Segment tree

June 1, 2018

Problema

Dado un arreglo A de enteros, se desea realizar Q queries de la forma "t x y", tal que:

- $\underline{t = 1:}$ realizar la asignación $A[x] = y$.
- $\underline{t = 2:}$ computar $\sum_{i=x}^{y} A[i]$

Problema

Dado un arreglo A de enteros, se desea realizar Q queries de la forma "t x y", tal que:

- $\underline{t = 1:}$ realizar la asignación $A[x] = y$.
- $\underline{t = 2:}$ computar $\sum_{i=x}^{y} A[i]$

¿Se les ocurren algunas ideas?

Segment tree

- Introducimos al curso una nueva estructura de datos que es útil es muchísimos problemas: segment tree. En particular, puede realizar todo lo necesario para resolver nuestro problema actual.
- Construirlo tiene complejidad $O(n)$.
- Realizar updates de elementos tiene complejidad $O(log(n))$.
- Realizar consultas en rangos tiene complejidad $O(log(n))$.

Segment tree

- Introducimos al curso una nueva estructura de datos que es útil es muchísimos problemas: segment tree. En particular, puede realizar todo lo necesario para resolver nuestro problema actual.
- Construirlo tiene complejidad $O(n)$.
- Realizar updates de elementos tiene complejidad $O(log(n))$.
- Realizar consultas en rangos tiene complejidad $O(log(n))$.

- Antes de ver en profundidad las operaciones de update/query, veamos qué es y cómo se representa un segment tree.

Estructura de un segment tree

- Un segment tree es un arbol binario completo en el cual sus nodos guardan el valor correspondiente a el cómputo de una operación **asociativa** determinada sobre intervalos o segmentos de un arreglo de tamaño N. Se debe cumplir que esta operación tenga un neutro.
- Un segment tree puede contener otras cosas que no sean simples valores en sus nodos, pero en esta clase nos enfocaremos en los que contienen números únicamente.
- La raiz del arbol representa el intervalo $[0, N-1]$.
- Las hojas del arbol representan los elementos del arreglo (por lo que hay N hojas).
- Los nodos intermedios representan intervalos de la forma $[i, j]$, donde $0 \leqslant i < j \leqslant N$.

Construcción de un segment tree

- El intervalo de la raiz ($[0, N-1]$) se divide en dos mitades que serán representadas por los hijos del nodo root.
- A su vez, los intervalos de sus dos hijos también se parten en dos mitades representadas por sus respectivos hijos.
- Este proceso se repite recursivamente hasta llegar a un nodo hoja (donde el intervalo representado es [i, i] tal que $0 \leqslant i < N$).
- Una vez que se encuentra una hoja, se le asigna como valor $A[i]$ y se devuelve este numero como resultado de la función.
- En el valor del nodo padre se guarda el resultado de aplicar la operación deseada sobre los valores retornados por sus hijos. Luego se devuelve este valor (para continuar con la recursión).

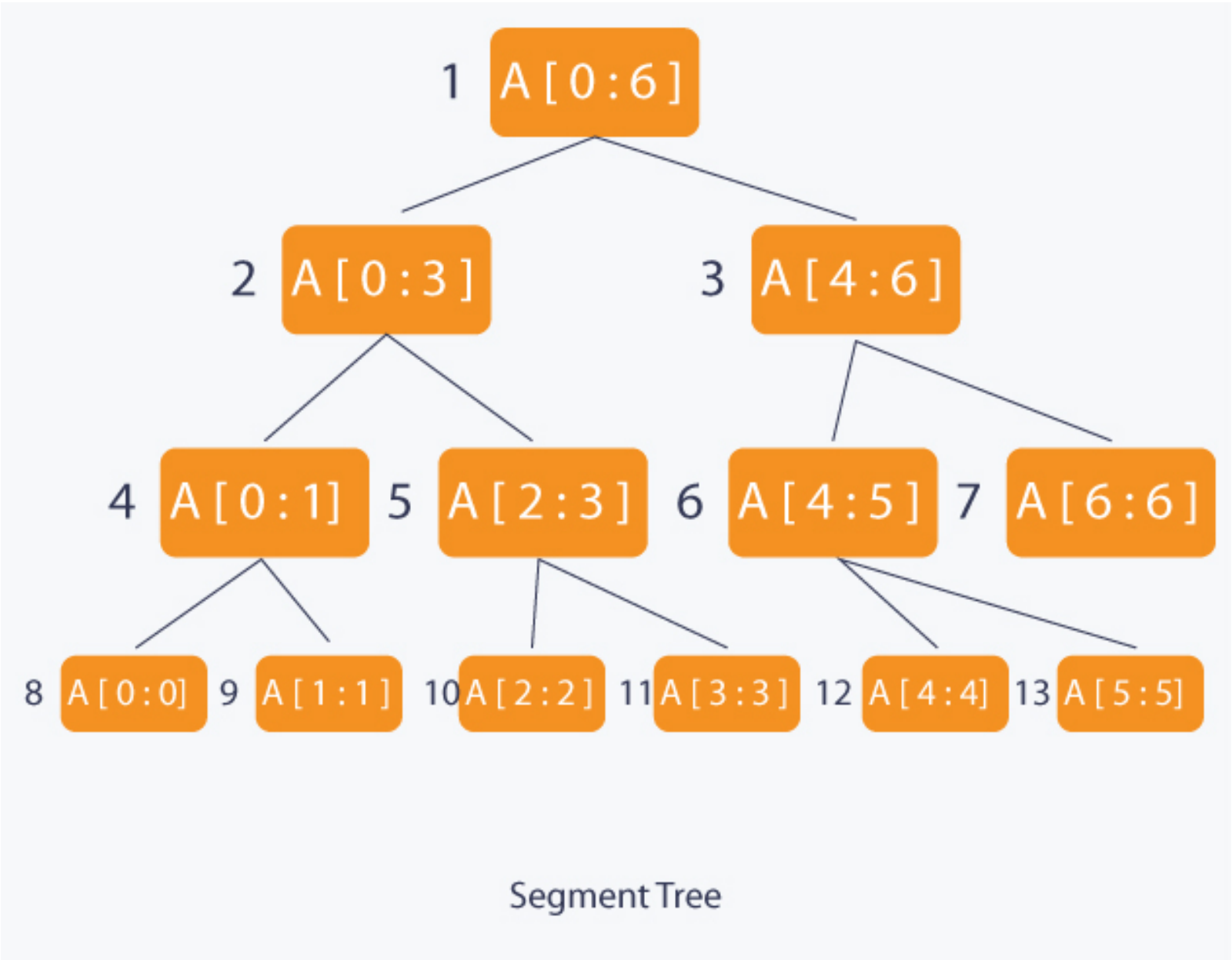
Construcción de un segment tree

- Luego de finalizada la recursión, cada nodo va a contener el valor de aplicar la operación elegida a su intervalo correspondiente.
- Pero... ¿Cuántos niveles tiene el árbol?

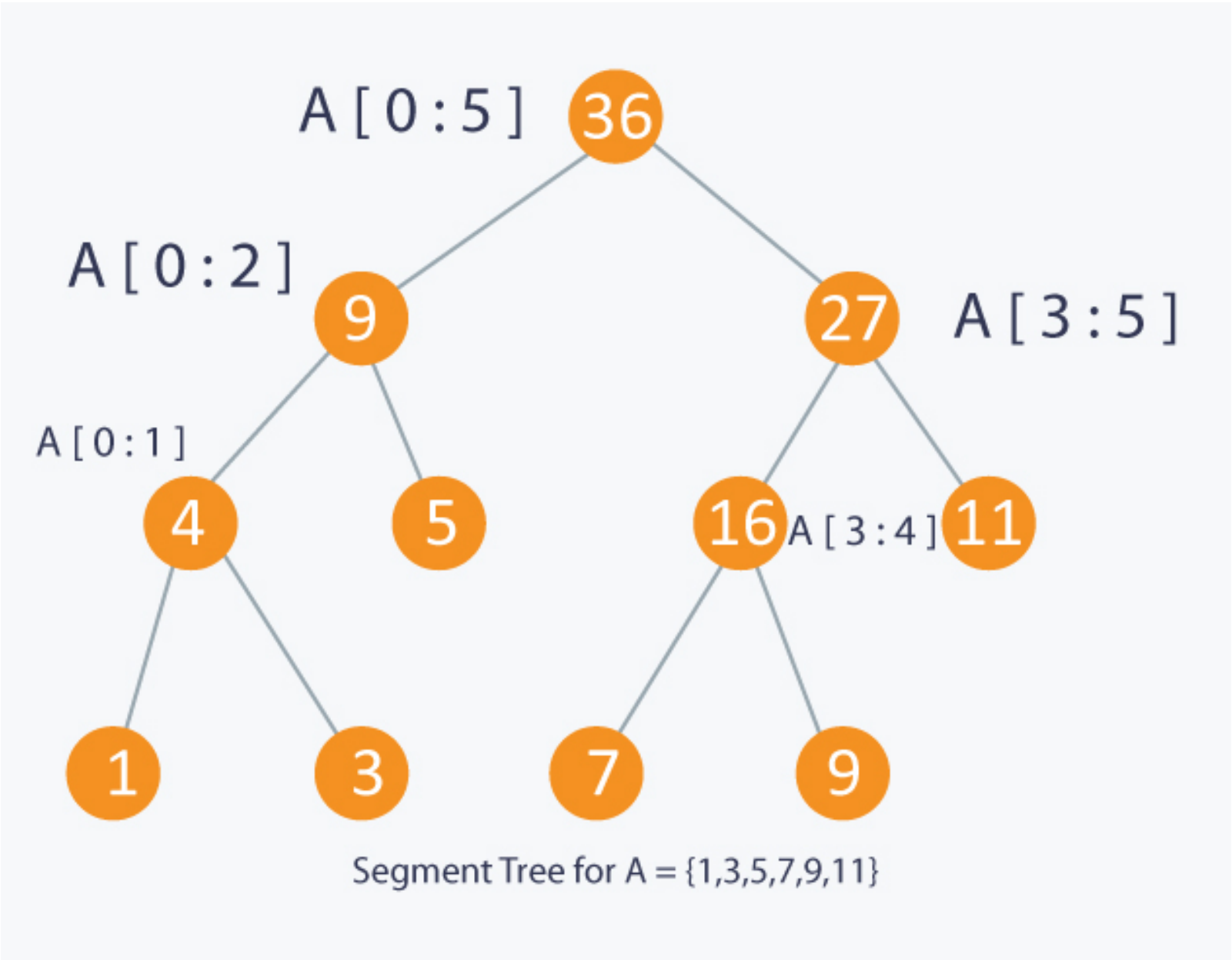
Construcción de un segment tree

- Luego de finalizada la recursión, cada nodo va a contener el valor de aplicar la operación elegida a su intervalo correspondiente.
- Pero... ¿Cuántos niveles tiene el árbol?
- Como en cada paso, se divide a un intevalo a la mitad, a lo sumo en $log(N)$ pasos me voy a encontrar en una hoja, por lo que la cantidad de niveles del arbol es $log(N)$.

Ejemplo


1 A[0:6]
2 A[0:3]
3 A[4:6]
4 A[0:1]
5 A[2:3]
6 A[4:5]
7 A[6:6]
8 A[0:0]
9 A[1:1]
10 A[2:2]
11 A[3:3]
12 A[4:4]
13 A[5:5]
Segment Tree

Ejemplo



Segment Tree for A = {1,3,5,7,9,11}

Implementación

- Pero, ¿Cómo implementar este arbol de forma rápida y eficiente?
- Podemos verlo como un arreglo, donde la raiz es la posición 1, y los hijos del nodo i se ubican en las posiciones $2 * i$, y $2 * i + 1$.
- ¿Y qué tamaño debería tener el arreglo?
- El suficiente para que todos los nodos que representen los intervalos necesarios tengan un espacio reservado.
- La cantidad de nodos nunca supera $4 * N$, por lo que esa es una buena cota para el tamaño.

Construcción - Código

```
1  int a[N], tree[4*N];
2
3  // La operacion asociativa que tendra el segment tree.
   int op(int a, int b){
5    return a+b;
   }
7
   void build(int node, int start, int end){
9    if(start == end){              // El nodo es una hoja
       tree[node] = a[start-1];
11   }
     else{
13     int mid = (start+end)/2;  // Partir al medio el intervalo
       build(2*node, start, mid);
15     build(2*node+1, mid+1, end);
       tree[node] = op(tree[2*node], tree[2*node+1]);
17   }
   }
19 // Llamada a la funcion
   build(1, 1, N)
```

Updates en un segment tree

- Al realizar un update del valor de una posición, la idea es sólo realizarlo en los intervalos que lo contengan.
- ¿Y cómo hacemos eso?

Updates en un segment tree

- Al realizar un update del valor de una posición, la idea es sólo realizarlo en los intervalos que lo contengan.
- ¿Y cómo hacemos eso?
- El intervalo de la raíz siempre va a contener al indice que se desea actualizar, por lo que podemos comenzar actualizando desde ahí, primero recursionando sobre los hijos, y luego recomputando el valor del nodo actual en base a los nuevos valores de ellos.
- El mismo procedimiento se realiza para los siguientes nodos. Cuando llegamos a una hoja, simplemente cambiamos su valor previo por el que se desea actualizar y se retorna el valor updateado (exactamente como en la construcción).

Updates en un segment tree

- Pero... llamar a la recursión sobre los hijos de cada nodo implicaría recorrer todos los nodos en cada update, y como hay a lo sumo 4*n, la operación de actualización sería de complejidad $O(N)$.
- Pero pensemos un poco. Cuando estamos en la raiz y llamamos a la recursión sobre los hijos, ¿Hace falta recursionar sobre ambos?

Updates en un segment tree

- Pero... llamar a la recursión sobre los hijos de cada nodo implicaría recorrer todos los nodos en cada update, y como hay a lo sumo 4*n, la operación de actualización sería de complejidad $O(N)$.
- Pero pensemos un poco. Cuando estamos en la raiz y llamamos a la recursión sobre los hijos, ¿Hace falta recursionar sobre ambos?
- No. Ambos intervalos son disjuntos (son las dos mitades del intervalo representado por el nodo actual). Esto nos dice que solo uno de ellos va a contener al índice que quiero actualizar, por lo que basta con llamar a la recursión sobre ese nodo (el rango representado por el otro no se va a ver afectado por la actualización).
- Luego vemos que se visita a lo sumo un nodo por nivel del árbol. Pero habíamos dicho que éste tenía profundidad a lo sumo $log(N)$, por lo que la operación de update es $O(log(N))$.

Actualización - Código

```
void update(int pos, int value, int node, int start, int end)
    {
  int mid = (start+end)/2;
  if(start == end){          // Estoy en una hoja
    tree[node] = value;
    return;
  }
  // El indice esta en el hijo izquierdo
  if(start <= pos && pos <= mid){
    update(pos, value, 2*node, start, mid);
  }
  // El indice esta en el hijo derecho
  else{
    update(pos, value, 2*node+1, mid+1, end);
  }
  // Actualizo valor del nodo actual con el de sus hijos
  tree[node] = op(tree[2*node], tree[2*node+1]);
}
update(x, y, 1, 1, N)    // Update A[x] = y
```

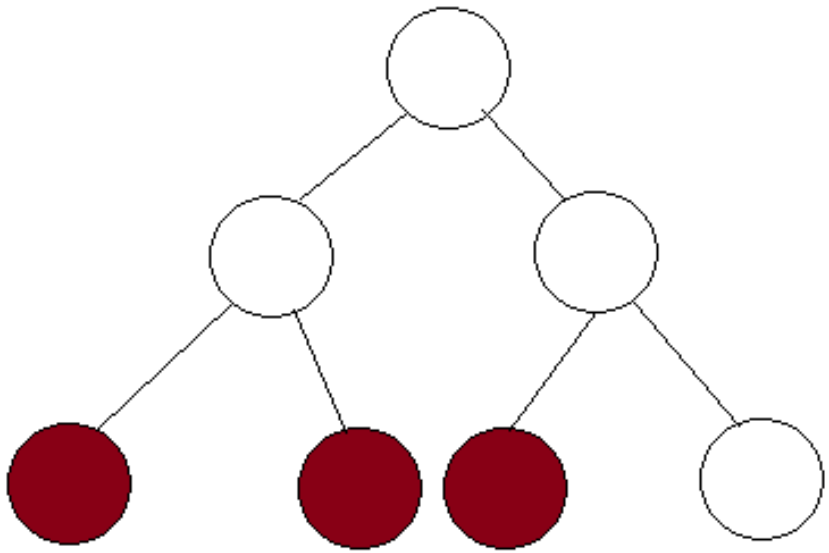
Queries en un segment tree

Al realizar una consulta sobre un rango, hay 3 posibles casos que pueden ocurrir dado que estoy en un nodo:

- El rango del nodo está incluído completamente en el de la query: se devuelve el valor del nodo.
- El rango del nodo está completamente afuera del de la query: se devuelve el elemento neutro de la operación.
- El rango del nodo está incluído parcialmente en el de la query: se recursiona sobre los hijos del nodo y se devuelve la operacion aplicada a los resultados de ambos hijos.

Queries en un segment tree

- ¿Qué complejidad tiene la query sobre el segment tree?
- Notemos que en cada nivel del arbol se expanden a lo sumo dos nodos. Podemos ver que es absurdo suponer que se expanden 3 o más:



- Entonces se realizan a lo sumo $2 * log(n)$ operaciones, por lo que la complejidad de la query es $O(log(n))$.

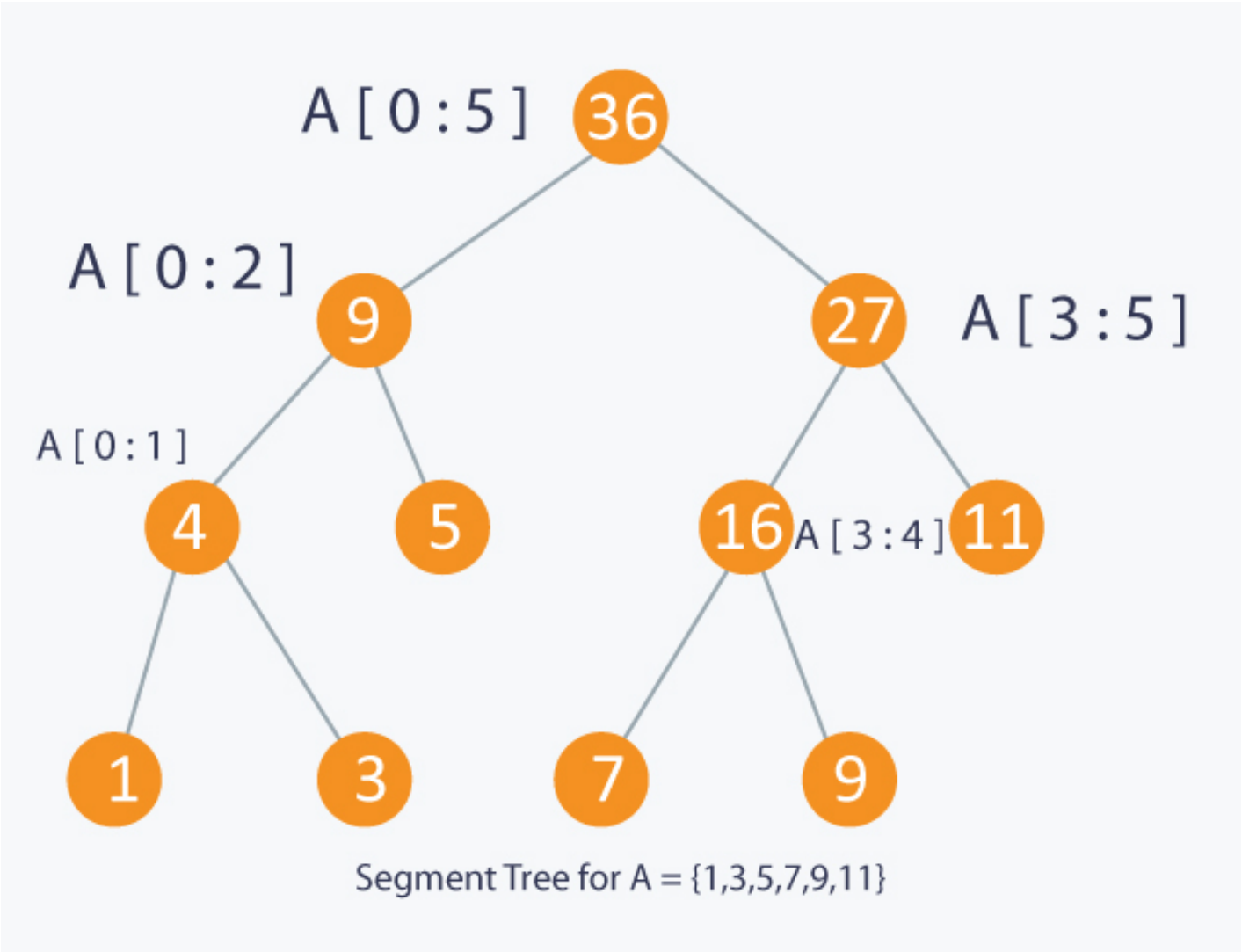
Query - Código

```
int query(int l, int r, int node, int start, int end){
  // Intervalo del nodo completamente afuera
  if(start > r || end  < l){
    return OP_NEUTRO;
  }
  // Intervalo del nodo completamente adentro
  if(l <= start && end <= r){
    return tree[node];
  }
  // Recursion sobre los hijos
  int mid = (start+end)/2;
  return op(query(l, r, 2*node, start, mid),
            query(l, r, 2*node+1, mid+1, end));
}

query(l, r, 1, 1, n);      // Query sobre el rango [l, r]
```

Ejemplo

Observemos el comportamiento de las operaciones sobre el segment tree dado como ejemplo:



Problemas para práctica

- **(Importante para probar de que su segment tree funciona)** `https://www.hackerearth.com/practice/data-structures/advanced-data-structures/segment-trees/tutorial/`
- `http://www.spoj.com/problems/DQUERY/`

Problema - www.spoj.com/problems/DQUERY/

Enunciado

Dado un arreglo de n (hasta 30000) números, responder hasta 200000 preguntas de la forma "¿Cuántos números distintos hay entre la posición l y la posición r?" ($1 \leq l \leq r \leq n$).

```
A = [1, 1, 2, 1, 3]
Queries:
[1, 5] -> 3
[2, 4] -> 2
[3, 5] -> 3
[1, 2] -> 1
```

DQUERY - Solución

- Resolvemos las queries en otro orden: de menor a mayor según r.
- Recorremos el arreglo de izquierda a derecha, manteniendo un segment tree con suma, que tenga 1 en la última ocurrencia de cada valor. 0 caso contrario.
- Cuando procesamos i, hacemos st[i] $= 1$, st[j] $= 0$, donde j es la última aparición de A[i] (si es que existe).
- Si luego de procesar i alguna query tiene $r = i$, entonces la suma entre l y r es la respuesta de la query.

DQUERY - Ejemplo

```
A = [1,1,2,1,3]
Queries (ordenadas): [1,2],[2,4],[1,5],[3,5]
ST (incial): [0,0,0,0,0]
```

DQUERY - Ejemplo

```
A = [1,1,2,1,3]
Queries (ordenadas): [1,2],[2,4],[1,5],[3,5]
ST (incial): [0,0,0,0,0]

[1,0,0,0,0]
```

DQUERY - Ejemplo

```
A = [1,1,2,1,3]
Queries (ordenadas): [1,2],[2,4],[1,5],[3,5]
ST (incial): [0,0,0,0,0]

[1,0,0,0,0]

[0,1,0,0,0] (query [1,2] -> 1)
```

DQUERY - Ejemplo

```
A = [1,1,2,1,3]
Queries (ordenadas): [1,2],[2,4],[1,5],[3,5]
ST (incial): [0,0,0,0,0]

[1,0,0,0,0]

[0,1,0,0,0] (query [1,2] -> 1)

[0,1,1,0,0]
```

DQUERY - Ejemplo

```
A = [1,1,2,1,3]
Queries (ordenadas): [1,2],[2,4],[1,5],[3,5]
ST (incial): [0,0,0,0,0]

[1,0,0,0,0]

[0,1,0,0,0] (query [1,2] -> 1)

[0,1,1,0,0]

[0,0,1,1,0] (query [2,4] -> 2)
```

DQUERY - Ejemplo

```
A = [1,1,2,1,3]
Queries (ordenadas): [1,2],[2,4],[1,5],[3,5]
ST (incial): [0,0,0,0,0]

[1,0,0,0,0]

[0,1,0,0,0] (query [1,2] -> 1)

[0,1,1,0,0]

[0,0,1,1,0] (query [2,4] -> 2)

[0,0,1,1,1] (query [1,5] -> 3, [3,5] -> 3)
```

Problema - Longest Increasing Subsequence

Enunciado

Dado una secuencia (arreglo) de n enteros, encontrar su subsecuencia creciente más larga en $O(nlog(n))$.
(Una subsecuencia es el resultado de eliminar algunos elementos de la secuencia)
Por ejemplo, la subsecuencia creciente mas larga de [3,2,3,5,1,2,5,7] es [2,3,5,7].

Pensemos primero el problema cuando los números de la secuencia son chicos (entre 1 y n).

LIS - solución

- Podemos recorrer la secuencia de izquierda a derecha, manteniendo para cada valor v, la longitud de la máxima subsecuencia que termina en valor v, en un arreglo L.
- Inicialmente $L[v] = 0$ para todo v.
- Cuando proceso el elemento a_i actualizo $L[a_i] = 1 + max(L[j] : j < a_i)$.
- Para calcular $max(L[j]) : j < a_i$ puedo usar segment tree, con lo que el algoritmo queda $O(nlog(n))$.

LIS - valores grandes

¿Qué pasa si los valores del arreglo no están acotados, sino que pueden ser cualquier cosa?

LIS - valores grandes

¿Qué pasa si los valores del arreglo no están acotados, sino que pueden ser cualquier cosa?

- Compresión de coordenadas: reemplazo los valores por índices en $[1, ..., n]$ respetando orden (puedo hacerlo porque tengo a lo sumo n valores distintos).
- Se puede hacer ordenando los valores y eliminando repeticiones, o con un set.
- Para mapear un valor a su valor comprimido, puedo usar binary search, o map.
- Una vez que mapeé todos los valores a su valor comprimido, puedo usar el approach anterior.