GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: DESCOMPOSICIÓN PESADA-LIGERA (*HEAVY-LIGHT DECOMPOSITION*)

1. Introducción

En varios ejercicios o problemas de concursos se nos puede presentar la situación que tenemos un árbol donde cada nodo o aristas tiene un valor y se nos pide un grupo de consulta de la forma (a, b) donde a y b son nodos del árbol y tenemos que buscar en el único camino que existe entre a y b en al árbol aquellos valores de los nodos o aristas que conforman el camino determinada propiedad. La idea trivial sería realizar una DFS de a a b y buscar la solución pero debido a la cantidad de de consultas y lo grande que puede ser el árbol (en la mayoría de los casos hasta 10^5 nodos) esta solución puede que su tiempo sea superior al tiempo limite permitido.

En la siguiente guía abordaremos la **descomposición pesada-ligera (*Heavy-light decomposition*)** es una técnica bastante general que nos permite resolver de manera efectiva muchos problemas que se reducen a consultas en un árbol.

2. Conocimientos previos

2.1. Árbol

En ciencias de la computación y en informática, un árbol es un tipo abstracto de datos (TAD) ampliamente usado que imita la estructura jerárquica de un árbol, con un valor en la raíz y subárboles con un nodo padre, representado como un conjunto de nodos enlazados.

Una estructura de datos de árbol se puede definir de forma recursiva (localmente) como una colección de nodos (a partir de un nodo raíz), donde cada nodo es una estructura de datos con un valor, junto con una lista de referencias a los nodos (los hijos), con la condición de que ninguna referencia esté duplicada ni que ningún nodo apunte a la raíz.

2.2. Recorrido en profundidad

La búsqueda en profundidad (DFS) es un algoritmo para atravesar o buscar estructuras de datos de árboles o grafos. El algoritmo comienza en el nodo raíz (seleccionando algún nodo arbitrario como nodo raíz en el caso de un grafo) y explora lo más lejos posible a lo largo de cada rama antes de retroceder. Se necesita memoria adicional, generalmente una pila, para realizar un seguimiento de los nodos descubiertos hasta el momento a lo largo de una rama específica, lo que ayuda a retroceder en el grafo.

2.3. Ancestro Común más Bajo (*Lowest Common Ancestor* (LCA))

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).

2.4. 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).

3. Desarrollo

Que haya un árbol G de n vértices, con una raíz arbitraria. La esencia de esta descomposición del árbol es dividir el árbol en varios caminos para que podamos llegar al vértice de la raíz desde cualquier v atravesando como máximo $\log n$ caminos. Además, ninguno de estos caminos debe cruzarse con otro.

Está claro que si encontramos tal descomposición para cualquier árbol, nos permitirá reducir ciertas consultas individuales de la forma *calcular algo en el camino desde a a b* a varias consultas del tipo *calcular algo en el segmento $[l, r]$ del k^{th} camino*.

Calculamos para cada vértice v el tamaño de su subárbol $s(v)$, es decir, el número de vértices en el subárbol del vértice v incluyéndose a sí mismo.

A continuación, considere todas las aristas que conducen a los hijos de un vértice v Llamamos pesada a una arista si conduce a un vértice c tal que:

$$s(c) \geq \frac{s(v)}{2} \iff \text{arista}(v, c) \text{ es pesada}$$

Todos los demás aristas están etiquetados como ligeras.

Es obvio que a lo sumo una arista pesada puede emanar de un vértice hacia abajo, porque de lo contrario el vértice v tendría al menos dos hijos de tamaño $\geq \frac{s(v)}{2}$, y por lo tanto el tamaño del subárbol de v sería demasiado grande, $s(v) \geq 1+2\frac{s(v)}{2} > s(v)$, lo que conduce a una contradicción.

Ahora descompondremos el árbol en caminos disjuntos. Considere todos los vértices de los que no descienden aristas pesadas. Subiremos desde cada uno de esos vértices hasta llegar a la raíz del árbol o pasar por una arista ligera. Como resultado, obtendremos varias rutas que se componen de cero o más aristas pesadas más una arista ligera. El camino que termina en la raíz es una excepción a esto y no tendrá una arista ligera. Dejemos que estos se llamen caminos pesados : estos son los caminos deseados de descomposición pesado-ligero.

Primero, notamos que los caminos pesados obtenidos por el algoritmo serán **disjuntos**. De hecho, si dos de estos caminos tienen una arista común, implicaría que hay dos aristas pesadas saliendo de un vértice, lo cual es imposible.

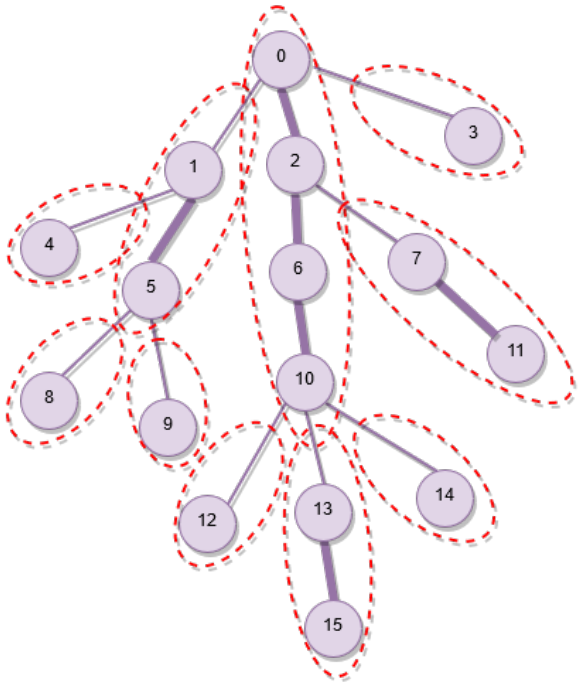
En segundo lugar, mostraremos que bajando desde la raíz del árbol hasta un vértice arbitrario, cambiaremos no más de $\log n$ caminos pesados a lo largo del camino. Bajar por una arista ligera reduce el tamaño del subárbol actual a la mitad o menos:

$$s(c) < \frac{s(v)}{2} \iff \text{arista}(v, c) \text{ es ligera}$$

Así, podemos pasar como máximo $\log n$ aristas ligeras antes de que el tamaño del subárbol se reduzca a uno.

Dado que podemos movernos de un camino pesado a otro solo a través de una arista ligera (cada camino pesado, excepto el que comienza en la raíz, tiene una arista ligera), no podemos cambiar los caminos pesados más de$\log n$ veces a lo largo del camino desde la raíz hasta cualquier vértice, según se requiera.

La siguiente imagen ilustra la descomposición de un árbol de muestra. Los bordes gruesos son más gruesos que las aristas ligeras. Los caminos pesados están marcados por límites punteados.



Ciertas partes del enfoque discutido anteriormente pueden modificarse para facilitar la implementación sin perder eficiencia.

- La definición de **arista pesada** se puede cambiar al arista que lleva al hijo con el subárbol más grande, con los lazos rotos arbitrariamente. Esto puede resultar en que algunas aristas ligeras se conviertan en pesadas, lo que significa que algunas rutas pesadas se combinarán para formar una sola ruta, pero todas las rutas pesadas permanecerán separadas. También

se garantiza que descender por una arista ligera reduce el tamaño del subárbol a la mitad o menos.

- En lugar de construir un árbol de rangos sobre cada ruta pesada, se puede usar un árbol de un solo rango con rangos separados asignados a cada ruta pesada.
- Se ha mencionado que responder consultas requiere el cálculo del LCA. Si bien LCA se puede calcular por separado, también es posible integrar el cálculo de LCA en el proceso de respuesta a consultas.

4. Implementación

4.1. C++

```
vector<int> parent, depth, heavy, head, pos;
int cur_pos;

int dfs(int v, vector<vector<int>> const& adj) {
   int size = 1;
   int max_c_size = 0;
   for (int c : adj[v]) {
      if (c != parent[v]) {
         parent[c] = v, depth[c] = depth[v] + 1;
         int c_size = dfs(c, adj);
         size += c_size;
         if(c_size > max_c_size)
            max_c_size = c_size, heavy[v] = c;
      }
   }
   return size;
}

void decompose(int v, int h, vector<vector<int>> const& adj) {
   head[v] = h, pos[v] = cur_pos++;
   if (heavy[v] != -1) decompose(heavy[v], h, adj);
   for (int c : adj[v]) {
      if (c != parent[v] && c != heavy[v])
         decompose(c, c, adj);
   }
}

void init(vector<vector<int>> const& adj, int root) {
   int n = adj.size(); parent = vector<int>(n);
   depth = vector<int>(n); heavy = vector<int>(n, -1);
   head = vector<int>(n); pos = vector<int>(n);
   cur_pos = 1; dfs(root, adj);
   decompose(root, 0, adj);
}
```

La lista de adyacencia del árbol y su raiz debe pasarse a la función *init*. La función *dfs* se usa para calcular $heavy[v]$, el hijo en el otro extremo de la arista pesada desde v, para cada vértice v. Además, *dfs* también almacena el padre y la profundidad de cada vértice, lo que será útil más adelante durante las consultas. La función *decompose* asigna para cada vértice v los valores $head[v]$ y $pos[v]$, que son, respectivamente, la cabeza del camino pesado al que v pertenece y la posición de v en el árbol de rango único que cubre todos los vértices.

```
struct segtree {
   struct node {
      // Valor inicial de los nodos hojas
      long long mx = LLONG_MIN;
      long long mn = LLONG_MAX;
      long long sum = 0;
      long long add = 0;

      void apply(int l, int r, long long v) {
         mx = v; mn = v;
         sum += (r - l + 1) * v;
         add += v;
     }
   };

   int nnodes;
   vector<node> tree;

   static node unite(const node &a, const node &b) {
      node res; res.mx = max(a.mx, b.mx);
      res.mn = min(a.mn, b.mn); res.sum = a.sum + b.sum;
      return res;
   }

   segtree(int _n) : nnodes(_n) {
      tree.resize(2 * nnodes - 1); build(0, 0, nnodes - 1);
   }

   void pull(int x, int y) {
      tree[x] = unite(tree[x + 1], tree[y]);
   }

   void push(int x, int l, int r) {
      int m = (l + r) >> 1; int y = x + ((m - l + 1) << 1);
      if (tree[x].add != 0) {
         tree[x + 1].apply(l, m, tree[x].add);
         tree[y].apply(m + 1, r, tree[x].add);
         tree[x].add = 0;
      }
   }

   void build(int x, int l, int r) {
```

```
      if (l == r) { return;}
      int m = (l + r) >> 1;
      int y = x + ((m - l + 1) << 1);
      build(x + 1, l, m); build(y, m + 1, r);
      pull(x, y);
   }

   node get(int x, int l, int r, int ll, int rr) {
      if (ll <= l && r <= rr){ return tree[x];}
      int m = (l+r)>>1; int y = x+((m-l+1)<<1);
      push(x, l, r);node res;
      if (rr <= m){res=get(x+1,l,m,ll,rr);}
      else{
         if(ll > m){res = get(y, m + 1, r, ll, rr);}
         else{
            res=unite(get(x+1,l,m,ll,rr),get(y,m+1,r,ll,rr));
         }
      }
      pull(x, y); return res;
   }

   node get(int ll, int rr) {
      return get(0, 0, nnodes - 1, ll, rr);
   }

   void modify(int x, int l, int r, int ll, int rr, long &v) {
      if(ll <= l && r <= rr){tree[x].apply(l,r,v);return;}
      int m = (l + r) >> 1; int y = x + ((m - l + 1) << 1);
      push(x, l, r);
      if (ll <= m){ modify(x + 1, l, m, ll, rr, v);}
      if (rr > m){ modify(y, m + 1, r, ll, rr, v);}
      pull(x, y);
   }

   void modify(int ll, int rr, long v) {
      modify(0, 0, nnodes - 1, ll, rr, v);
   }
};

class HeavyLight {
   public:
      vector<vector<int>> tree;
      // verdadero los valores en los vertices
      // falso los valores en las aristas
      bool valuesOnVertices;
      segtree segment_tree;
      vector<int> parent,depth,pathRoot,in;

      HeavyLight(const vector<vector<int>> &t,int root, bool valuesOnVertices)
         :
```

```
    tree(t), valuesOnVertices(valuesOnVertices), segment_tree(t.size()),
    parent(t.size()), depth(t.size()), pathRoot(t.size()), in(t.size()){

    int time = 0;parent[root] = -1;

    function<int(int)> dfs1 = [&](int u) {
       int size = 1; int maxSubtree = 0;
       for (int &v : tree[u]) {
          if (v == parent[u]) continue;
          parent[v] = u;
          depth[v] = depth[u] + 1;
          int subtree = dfs1(v);
                  if (maxSubtree < subtree) {
             maxSubtree = subtree;
             int t = v; v = tree[u][0]; tree[u][0] = t;
          }
          size += subtree;
       }
       return size;
     };

     function<void(int)> dfs2 = [&](int u) {
        in[u] = time++;
        for (int v : t[u]) {
           if (v == parent[u]) continue;
           pathRoot[v] = v == t[u][0] ? pathRoot[u] : v;
           dfs2(v);
        }
     };
     dfs1(root);dfs2(root);
}

segtree::node get(int u, int v) {
   segtree::node res;
   process_path(u, v, [this, &res](int a, int b) {
      res = segtree::unite(res, segment_tree.get(a, b));});
   return res;
}

void modify(int u, int v, long long delta) {
   process_path(u, v, [this, delta](int a, int b) {
      segment_tree.modify(a, b, delta);});
}

void process_path(int u,int v, const function<void(int x,int y)>&op){
   for(;pathRoot[u]!=pathRoot[v];v=parent[pathRoot[v]]){
      if(depth[pathRoot[u]]>depth[pathRoot[v]]){
         int t = u; u = v; v = t;
      }
      op(in[pathRoot[v]], in[v]);
```

```
            }
            if (u != v || valuesOnVertices)
               op(min(in[u],in[v])+(valuesOnVertices ? 0:1),max(in[u],in[v]));
         }
};
```

4.2. Java

```
import java.util.function.BiConsumer;

private class HeavyLightDescomposition {
   private List<Integer>[] tree;
   // Si es verdadero los valores en los vertices
   // falso los valores en la aristas
   private boolean valuesOnVertices;
   private SegmentTree segmentTree;
   private int[] parent;
   private int[] depth;
   private int[] pathRoot;
   private int[] in;
   private int time;

   public HeavyLightDescomposition(List<Integer>[] tree,int root,boolean vOV){
      this.tree = tree; this.valuesOnVertices = vOV;
      int n = tree.length; segmentTree = new SegmentTree(n);
      parent = new int[n]; depth = new int[n];
      pathRoot = new int[n]; in = new int[n];
      parent[root] = -1; dfs1(root); dfs2(root);
   }

   private int dfs1(int u) {
      int size = 1; int maxSubtree = 0;
      for (int i = 0; i < tree[u].size(); i++) {
         int v = tree[u].get(i);
         if (v == parent[u]) continue;
         parent[v] = u; depth[v] = depth[u] + 1;
         int subtree = dfs1(v);
         if (maxSubtree < subtree) {
            maxSubtree = subtree;
            tree[u].set(i, tree[u].set(0, v));
         }
         size += subtree;
      }
      return size;
   }

   void dfs2(int u) {
      in[u] = time++;
```

```
        for (int v : tree[u]) {
            if (v == parent[u]) continue;
            pathRoot[v] = v == tree[u].get(0) ? pathRoot[u] : v;
            dfs2(v);
        }
    }

    public Node get(int u, int v) {
        Node[] res = { new Node() };
        processPath(u, v, (a, b) -> res[0] = segmentTree.unite(res[0],
            segmentTree.get(a, b)));
        return res[0];
    }

    public void modify(int u, int v, long delta) {
        processPath(u, v, (a, b) -> segmentTree.modify(a, b, delta));
    }

    public void processPath(int u, int v, BiConsumer<Integer, Integer> op) {
        for (; pathRoot[u] != pathRoot[v]; v = parent[pathRoot[v]]) {
            if (depth[pathRoot[u]] > depth[pathRoot[v]]) {
                int t = u; u = v; v = t;
            }
            op.accept(in[pathRoot[v]], in[v]);
        }
        if (u != v || valuesOnVertices)
          op.accept(Math.min(in[u], in[v]) + (valuesOnVertices ? 0 : 1), Math.
              max(in[u], in[v]));
    }

    private class Node {
      // Valor inicial para las hojas
        public long mx = Long.MIN_VALUE;
        public long mn = Long.MAX_VALUE;
        public long sum = 0;
        public long add = 0;

        public Node(){}

        public void apply(int l, int r, long v) {
            mx = v; sum += v * (r - l + 1);
            add += v; mn = v;
        }
    }

    private class SegmentTree {
        private int nnodes;
        private Node[] tree;

        public Node unite(Node a, Node b) {
```

```
        Node res = new Node(); res.mx = Math.max(a.mx, b.mx);
        res.mn = Math.min(a.mn, b.mn); res.sum = a.sum + b.sum;
        return res;
    }

    public SegmentTree(int n) {
        this.nnodes = n; tree = new Node[2 * n - 1];
        for (int i = 0; i < tree.length; i++) tree[i] = new Node();
        build(0, 1, n - 1);
    }

    private void pull(int x, int y) {
        tree[x] = unite(tree[x + 1], tree[y]);
    }

    void push(int x, int l, int r) {
        int m = (l + r) >> 1; int y = x + ((m - l + 1) << 1);
        if (tree[x].add != 0) {
            tree[x + 1].apply(l, m, tree[x].add);
            tree[y].apply(m + 1, r, tree[x].add); tree[x].add = 0;
        }
    }

    private void build(int x, int l, int r) {
        if (l == r) { return; }
        int m = (l + r) >> 1; int y = x + ((m - l + 1) << 1);
        build(x + 1, l, m); build(y, m + 1, r); pull(x, y);
    }

    private Node get(int ll, int rr) {
        return get(ll, rr, 0, 0, this.nnodes - 1);
    }

    public Node get(int ll, int rr, int x, int l, int r) {
        if (ll <= l && r <= rr) { return tree[x]; }
        int m = (l + r) >> 1; int y = x + ((m - l + 1) << 1);
        push(x, l, r); Node res;
        if (rr <= m) { res = get(ll, rr, x + 1, l, m);}
        else {
            if(ll > m){ res = get(ll, rr, y, m + 1, r);}
            else{
                res=unite(get(ll,rr,x+1,l,m),get(ll,rr,y,m+1,r));
            }
        }
        pull(x, y); return res;
    }

    public void modify(int ll, int rr, long v) {
        modify(ll, rr, v, 0, 0, this.nnodes - 1);
    }
```

```
        public void modify(int ll, int rr, long v, int x, int l, int r) {
            if(ll<=l && r<=rr){
                tree[x].apply(l, r, v); return;
            }
            int m = (l + r) >> 1; int y = x + ((m - l + 1) << 1);
            push(x, l, r);
            if(ll<=m){modify(ll, rr, v, x + 1, l, m);}
            if(rr>m){ modify(ll, rr, v, y, m + 1, r);}
            pull(x, y);
        }
    }
}
```

5. Complejidad

La complejidad de esta técnica va a depender en gran medida de la situación que se use pero cuando veamos algunas de sus aplicaciones veremos que los tiempos de ejecucción estarán en el orden de los $\mathcal{O}(\log n)$ y $\mathcal{O}(\log^2 n)$.

6. Aplicaciones

A continuación, veremos algunas tareas típicas que se pueden resolver con la ayuda de la descomposición pesada-ligera.

Por separado, vale la pena prestar atención al problema de la **suma de números en el camino** , ya que este es un ejemplo de un problema que puede resolverse con técnicas más simples.

6.1. Valor máximo en el camino entre dos vértices

Dado un árbol, a cada vértice se le asigna un valor. Hay consultas de la forma (a, b), dónde a y b son dos vértices en el árbol, y se requiere encontrar el valor máximo en el camino entre los vértices a y b.

Construimos de antemano una descomposición pesada-ligera del árbol. Sobre cada ruta pesada construiremos un árbol de segmentos, que nos permitirá buscar un vértice con el valor máximo asignado en el segmento especificado de la ruta pesada especificada en $\mathcal{O}(\log n)$. Aunque el número de caminos pesados en la descomposición pesado-ligero puede alcanzar $n - 1$, el tamaño total de todos los caminos está limitado por $\mathcal{O}(n)$, por lo tanto, el tamaño total de los árboles de segmento también será lineal.

Para responder una consulta (a, b), encontramos el ancestro común más bajo de a y b como l, por cualquier método preferido. Ahora la tarea se ha reducido a dos consultas. (a, l) y (b, l), para cada uno de los cuales podemos hacer lo siguiente: encontrar el camino pesado en el que se encuentra el vértice inferior, hacer una consulta sobre este camino, movernos a la parte superior de

este camino, nuevamente determinar en qué camino pesado estamos y hacer una consulta sobre y así sucesivamente, hasta llegar a la ruta que contiene l.

Se debe tener cuidado con el caso cuando, por ejemplo, a y l están en la misma ruta pesada; entonces, la consulta máxima en esta ruta no debe realizarse en ningún prefijo, sino en la sección interna entre a y l.

Respondiendo a las subconsultas (a, l) y (b, l) cada uno requiere pasar por $\mathcal{O}(\log n)$ rutas pesadas y para cada ruta se realiza una consulta máxima en alguna sección de la ruta, lo que nuevamente requiere $\mathcal{O}(\log n)$ operaciones en el árbol de rangos. Por lo tanto, una consulta (a, b) acepta $\mathcal{O}(\log^2 n)$ tiempo.

Si adicionalmente calcula y almacena los máximos de todos los prefijos para cada ruta pesada, entonces obtiene un $\mathcal{O}(\log n)$ solución porque todas las consultas máximas son sobre prefijos, excepto como máximo una vez cuando llegamos al antepasado l.

Para responder consultas sobre rutas, por ejemplo, la consulta máxima discutida, podemos hacer algo como esto:

```
int query(int a, int b) {
   int res = 0;
   for (; head[a] != head[b]; b = parent[head[b]]) {
      if (depth[head[a]] > depth[head[b]]){
         int t = a; a = b; b =t;
          }
      int cur_heavy_path_max = segment_tree_query(pos[head[b]], pos[b]);
      res = max(res, cur_heavy_path_max);
   }
   if (depth[a] > depth[b]){
      int t = a; a = b; b =t;
   }
   int last_heavy_path_max = segment_tree_query(pos[a], pos[b]);
   res = max(res, last_heavy_path_max); return res;
}
```

6.2. Suma de los números en el camino entre dos vértices

Dado un árbol, a cada vértice se le asigna un valor. Hay consultas de la forma (a, b), dónde a y b son dos vértices en el árbol, y se requiere encontrar la suma de los valores en el camino entre los vértices a y b. Es posible una variante de esta tarea donde adicionalmente hay operaciones de actualización que cambian el número asignado a uno o más vértices.

Esta tarea se puede resolver de manera similar al problema anterior de máximos con la ayuda de la descomposición pesada-ligera mediante la construcción de árboles de segmentos en caminos pesados. En su lugar, se pueden usar sumas de prefijos si no hay actualizaciones. Sin embargo, este problema también se puede resolver con técnicas más simples.

Si no hay actualizaciones, entonces es posible averiguar la suma en el camino entre dos vértices

en paralelo con la búsqueda LCA de dos vértices por elevación binaria ; para esto, junto con el 2^k-th ancestros de cada vértice también es necesario almacenar la suma en los caminos hasta esos ancestros durante el preprocesamiento.

Hay un enfoque fundamentalmente diferente para este problema: considerar el recorrido de Euler por el árbol y construir un árbol de segmentos sobre él. Nuevamente, si no hay actualizaciones, almacenar sumas de prefijos es suficiente y no se requiere un árbol de segmentos.

Ambos métodos proporcionan soluciones relativamente simples que toman $\mathcal{O}(\log n)$ para una consulta.

6.3. Repintar los bordes del camino entre dos vértices

Dado un árbol, cada arista se pinta inicialmente de blanco. Hay actualizaciones de la forma (a, b, c), dónde a y b son dos vértices y c es un color, que indica que todos las aristas en el camino desde a a b debe ser repintado con color c. Después de todos los repintados, se requiere informar cuántos aristas de cada color se obtuvieron.

Similar a los problemas anteriores, la solución es simplemente aplicar la descomposición pesada-ligera y hacer un árbol de segmentos sobre cada camino pesado.

Cada repintado en el camino (a, b) se convertirá en dos actualizaciones (a, l) y (b, l), dónde l es el ancestro común más bajo de los vértices a y b $\mathcal{O}(\log n)$ por camino para $\mathcal{O}(\log n)$ caminos conduce a una complejidad de $\mathcal{O}(\log^2 n)$ por actualización.

7. Ejercicios propuestos

A continuación una lista de ejercicios que su solución se basan en lo abordado en esta guía

- SPOJ - QTREE - Query on a tree
- DMOJ - Estructuras Celulares