Resolución de Tarea 6 - Árboles (Fecha: 10 de Noviembre de 2025)

Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI5651 - Diseño de Algoritmos I
Septiembre - Diciembre 2025
Estudiante: Junior Miguel Lara Torres (17-10303)

Tarea 6 (9 puntos)

Indice

Pregunta 1

Las pistas sugieren usar una estructura que permita **dividir o reunir subarreglos eficientemente con una alta probabilidad** y **evitar mover los elementos uno por uno**, por lo que la solución propuesta a continuación usará **treaps**.

Un Treap (combinación de *tree* y *heap*) es un árbol binario donde cada nodo tiene un par (clave, prioridad).

- El Treap es un árbol binario de búsqueda (BST) respecto a las claves.
- Es un *heap* respecto a las prioridades.
- Al asignar prioridades aleatorias a los nodos, **el Treap logra mantener un promedio de** $O(\log n)$ **niveles**, lo que lo hace útil para implementar BSTs eficientes.
- Llamaremos **Treap Implícito** a una variación de **treaps** donde las claves corresponden a los índices de una secuencia, sin ser almacenados explícitamente. Esta estructura es ideal para manejar operaciones sobre rangos en **tiempo** $O(\log n)$ **en promedio**.

La operación multiswap(a, b) implica modificar grandes sub-rangos del arreglo mediante intercambios secuenciales y recursivos. Dado que un Treap Implícito está diseñado para manipular eficientemente los subarreglos (rangos) de una secuencia en tiempo $O(\log N)$ en promedio, modelar la permutación $A[1..N]$ como un Treap Implícito permite implementar la operación eficientemente, evitando mover los elementos uno a uno.

Si cada una de las N operaciones multiswap se implementa utilizando las manipulaciones de Treap (que toman $O(\log N)$ en promedio), **la complejidad total promedio resultante es** $O(N \log N)$, cumpliendo con la restricción de tiempo propuesta.

Implementación en C++

El archivo funcional se encuentra en treaps.cpp

```cpp
struct Node {
    int value;
    int priority;
    int size;
    Node *left, *right;

    // Asignamos una prioridad aleatoria (crucial para el balanceo O(log n) promedio)
    Node(int val) : value(val), size(1), left(nullptr), right(nullptr) {
        priority = rand();
    }
};

int get_size(Node *t) {
    return t ? t->size : 0;
}

void update_size(Node *t) {
    if (t)
        t->size = get_size(t->left) + get_size(t->right) + 1;
}

/**
 * Operación SPLIT (División)
 * Divide el Treap 't' en dos ('l' con los primeros 'k' elementos y 'r' con el resto).
 * Tiempo promedio O(log n).
 */
void split(Node *t, Node *&l, Node *&r, int k) {
    if (!t) {
        l = r = nullptr;
        return;
    }

    int current_rank = get_size(t->left) + 1;

    if (k < current_rank) {
        split(t->left, l, t->left, k);
        r = t;
    } else {
        split(t->right, t->right, r, k - current_rank);
        l = t;
    }
    update_size(t);
}

/**
 * Operación MERGE (Unión)
 * Combina dos Treaps 'l' y 'r' en 't', manteniendo la propiedad de heap.
 * Tiempo promedio O(log n).
 */
void merge(Node *&t, Node *l, Node *r) {
    if (!l) {
        t = r;
        return;
    }
    if (!r) {
        t = l;
        return;
    }

```

```
    // Se elige la raíz basada en la prioridad (la de menor prioridad va arriba)
    if (l->priority < r->priority) {
        merge(l->right, l->right, r);
        t = l;
    } else {
        merge(r->left, l, r->left);
        t = r;
    }
    update_size(t);
}

/**
 * Implementa la reordenación de subarreglos simulando multiswap(a, b).
 * Intercambia el segmento T1 = A[a..b-1] por T2 = A[b..2b-a-1].
 */
void multiswap_treap(Node *&t, int a, int b, int N) {
    int L = b - a;
    int end_T2 = b + L - 1;

    // a y b están basados en 1. Las operaciones split esperan un número de elementos.
    if (end_T2 > N || L <= 0)
        return;

    Node *T_prefix = nullptr, *T_rest = nullptr;
    Node *T1 = nullptr, *T_T2_suffix = nullptr;
    Node *T2 = nullptr, *T_suffix = nullptr;

    // 1. Separar T_prefix (A[1..a-1])
    split(t, T_prefix, T_rest, a - 1);

    // 2. Separar T1 (A[a..b-1]). Longitud L.
    split(T_rest, T1, T_T2_suffix, L);

    // 3. Separar T2 (A[b..b+L-1]). Longitud L.
    split(T_T2_suffix, T2, T_suffix, L);

    // 4. Re-unir en el orden: T_prefix, T2, T1, T_suffix
    Node *T_new_1 = nullptr, *T_new_2 = nullptr;

    merge(T_new_1, T_prefix, T2);
    merge(T_new_2, T_new_1, T1);
    merge(t, T_new_2, T_suffix);
}

// Recorrido In-order para obtener la secuencia resultante
vector<int> inorder_traversal(Node *t) {
    vector<int> result;
    function<void(Node *)> traverse = [&](Node *node) {
        if (!node)
            return;
        traverse(node->left);
        result.push_back(node->value);
        traverse(node->right);
    };
    traverse(t);
    return result;
}
```

Pregunta 2

La solución a este problema es la **Descomposición Heavy-Light (Heavy-Light Decomposition, HLD)** combinada con **Árboles de Segmentos con Propagación Perezosa (Lazy Propagation)**.

El problema requiere manejar consultas de agregación (forall, exists) y actualizaciones de rango (toggle) sobre las conexiones (aristas) que forman el camino simple entre dos nodos x e y en un árbol.

El primer paso es un preacondicionamiento del árbol en $O(|N|)$ para prepararlo para las consultas de camino.

Se aplica la Descomposición Heavy-Light, cuyo objetivo es dividir el árbol $A = (N, C)$ en un conjunto de caminos disjuntos llamados "cadenas".

- Para cada nodo u, se marca la conexión con el hijo que tiene el sub-árbol de mayor tamaño como una **conexión pesada**. Todas las demás son **conexiones livianas**.
- Las cadenas pesadas están formadas por secuencias de conexiones pesadas.
- Esta técnica es crucial porque un camino entre dos nodos cualesquiera a y b solo cruzará un máximo de $O(\log |N|)$ conexiones livianas (y, por lo tanto, a lo sumo $\log n$ cadenas diferentes).
- Una vez que el árbol está descompuesto, las conexiones (aristas) se mapean a índices en arreglos lineales, uno para cada cadena.
- Se construye un **Árbol de Segmentos** asociado a cada una de estas cadenas. Estos árboles de segmentos deben estar diseñados para manejar operaciones de agregación y, fundamentalmente, operaciones de **actualización de rango** mediante la **Propagación Perezosa**.

Todas las consultas y acciones Q se realizan dividiendo el camino entre x e y en $O(\log |N|)$ segmentos lineales correspondientes a las cadenas de HLD.

La acción toggle(x, y) implica invertir el valor de verdad del predicado p para todas las aristas en el camino. Esta es una operación de actualización de rango sobre $O(\log |N|)$ segmentos lineales.

- Para realizar esto en $O(\log^2 |N|)$, se utiliza la técnica de **Propagación Perezosa** (Lazy Propagation) en los Árboles de Segmentos. La función toggle es análoga a una actualización de rango que invierte el valor (por ejemplo, XOR).
- La propagación perezosa permite posponer las actualizaciones a los nodos de bajo nivel hasta que sean estrictamente necesarias (generalmente durante una consulta), almacenando una "promesa" en los nodos intermedios. Esto evita recorrer y modificar $O(|N|)$ nodos en el peor caso de una actualización de rango.

Consultas forall(x, y) y exists(x, y) requieren combinar información a lo largo de los $O(\log |N|)$ segmentos lineales que componen el camino.

- Se recorre el camino de x a y subiendo por las cadenas pesadas hasta alcanzar el Ancestor Común Más Bajo (LCA). En cada segmento lineal recorrido:
  - Se resuelve cualquier promesa de actualización pendiente (Propagación Perezosa).
  - Se ejecuta la consulta de agregación (mínimo, máximo, o en este caso, AND/OR) en el rango relevante del Árbol de Segmentos.
- Los resultados parciales de cada segmento lineal se combinan para obtener la respuesta final del camino completo.

La complejidad total es $O(|N| + Q \log^2 |N|)$:

1. La construcción de la HLD (incluyendo el cálculo de los tamaños de sub-árboles y la identificación de las conexiones pesadas) toma $O(|N|)$. La construcción inicial de los Árboles de Segmentos (sin lazy propagation activa) también toma $O(|N|)$ en total.
2. **Tiempo por Operación ($O(\log^2 |N|)$):**
   - Una operación (consulta o acción) debe atravesar un máximo de $O(\log |N|)$ cadenas pesadas.
   - En cada cadena, se realiza una operación (consulta de rango o actualización de rango) en el Árbol de Segmentos correspondiente. Gracias a la propagación perezosa, estas operaciones en un Árbol de Segmentos balanceado toman $O(\log |N|)$.

- El costo total por operación es $O(\log |N|) \times O(\log |N|) = O(\log^2 |N|)$.

3. Se requiere $O(|N|)$ de memoria adicional para almacenar la información de la HLD, los Árboles de Segmentos (que almacenan $O(|N|)$ nodos en total) y la memoria auxiliar para la propagación perezosa.

Implementación en C++

El archivo funcional se encuentra en lazy_descomposition.cpp

```cpp
// Constante para el tamaño máximo de nodos. Asumimos N <= 100000.
const int MAX_N = 100000 + 5;
const int INF = 1e9;

// ================================================================
// 1. Segment Tree con Lazy Propagation (para operaciones de rango)
// ================================================================

// Estructura que representa un nodo en el Segment Tree
struct Node {
    int count_true; // Cantidad de aristas 'true' en el rango [L, R]
    int length;     // Longitud total del segmento (R - L + 1)
    bool lazy;      // Bandera de propagación perezosa: true si necesita toggle
};

Node st[4 * MAX_N]; // Arreglo para almacenar el Segment Tree (4*N de espacio es común)

// Función auxiliar para calcular la cantidad de nodos por nivel
void combine_nodes(int p) {
    // La cuenta total es la suma de las cuentas de los hijos
    st[p].count_true = st[2 * p].count_true + st[2 * p + 1].count_true;
}

// Aplica la operación 'toggle' (inversión de valor) a un nodo
void apply_lazy(int p) {
    if (st[p].lazy) {
        // La inversión significa que count_true se convierte en (total - count_true)
        st[p].count_true = st[p].length - st[p].count_true;

        // Si no es una hoja, pasa la promesa a los hijos
        if (st[p].length > 1) {
            st[2 * p].lazy = !st[2 * p].lazy;
            st[2 * p + 1].lazy = !st[2 * p + 1].lazy;
        }
        st[p].lazy = false; // La promesa ya fue resuelta en este nivel
    }
}

// Construye el Segment Tree
void build_st(int p, int L, int R, const vector<int> &initial_values) {
    st[p].length = R - L + 1;
    st[p].lazy = false;

    if (L == R) {
        // Las hojas representan las aristas (indexadas de 0 a |N|-2)
        // El valor inicial del predicado es 1 si es true, 0 si es false
        st[p].count_true = initial_values[L];
        return;
    }

    int mid = L + (R - L) / 2;
```

```
    build_st(2 * p, L, mid, initial_values);
    build_st(2 * p + 1, mid + 1, R, initial_values);
    combine_nodes(p); // Combina los resultados de los hijos
}

// Actualización de Rango (Toggle) con Lazy Propagation (O(log N))
void update_range_st(int p, int L, int R, int i, int j) {
    apply_lazy(p); // Resuelve promesas pendientes antes de operar

    // Si el rango del nodo está completamente fuera del rango de consulta
    if (L > j || R < i)
        return;

    // Si el rango del nodo está completamente contenido en el rango de consulta
    if (L >= i && R <= j) {
        // Aplicamos la promesa 'toggle' al nodo actual
        st[p].lazy = !st[p].lazy;
        apply_lazy(p); // Aplicar inmediatamente para actualizar count_true
        return;
    }

    // El rango intersecta parcialmente, propagamos y recursamos
    int mid = L + (R - L) / 2;
    update_range_st(2 * p, L, mid, i, j);
    update_range_st(2 * p + 1, mid + 1, R, i, j);

    // Recalculamos el valor después de las llamadas recursivas
    combine_nodes(p);
}

// Consulta de Rango (Agregación: devuelve el conteo de 'true's) (O(log N))
Node query_st(int p, int L, int R, int i, int j) {
    apply_lazy(p); // Resuelve promesas pendientes

    // Si el rango está completamente fuera
    if (L > j || R < i)
        return {0, 0, false};

    // Si el rango está completamente dentro
    if (L >= i && R <= j)
        return st[p]; // Devolvemos el nodo con su valor actualizado

    // Intersección parcial
    int mid = L + (R - L) / 2;
    Node left_res = query_st(2 * p, L, mid, i, j);
    Node right_res = query_st(2 * p + 1, mid + 1, R, i, j);

    // Combinación de resultados
    Node result;
    result.count_true = left_res.count_true + right_res.count_true;
    result.length = left_res.length + right_res.length;
    result.lazy = false;
    return result;
}

// ============================================================
// 2. Heavy-Light Decomposition (HLD)
// ============================================================

```

```cpp
// Arreglos de preprocesamiento DFS (O(|N|))
vector<pair<int, int>> adj[MAX_N]; // {vecino, índice_arista}
int parent[MAX_N];                 // Padre del nodo (para subir)
int depth[MAX_N];                  // Profundidad
int subtree_size[MAX_N];           // Tamaño del subárbol
int heavy_child[MAX_N];            // Hijo pesado (con el subárbol más grande)

// Arreglos de linealización y mapeo HLD
int chain_head[MAX_N];   // Nodo inicial (raíz) de cada cadena pesada
int pos_in_base[MAX_N];  // Posición en el array lineal del Segment Tree
int edge_to_node[MAX_N]; // Mapeo: índice lineal -> nodo 'inferior' de la arista

int time_counter; // Contador global para asignar posiciones lineales
int N_nodes;      // Número de nodos

// Arreglo temporal para inicializar el Segment Tree
vector<int> edge_initial_values;

// Primera pasada DFS: calcula profundidad, tamaño de subárbol
// e identifica hijos pesados
void dfs_preprocess(int u, int p, int d) {
    parent[u] = p;
    depth[u] = d;
    subtree_size[u] = 1;
    heavy_child[u] = -1;
    int max_size = -1;

    for (auto &edge : adj[u]) {
        int v = edge.first;
        if (v != p) {
            // Guardar el valor inicial del predicado para la arista (u, v)
            // Usamos el índice de 'v' como índice lineal temporal
            edge_initial_values[v] = edge.second;
            dfs_preprocess(v, u, d + 1);
            subtree_size[u] += subtree_size[v];

            if (subtree_size[v] > max_size) {
                max_size = subtree_size[v];
                heavy_child[u] = v; // La conexión a v es pesada
            }
        }
    }
}

// Segunda pasada DFS: realiza la descomposición y linealiza
void hld_decompose(int u, int head_node) {
    chain_head[u] = head_node;
    pos_in_base[u] = time_counter++;
    edge_to_node[pos_in_base[u]] = u;

    // Si la arista lineal representa la conexión (parent[u], u),
    // su valor inicial es el del predicado en esa arista.

    // 1. Visita primero al hijo pesado (Heavy Child)
    if (heavy_child[u] != -1)
        hld_decompose(heavy_child[u], head_node);

    // 2. Visita a los hijos livianos (Light Children),
    // que inician nuevas cadenas
```

```
    for (auto &edge : adj[u]) {
        int v = edge.first;
        if (v != parent[u] && v != heavy_child[u])
            hld_decompose(v, v); // v es la nueva cabeza de cadena
    }
}

// Inicialización completa de HLD y Segment Tree (O(|N|))
void initialize_hld(int N, int root = 1) {
    N_nodes = N;
    time_counter = 0;
    // El Segment Tree operará sobre N-1 aristas (aunque
    // la indexación va de 1 a N)
    edge_initial_values.assign(N + 1, 0);

    // Asumimos 0 como el padre de la raíz si usamos 1-based indexing
    dfs_preprocess(root, 0, 0);
    hld_decompose(root, root);

    // Reajustar edge_initial_values para que
    // coincida con la posición lineal
    vector<int> linearized_edge_values(N_nodes, 0);
    for (int u = 1; u <= N_nodes; ++u) {
        // La arista linealizada en pos_in_base[u] corresponde
        // al valor del predicado en la arista que conecta
        // parent[u] con u. Ignoramos la arista de la raíz (u=root),
        // ya que no tiene parent.
        if (u != root)
            linearized_edge_values[pos_in_base[u]] = edge_initial_values[u];
    }

    // El Segment Tree opera sobre las N-1 aristas, indexadas
    // desde 1 (o 0 si usamos 0-based). Usaremos las N-1 aristas
    // linealizadas, ocupando de 1 a N-1 en el arreglo ST. El Segment
    // Tree se construye sobre las posiciones lineales 1 a N_nodes-1.
    build_st(1, 1, N_nodes - 1, linearized_edge_values);
}

// Función de utilidad: realiza la operación
// (toggle o query) en el camino (O(log^2 N))
Node perform_path_op(int u, int v, bool is_query) {
    Node total_result = {0, 0, false}; // Contendrá el resultado agregado del camino

    while (chain_head[u] != chain_head[v]) {
        // Mueve el nodo más profundo (con cabeza de cadena más profunda) hacia arriba
        if (depth[chain_head[u]] < depth[chain_head[v]])
            swap(u, v);

        int head = chain_head[u];
        int start_pos = pos_in_base[head];
        int end_pos = pos_in_base[u];

        // La operación es sobre el rango lineal de aristas [start_pos, end_pos]
        // Las aristas están indexadas por la posición del nodo 'hijo' en la cadena.

        if (start_pos > end_pos)
            swap(start_pos, end_pos);

        if (is_query) {
```

```
            Node current_res = query_st(1, 1, N_nodes - 1, start_pos, end_pos);
            // Agregación de resultados (suma de 'true's y longitud)
            total_result.count_true += current_res.count_true;
            total_result.length += current_res.length;
        } else {
            // Acción toggle
            update_range_st(1, 1, N_nodes - 1, start_pos, end_pos);
        }

        // Sube al padre de la cabeza de la cadena actual
        u = parent[head];
    }

    // Procesar el segmento final (u y v están ahora en la misma cadena)
    // Si uno es LCA, no hay aristas entre ellos en esta pasada
    if (u == v)
        return total_result;

    // Asegura que u es el nodo más profundo para definir el rango
    if (depth[u] < depth[v])
        swap(u, v);

    // El rango final es: (pos_in_base[v] + 1) hasta pos_in_base[u]
    int start_pos = pos_in_base[v] + 1; // Arista inmediatamente después de v
    int end_pos = pos_in_base[u];

    if (start_pos > end_pos)
        swap(start_pos, end_pos);

    if (is_query) {
        Node current_res = query_st(1, 1, N_nodes - 1, start_pos, end_pos);
        total_result.count_true += current_res.count_true;
        total_result.length += current_res.length;
    }
    else {
        update_range_st(1, 1, N_nodes - 1, start_pos, end_pos);
    }

    return total_result;
}

// ========================================================
// 3. Operaciones del Problema
// ========================================================
void toggle_action(int x, int y) {
    perform_path_op(x, y, false);
}

bool exists_query(int x, int y) {
    Node result = perform_path_op(x, y, true);
    // Si el conteo de aristas 'true' es > 0, al menos una existe
    return result.count_true > 0;
}

bool forall_query(int x, int y) {
    Node result = perform_path_op(x, y, true);
    // Si el conteo de aristas 'true' es igual a la longitud total, todas cumplen
    return result.count_true == result.length && result.length > 0;
}
```

Pregunta 3

La solución para esta consulta de selección de rango se basa en el uso de **Árboles de Segmentos Persistentes (Persistent Segment Trees)**.

La solución transforma el problema de la selección del k-ésimo elemento de un rango en un problema de conteo de frecuencias, y utiliza la persistencia para manejar la dimensión de los rangos.

Para un rango fijo $A[i..j]$, si conociéramos cuántas veces aparece cada valor en ese rango, podríamos responder la consulta. Por ejemplo, para saber si el k-ésimo elemento es menor o igual a un valor v, simplemente contamos cuántos elementos en $A[i..j]$ son $\leq v$. Si este conteo es $\geq k$, sabemos que el k-ésimo elemento debe ser v o menor. Esta idea transforma el problema en una **búsqueda binaria sobre los *valores*** posibles.

Si tenemos un arreglo de frecuencias de valores $F[1..N]$ para el rango $A[i..j]$, podemos calcular la frecuencia de un sub-rango de valores $F[v_1..v_2]$ eficientemente. Para manejar la dimensión del rango de *índices* $[i..j]$ de la entrada, utilizamos el concepto de **arreglos cumulativos**. Si C_x es la estructura que almacena la información de *ocurrencias* de valores en el prefijo $A[1..x]$, entonces la información de ocurrencias para el rango $A[i..j]$ se obtiene restando las estructuras: $C_j - C_{i-1}$.

Para manejar eficientemente la dimensión de los valores (que van de 1 a N) y la dimensión de los prefijos del arreglo (que van de 1 a N), se utiliza un Árbol de Segmentos (que opera sobre los *valores*) y se hace persistente a lo largo de los índices del arreglo de entrada.

1. **Construcción Persistente ($O(N \log N)$):**
   - Construimos el arreglo de entrada A en orden, del índice 1 al N.
   - Para cada índice i, creamos una nueva versión del Árbol de Segmentos, R_i, que representa las frecuencias de los valores en el prefijo $A[1..i]$.
   - La versión R_i es una *copia parcial* de la versión anterior R_{i-1}, modificando únicamente el camino de $O(\log N)$ nodos afectados por la adición del valor $A[i]$.

   La complejidad de memoria de esta construcción es $O(N \log N)$, ya que cada una de las N actualizaciones solo crea $O(\log N)$ nodos nuevos. El tiempo de pre-condicionamiento es también $O(N \log N)$.

2. **Consulta ($O(\log N)$):** Para una consulta seleccion(i, j, k):
   - Obtenemos las raíces de las estructuras persistentes: R_j (que contiene las frecuencias hasta el índice j) y R_{i-1} (que contiene las frecuencias hasta el índice $i-1$).
   - Realizamos una **búsqueda binaria sobre el *valor*** que es el k-ésimo menor. En lugar de hacer una búsqueda binaria explícita, se usa la estructura del árbol de segmentos persistente para *contar* cuántos elementos hay en la mitad izquierda del rango de valores:
     - Restamos los valores acumulados en el hijo izquierdo de R_j con el hijo izquierdo de R_{i-1}. Esto nos da la cantidad total de ocurrencias de valores en el rango $[i..j]$ que caen en la primera mitad del rango de valores posibles.
     - Si este conteo es $\geq k$, el k-ésimo elemento está en la mitad izquierda. Recurrimos a esa rama (el nuevo rango de valores es la mitad izquierda).
     - Si el conteo es $< k$, restamos este conteo a k y recurrimos a la mitad derecha del rango de valores.

Este proceso recursivo en el árbol de segmentos persiste a través de $O(\log N)$ niveles, ya que la altura del árbol de segmentos es logarítmica. Por lo tanto, cada consulta toma $O(\log N)$ tiempo.

La técnica de copia de caminos asegura que la adición de cada nuevo elemento $A[i]$ al arreglo (creando una nueva versión R_i) solo requiera $O(\log N)$ tiempo y memoria adicional. Esto cumple con las restricciones de memoria $O(N \log N)$ y permite que el tiempo total de Q consultas sea $O((N+Q) \log N)$, cumpliendo con la complejidad asintótica solicitada.

Implementación en C++

El archivo funcional se encuentra en persistent_seg_trees.cpp

```cpp
// Definiciones de constantes
const int MAX_NODES = 200000 * 20; // N * log N * 4 para nodos del ST Persistente
const int MAX_N_ARR = 200000 + 5;

// ========================================================
// 1. Segment Tree Persistente (PST)
// ========================================================

// Estructura de un nodo del Árbol de Segmentos Persistente
// Solo necesitamos almacenar el conteo total de elementos en el subárbol
struct Node {
    int count;                  // Frecuencia de valores en este rango [L, R]
    int left_child, right_child; // Índice del nodo hijo en el array global
};

Node tree[MAX_NODES];
int root[MAX_N_ARR]; // Array de raíces: root[i] almacena la raíz de la versión para A[1..i]
int node_counter;    // Contador global para asignar índices de nodos

// Rango de valores (después de la compresión de coordenadas)
int N_compressed;
// Vector global para la descompresión (índice comprimido -> valor original)
vector<int> decompressed_values;

// Función para construir el Segment Tree inicial (versión 0, vacío)
void build_pst(int p, int L, int R) {
    if (L == R) {
        // Hoja: representa un valor único
        tree[p].count = 0;
        return;
    }
    int mid = L + (R - L) / 2;
    tree[p].left_child = ++node_counter;
    tree[p].right_child = ++node_counter;
    build_pst(tree[p].left_child, L, mid);
    build_pst(tree[p].right_child, mid + 1, R);
    tree[p].count = 0;
}

// Función para actualizar el Segment Tree (añadir una ocurrencia) y hacerlo persistente
// Toma la raíz de la versión anterior (prev_root) y el valor a insertar (val)
int update_pst(int prev_root, int L, int R, int val) {
    // 1. Creamos un nuevo nodo (copia de camino)
    int new_root = ++node_counter;
    tree[new_root] = tree[prev_root];
    tree[new_root].count++; // Aumentamos la cuenta en el nuevo nodo

    if (L == R) {
        return new_root;
    }

    int mid = L + (R - L) / 2;

    // Si el valor está a la izquierda
    if (val <= mid) {
        // Copiamos el hijo derecho sin cambios (compartición de nodos)
        tree[new_root].right_child = tree[prev_root].right_child;

        // Actualizamos recursivamente el hijo izquierdo
```

```
        tree[new_root].left_child = update_pst(tree[prev_root].left_child, L, mid, val)
    }
    // Si el valor está a la derecha
    else {
        // Copiamos el hijo izquierdo sin cambios
        tree[new_root].left_child = tree[prev_root].left_child;

        // Actualizamos recursivamente el hijo derecho
        tree[new_root].right_child = update_pst(tree[prev_root].right_child, mid + 1, R
    }

    return new_root;
}

// Función de consulta: busca el k-ésimo elemento en el rango virtual [root_j] - [root_i_minus
// Devuelve el valor (índice comprimido) del k-ésimo elemento
int query_kth(int root_j, int root_i_minus_1, int L, int R, int k) {
    if (L == R)
        return L;

    // Calculamos la diferencia de frecuencias en el hijo izquierdo (rango [L, mid] de valores
    // count_L_j = frecuencia de valores en [L, mid] en el prefijo A[1..j]
    int count_L_j = tree[tree[root_j].left_child].count;

    // count_L_i_minus_1 = frecuencia de valores en [L, mid] en el prefijo A[1..i-1]
    int count_L_i_minus_1 = tree[tree[root_i_minus_1].left_child].count;

    // count_L_range = Frecuencia de valores en [L, mid] para el rango A[i..j]
    int count_L_range = count_L_j - count_L_i_minus_1;

    int mid = L + (R - L) / 2;

    if (k <= count_L_range) {
        // El k-ésimo elemento se encuentra en el rango de valores izquierdo [L, mid]
        return query_kth(tree[root_j].left_child, tree[root_i_minus_1].left_child, L, m
    } else {
        // El k-ésimo elemento se encuentra en el rango de valores derecho [mid+1, R]
        // Ajustamos k restando los elementos que ya contamos en el lado izquierdo.
        return query_kth(tree[root_j].right_child, tree[root_i_minus_1].right_child, mi
    }
}

// ============================================================
// 2. Lógica de Compresión y Preprocesamiento
// ============================================================

// Almacenamiento del arreglo original
int A_original[MAX_N_ARR];
int N_arr;

// Función que maneja la compresión de coordenadas y la construcción del PST
void preprocess_data() {
    // Los valores de A son la clave para la compresión
    vector<int> sorted_values(A_original + 1, A_original + N_arr + 1);

    // 1. Ordenar y obtener valores únicos
    sort(sorted_values.begin(), sorted_values.end());
    sorted_values.erase(unique(sorted_values.begin(), sorted_values.end()), sorted_valu
    N_compressed = sorted_values.size();
```

```
119
120     // Guardamos los valores ordenados para la descompresión global
121     decompressed_values = sorted_values;
122
123     // 2. Inicialización del PST (versión 0 -> root[0])
124     // node_counter empieza en 0; el primer nodo será 1
125     node_counter = 0;
126     root[0] = ++node_counter;
127     build_pst(root[0], 1, N_compressed);
128
129     // 3. Construir N versiones persistentes (O(N log N))
130     for (int i = 1; i <= N_arr; ++i) {
131         // Encontramos el índice comprimido del valor A_original[i]
132         int val_compressed = lower_bound(sorted_values.begin(), sorted_values.end(), A_
133
134         // Creamos la versión R_i a partir de R_{i-1}
135         root[i] = update_pst(root[i - 1], 1, N_compressed, val_compressed);
136     }
137 }
138
139 // ============================================================
140 // 3. Función Main: Interfaz Metodológica
141 // ============================================================
142
143 // Función auxiliar para obtener el k-ésimo valor
144 int get_kth_element(int i, int j, int k) {
145     if (i > j || k <= 0)
146         return -1; // Error o rango inválido
147
148     // Los índices i y j en la consulta son 1-based.
149
150     // 1. Obtener los rangos de las raíces persistentes
151     int root_j = root[j];
152     int root_i_minus_1 = root[i - 1];
153
154     // 2. Realizar la consulta O(log N)
155     int compressed_rank = query_kth(root_j, root_i_minus_1, 1, N_compressed, k);
156
157     // 3. Descompresión de Coordenadas
158     // Usamos el vector global 'decompressed_values' creado en preprocess_data
159     if (compressed_rank < 1 || compressed_rank > (int)decompressed_values.size())
160         return -1;
161     return decompressed_values[compressed_rank - 1]; // compressed_rank es 1-based
162 }
```