) en caso contrario. Uso de memoria: $O(n^2)$

```
int main (){
  int n , m ;
  cin >> n >> m;
  int adj[n][n];
  for (int i=0; i<m; i++){
    int a, b;
    cin >> a >> b; // 0 - index
    adj[a][b] = 1;
    adj[b][a] = 1; // si el grafo es NO dirigido
  }
}
```

Lista de adyacencia

La lista de adyacencia, es un vector de vectores de enteros. En la i – esima posicion del vector, se encuentra un vector que indica hacia que nodos apunta el nodo i. Uso de memoria: $O(m)$, siendo m la cantidad de aristas del grafo.

Repaso de Grafos

```
int main (){
  int n , m ;
  cin >> n >> m;
  vector<vector<int>> adj;
  grafo.resize(n);
  for(int i=0; i<m; i++){
    int a, b;
    cin >> a >> b; // indexado en 0
    adj[a].push_back(b);
    adj[b].push_back(a); // si el grafo es NO dirigido
  }
}
```

Depth first search

Como indica su nombre, se realiza un recorrido en profundidad, de la siguiente manera:

- Seleccionamos un nodo inicial, elegimos un vecino aun no procesado, y procedemos a procesarlo.
- Aplicamos la logica anterior al nodo seleccionado, y cuando terminemos, procedemos a procesar el siguiente en la lista.
  Complejidad: $O(n + m)$;

Repaso de Grafos

```cpp
vector<vector<int>> adj;
vector<bool> vis;

void dfs(int u){
  vis[u] = true;
  for(int v : adj[u]){
    if(!vis[v]){
      dfs(v);
    }
  }
}

int main(){
  int n, m;
  cin >> n >> m;
  adj.resize(n);
  vis.resize(n);
  // leer el grafo
  dfs(0);
}
```

Problema

Estás en un juego donde tu personaje está en una cueva de hielo con múltiples niveles. Para avanzar, debes caer al nivel inferior rompiendo el hielo.

Descripción del Nivel:

- Una cuadrícula de $n \leq 500$ filas y $m \leq 500$ columnas.
- Cada celda tiene hielo **intacto** o **agrietado**.
- Movimientos válidos: a celdas adyacentes (arriba, abajo, izquierda, derecha).
  - Si te mueves a hielo **agrietado**, caes al siguiente nivel.
  - Si te mueves a hielo **intacto**, este se agrieta (pero no caes).

Posiciones:

- Inicias en $(r1, c1)$ (ya agrietada).
- Debes caer por $(r2, c2)$ para pasar al siguiente nivel.

Objetivo:
Imprimir si es posible llegar de $(r1, c1)$ a $(r2, c2)$ y caer (terminar en hielo agrietado).

Problema

	1	2	3	4	5	6
1	X	.	.	.	X	X
2	.	.	.	X	X	.
3	.	X	.	.	X	.
4	.	.	.	.	.	.

$inicio = (1, 6)$
$fin = (2, 2)$

Problema

```
bool e(int i, int j){
    return i>=0 && i<n && j>=0 && j<m;
}

int dx[4] = {-1, 0, 1, 0};
int dy[4] = {0, 1, 0, -1};

void dfs(int i, int j, int ffi, int ffj){
    vis[i][j] = true;
    rep(k, 4){
        int ni = i+dx[k];
        int nj = j+dy[k];
        if(e(ni, nj) && !vis[ni][nj]){
            if(ni == ffi && nj == ffj) vis[ni][nj] = true;
            if(a[ni][nj] == 'X') continue;
            dfs(ni, nj, ffi, ffj);
        }
    }
}
```

Problema

```
if(x1 == x2 && y1 == y2){
    bool can = false;
    rep(k, 4){
        int ni = x1+dx[k];
        int nj = y1+dy[k];
        if(e(ni, nj) && a[ni][nj] == '.'){
            can = true;
        }
    }
    cout << (can ? "YES\n" : "NO\n");
    return;
}

dfs(x1, y1, x2, y2);

a[x1][y1] = '.';
if(vis[x2][y2]){
    if(a[x2][y2] == 'X'){
        cout << "YES\n";
        return;
    }else{
        int cnt = 0;
        rep(k, 4){
            int ni = x2 + dx[k];
            int nj = y2 + dy[k];
            cnt += (e(ni, nj) && a[ni][nj] == '.');
        }
        cout << (cnt >= 2 ? "YES\n" : "NO\n");
    }
}else{
    cout << "NO\n";
}
```

Breadth first search

La propiedad de este recorrido es que se realiza una busqueda ”por niveles” o ”en ancho” de profundidad, de la siguiente manera:

- Seleccionamos un nodo inicial, y luego procesamos todos los nodos que se encuentran a distancia 1, es decir, que tienen una arista directa.
- Luego, procesamos todos los vecinos (no procesados aun) de los utilizados en la anterior iteracion (por lo tanto, van a ser todos los nodos a distancia 2 del nodo inicial).
- Repetimos hasta procesar todos los nodos.
  Complejidad: $O(n + m)$

Breadth first search

```
vector<int> dis; // incialmente todas las posiciones en -1
vector<vector<int>> adj;

void bfs(int start){
  queue<int> q;
  dis[start] = 0;
  q.push(start);
  while(!q.empty()){
    int nodoActual = q.front();
    q.pop();
    for(int vecino : adj[nodoActual]){
      if(dis[vecino] == -1){
        dis[vecino] = dis[nodoActual] + 1;
        q.push(vecino);
      }
    }
  }
}
```

Problema: Trapmigiano Reggiano

Un ratón hambriento comienza en el vértice st de un árbol con n vértices. Dada una permutación p de longitud n, en cada paso i:

- Aparece queso en el vértice p_i.
- Si el ratón ya está en p_i, se queda ahí.
- Si no, se mueve un paso hacia p_i por el camino más corto.

Objetivo:
Imprimir una permutación p tal que después de n pasos, el ratón termine en el vértice en (trampa).

Problema: Trapmigiano Reggiano

```
int n, dis[mxN], p[mxN];
vector<int> adj[mxN];

void init(){
    for(int i = 0; i<n; ++i){
        dis[i] = 1e9;
        p[i] = i;
    }
}
```

Problema: Trapmigiano Reggiano

```
void solve(){
    int st, en;
    cin >> n >> st >> en, --st, --en;
    init();
    rep(i, n-1){
        int a, b;
        cin >> a >> b, --a, --b;
        adj[a].push_back(b);
        adj[b].push_back(a);
    }
    bfs(en);
    sort(p, p+n, [&](int x, int y){
        return dis[x] > dis[y];
    });
    for(int i = 0; i<n; ++i) cout << p[i]+1 << " ";
}
```

Problema

Dado:

- Un grafo ponderado, donde todos los pesos son no-negativos
- Un vertice de inicio st

Calcular la minima distancia de st hacia todos los nodos

Algoritmo

Inicialmente:

- La distancia de st a si mismo es 0, es decir, d_{st} = 0.
- La distancia de st a cualquier otro vertice x es infinito (un numero lo suficientemente grande), es decir, $d_x = \infty$.

Luego, en cada paso del algoritmo:

- Elegimos un vertice x (no procesado) cuya d_x sea la menor entre los vertices no procesados.
- Por cada vecino y de x, actualizamos su distancia: $d_y = min(d_y, d_x + w_{x,y})$.

Complejidad

- En cada paso del algoritmo (son $|V|$ pasos), seleccionamos el nodo que esta a menor distancia (buscamos en los $|V|$ nodos).
- En cada paso vamos a mirar todas las aristas del vertice actual.
- La complejidad resultante es: $O(|V| + |E|)$.
- Si usamos una estructura (como un set o una cola de prioridades) para buscar el vertice sin procesar a menor distancia. La complejidad final sera: $O(|V|log(|E|) + |E|)$.

Dijkstra

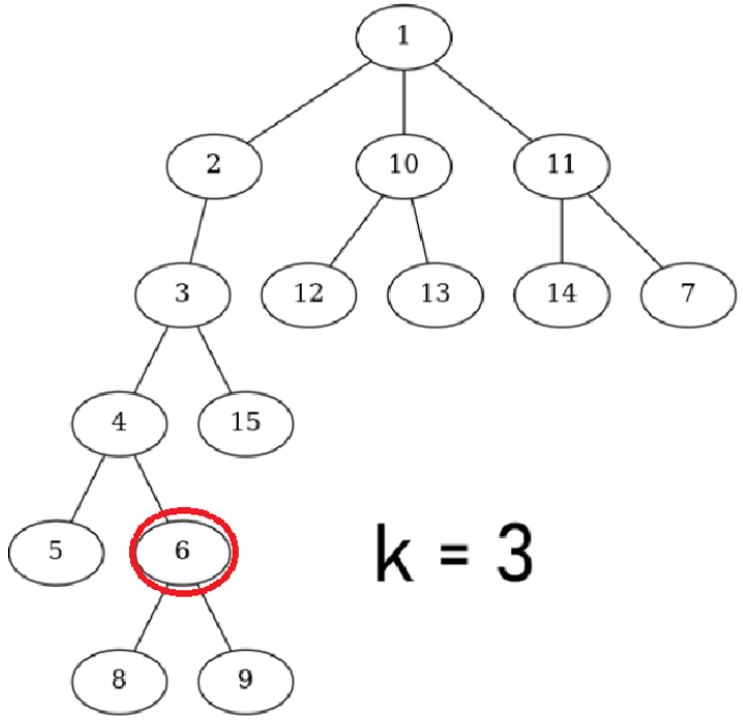
```cpp
vector<int> dis; // distancia minima hacia los nodos si empiezo en st
vector<vector<pair<int, int>>> adj; // {costo_arista, vecino}

void dijkstra(int st){
  for(int i = 0; i<n; ++i) dis[i] = 1e9; // un numero suficientemente grande
  dis[st] = 0;
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
  pq.push({0, st}); // {dx, x}
  while(!pq.empty()){
    int dx = pq.top().first;
    int x = pq.top().second;
    if(dx > dis[x]) continue;
    for(auto &[w_xy, y] : adj[x]){
      if(dis[y] > dx + w_xy){
        dis[y] = dx+w_xy;
        pq.push({dis[y], y});
      }
    }
  }
}
```

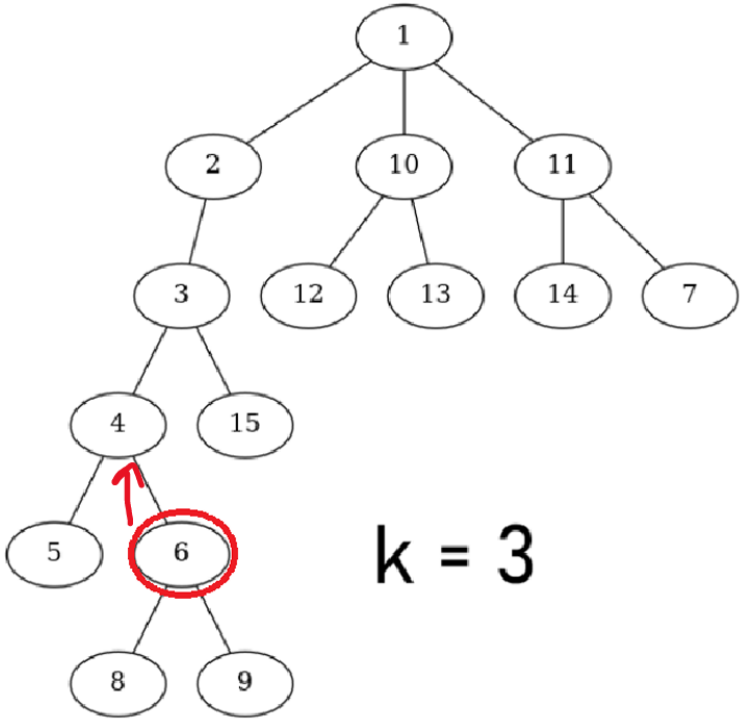
LCA y Binary Lifting

Problema

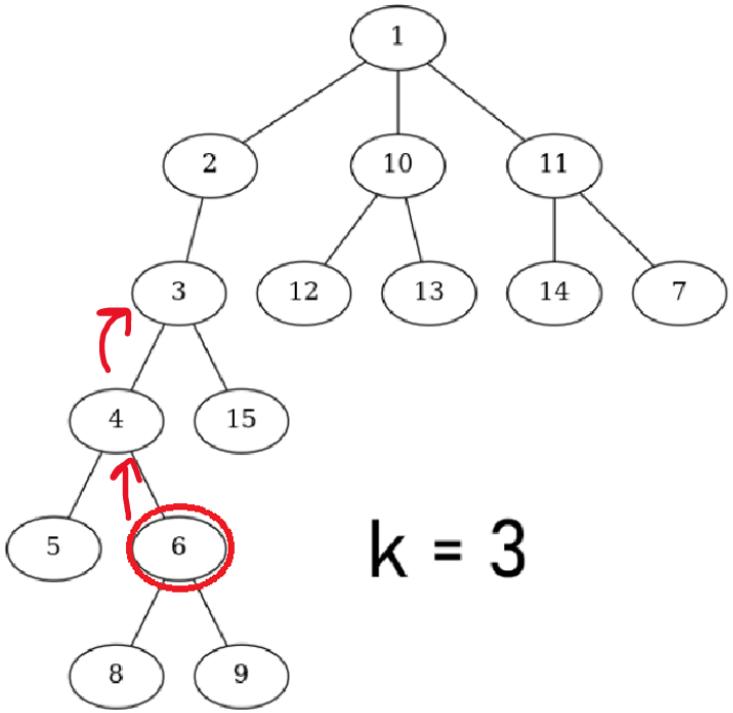
Dado un árbol, queremos conseguir el k-esimo ancestro de un nodo "u" de forma eficiente. Queremos hacer muchas consultas de este tipo



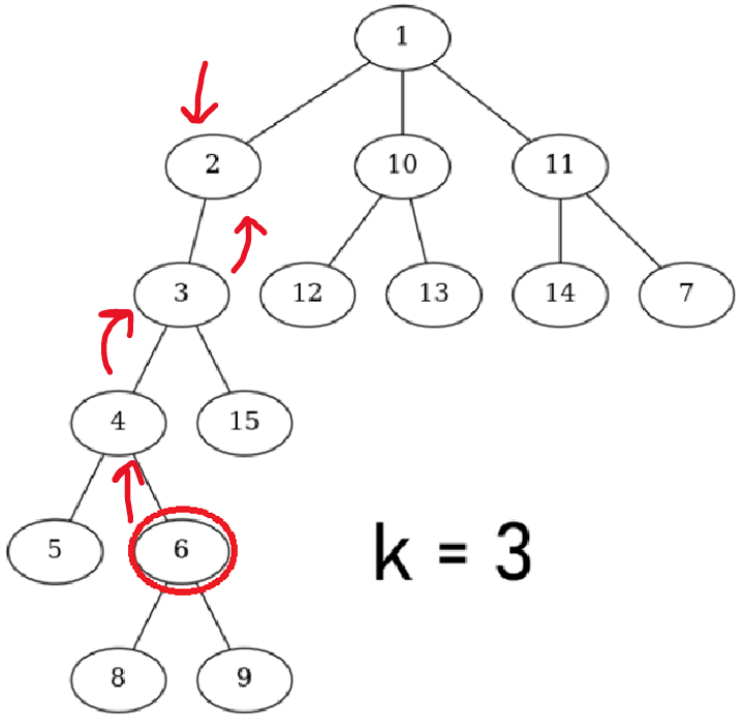
Problema



Problema



Problema



Posible solución

Iterar k veces, cada vez, subo un nodo hacia arriba.

```
for(int i = 0; i<k; ++i){
  u = parent[u];
}
```

$O(k) \rightarrow O(n)$. Con muchas consultas, $\rightarrow O(Q * n)$

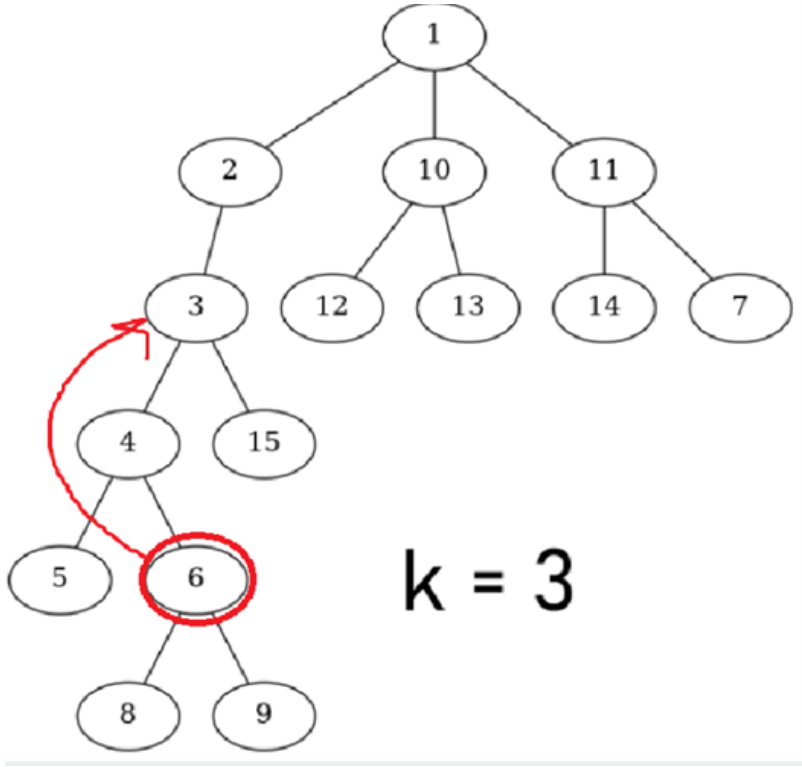
Como hacerlo más rapido

Podemos tomar la representacion binaria de k, sabiendo que tiene a lo mucho $\lceil log_2(k) \rceil$ bits y precalcular los "saltos" en potencias de 2

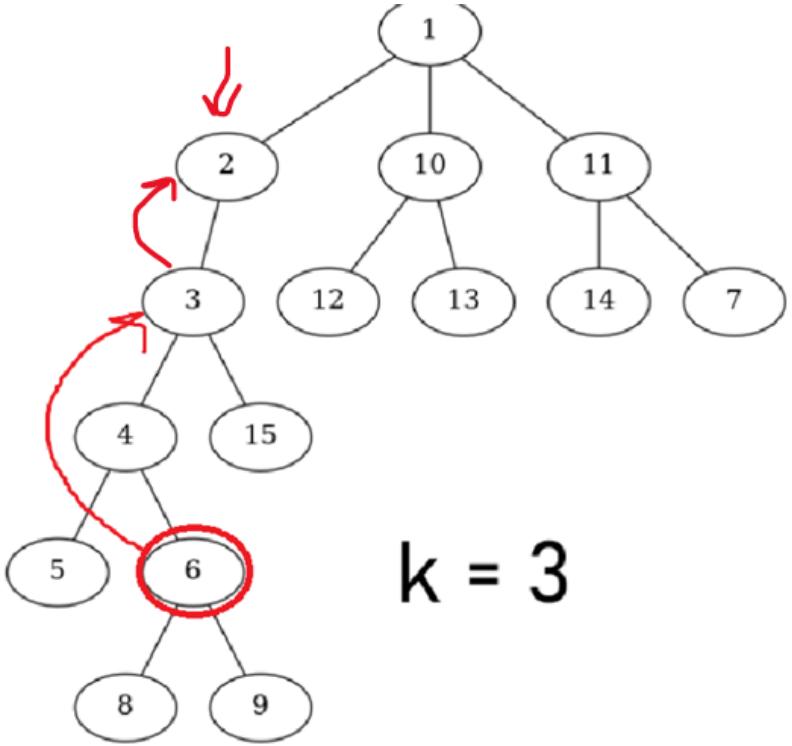
```
k = 19 = 10011 = 16+2+1
```

Si precalculamos estos saltos en potencias de 2 por cada nodo, podemos resolver cada consulta en tiempo $O(log_2(k))$

Binary lifting



Binary lifting



Binary lifting

```
const int LOG = 25, mxN = 2e5+5;
int parent[mxN], depth[mxN], up[LOG][mxN];

void preprocess(int u = 0, int p = 0){
  parent[u] = up[0][u] = p;
  for(int x = 1; x<LOG; ++x){
    up[x][u] = up[x-1][up[x-1][u]];
  }
  for(int v : adj[u]){
    if(v == p) continue;
    depth[v] = depth[u]+1;
    preprocess(v, u);
  }
}
```

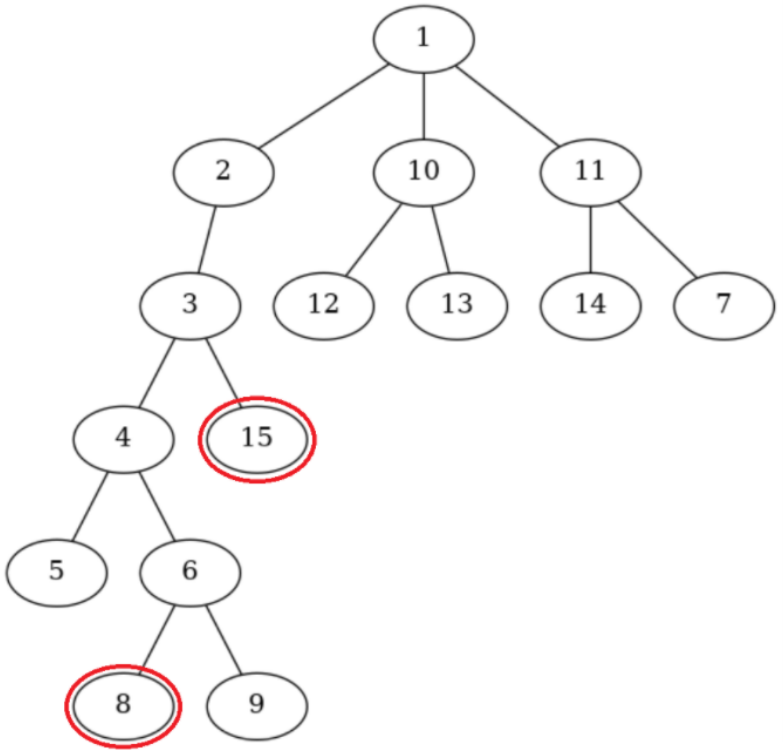
Binary lifting

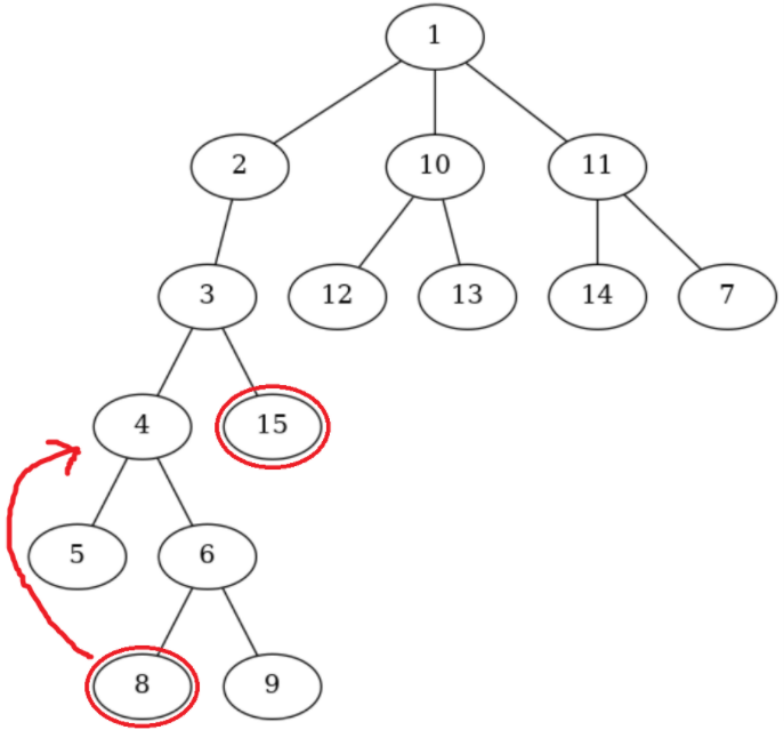
```
int k_ancestro(int u, int k){
  for(int bit = 0; bit<LOG; ++bit){
    if(k & (1<<bit)){
      u = up[k][u];
    }
  }
  return u;
}
```

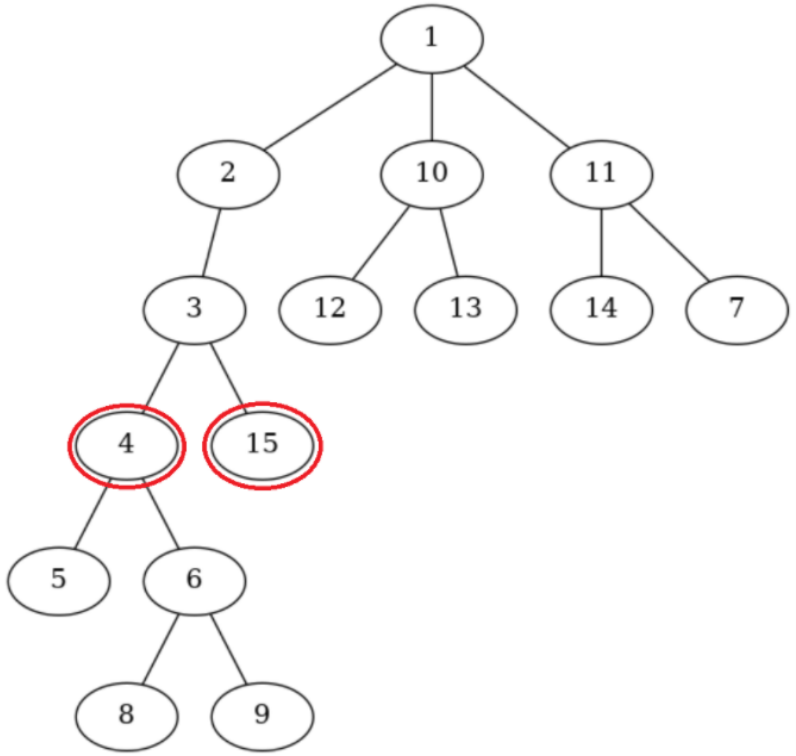
LCA

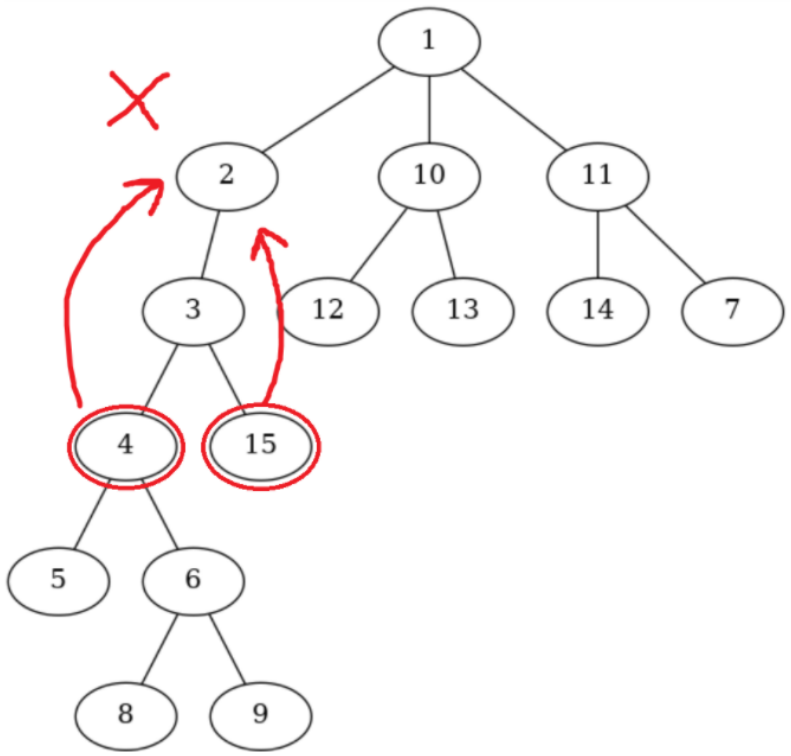
Problema: Dado un árbol, queremos obtener el ancestro común mas bajo de dos nodos (LCA).
El LCA de dos nodos es un nodo tal que es el primer ancestro de ambos de estos nodos, para calcularlo debemos seguir los siguientes pasos:

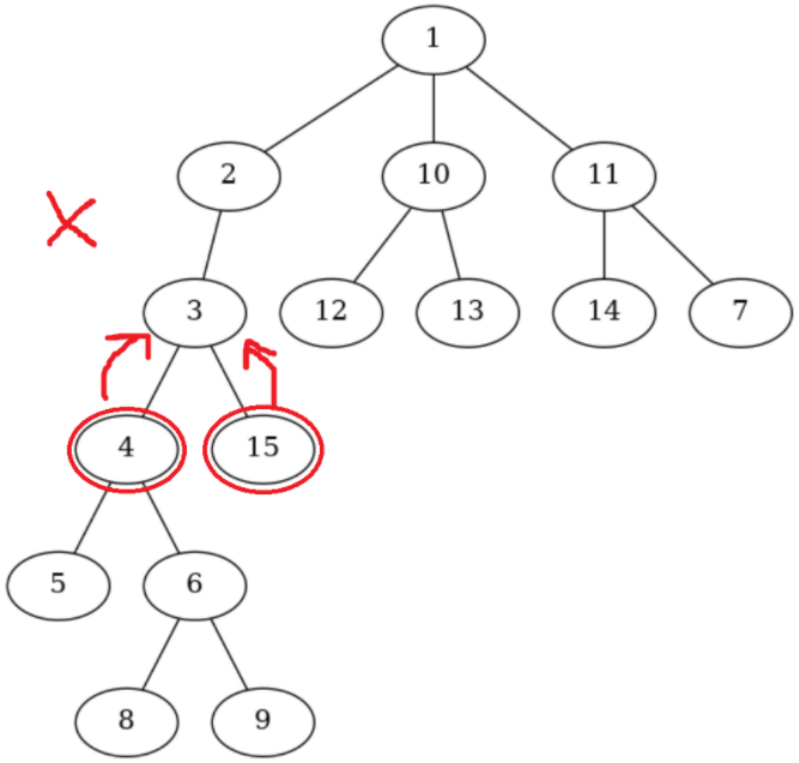
1. igualar ambos nodos en altura (usando binary lifting).
2. si ambos nodos quedan en la misma altura, entonces uno de los nodos era LCA del otro, por lo que retornamos el nodo que estaba más arriba.
3. en caso contrario, iteramos desde la potencia de 2 más grande a la más pequeña, si el 2^k ancestro de ambos nodos es el mismo, entonces estamos revisando más arriba del LCA, en caso contrario, aún no llegamos al LCA, entonces actualizamos ambos nodos a su 2^k ancestro y repetimos este paso
4. retorna el padre de alguno de los dos nodos

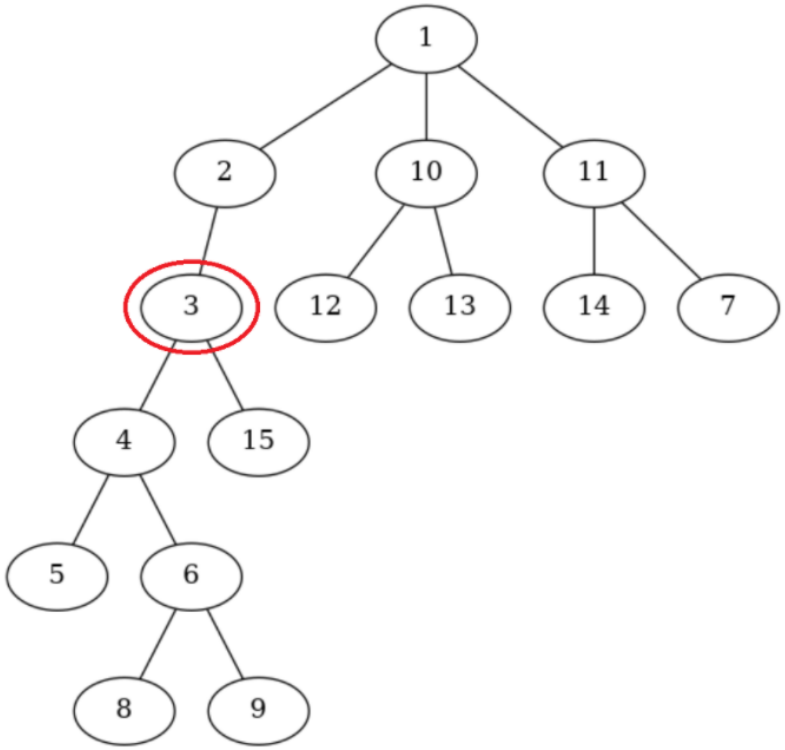

1
2
10
11
3
12
13
14
7
4
15
5
6
8
9


1
2
10
11
3
12
13
14
7
4
15
5
6
8
9


1
2
10
11
3
12
13
14
7
4
15
5
6
8
9


1
2
10
11
3
12
13
14
7
4
15
5
6
8
9


1
2
10
11
3
12
13
14
7
4
15
5
6
8
9


1
2
10
11
3
12
13
14
7
4
15
5
6
8
9

LCA

```
int lca(int a, int b){
    if(depth[a] < depth[b]) swap(a, b);
    int diff = depth[a] - depth[b];
    for(int i = 0; i<LOG; ++i){
        if(diff & (1<<i)){
            a = up[i][a];
        }
    }
    for(int i = LOG-1; i>=0; --i){
        if(up[i][a] != up[i][b]){
            a = up[i][a];
            b = up[i][b];
        }
    }

    if(a == b) return a;
    return up[0][a];
}

// codigo util, distancia entre dos nodos
int distancia(int a, int b){
  return depth[a] + depth[b] - 2*depth[lca(a, b)];
}
```

Problema: A and B and Lecture Rooms

Descripción:
La universidad donde estudian A y B tiene n salas conectadas por $n-1$ pasillos, formando un árbol. Cada día:

- A y B escriben concursos en salas distintas (a y b).
- Luego, se reúnen en una sala v tal que:
  - La distancia de v a a sea igual a la distancia de v a b.

Objetivo:
Para cada uno de los m días, calcular el número de salas v que cumplen la condición anterior.