DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Range Queries Con Updates

Segment Tree y otras variantes

5 de noviembre de 2023 — Programación Competitiva II

Integrante	LU	Correo electrónico
López Pacholczak, Ulises	026/23	ulilopezpacho@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

Un problema muy interesante en las ciencias de la computación es como poder realizar consultas sobre un arreglo de información de forma eficiente. Por ejemplo, se puede utilizar la técnica de "Suma de Prefijos" para poder, con un preprocesamiento de $O(n)$, hacer queries de suma en un rango $[l, r]$ en $O(1)$. O se pueden usar las "Sparse Tables" para realizar queries de máximo y mínimo en $O(1)$ con preprocesamiento en $O(n \log n)$

De todas formas, en muchos casos, es necesario ir haciendo modificaciones sobre el arreglo donde se desean hacer las consultas. En esta monografía trabajaremos sobre una estructura de datos llamada "Segment Tree" que nos permitirá hacer consultas y actualizaciones en rango de forma eficiente sobre distintas operaciones (suma, máximo, mínimo, entre otras). Además, trabajaremos sobre distintas variantes de esta estructura que nos permitirán obtener ciertas flexibilidades para algunas consultas y actualizaciones.

2. Segment Tree

2.1. Operaciones de la Estructura

El "Segment Tree" es una estructura de datos que nos permitirá realizar queries en rangos y updates en $O(\log n)$ con solo $O(n)$ memoria.

Esta estructura de datos soporta dos operaciones principales:

- *query(l: int, r: int): T*, que dado un rango, nos permitirá obtener el resultado de aplicar una cierta operación en el rango $[l, r]$
- *update(pos: int, val: T)* que actualizará el valor de la posición *pos* al indicado en *val*.

Ambas operaciones, como fue mencionado anteriormente, tendrán complejidad $O(\log n)$. Además, se puede proponer la operación *build(l: seq<T>)*, que dada un arreglo, construye un Segment Tree en complejidad $O(n)$. De todas formas, esta no suele ser necesaria, ya que se podría construir el mismo inicializándolo con un Segment Tree vacío, y luego aplicando la operación *update*. Lo que sí es necesario es definir n, la cantidad de elementos que tendrá el arreglo, previo a realizar las operaciones, ya que se reservará la memoria al inicializar la estructura.

2.2. Monoides y Segment Tree

Los "Segment Trees" soportan cualquier Monoide. Esto quiere decir que, cualquier estructura algebraica que sea un Monoide podrá ser utilizada con un "Segment Tree" para poder realizar queries y updates.

Un Monoide es una estructura algebraica con una operación binaria, que es asociativa y tiene elemento neutro. Un Monoide se lo puede denotar como $(A, \oplus)$ y, en lenguaje matemático formal, cumple las siguientes propiedades:

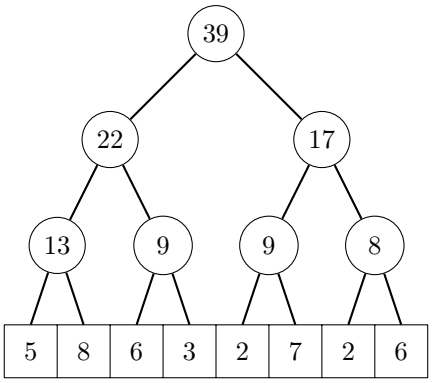
- $\forall a, b, c \in A, (a \oplus b) \oplus c = a \oplus (b \oplus c)$ (es asociativa)
- $\exists! e \in A, \forall a \in A, a \oplus e = e \oplus a = a$ (existe neutro)
- $\forall a, b \in A, a \oplus b \in A$ (operación cerrada)

Entre las operaciones interesantes que pueden utilizarse en un Segment Tree son la suma $(\mathbb{N}, +)$, el máximo $(\mathbb{N}, \text{máx})$, el mínimo $(\mathbb{N}, \text{mín})$, el máximo común divisor $(\mathbb{N}, \text{gcd})$ entre otras.

2.3. Representación de la Estructura

El "Segment Tree" es representado en memoria como un Árbol Binario de $2n-1$ nodos. Para el valor n, debe tomarse un número tal que, si llamamos l a la cantidad de elementos del arreglo sobre el cual se desean hacer las consultas, $n = 2^k, k \in \mathbb{N}$, tal que $l \leq 2^k$. De esta forma, el árbol binario representado será completo, lo que nos facilitará guardarlo en un arreglo de tal forma que los hijos del nodo k sean $2k$ el hijo izquierdo y $2k + 1$ el hijo derecho, siempre que $1 \leq k < n$. Esto se debe a que, para los nodos entre $n \leq k < 2n$ no tendrán hijos.

Por ejemplo, un "Segment Tree" de la siguiente forma (es de suma, pero se puede usar cualquier Monoide):



Será representado en memoria de la siguiente forma:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
39	22	17	13	9	9	8	5	8	6	3	2	7	2	6

Como invariante de representación, los nodos entre $n \leq k < 2n$ serán los nodos originales del arreglo. Esto quiere decir que cuando se llame al procedimiento *update* se actualizará inicialmente el valor de la posición en el arreglo original i en la posición del "Segment Tree" $n + i$. Luego, para el resto de los nodos $1 \leq k < n$, el nodo k contiene el valor de aplicar la operación para todos sus hijos. Es decir, el nodo k tendrá el valor de aplicar la operación en los nodos $2k$ y $2k + 1$, luego el nodo $2k$ las aplicará en los nodos $4k$ y $4k + 1$, hasta llegar a la profundiad máxima del arbol. De esta forma, cada nodo guardará los valores de aplicar la operación en un rango $[l, r]$ del arreglo original, siendo ese rango al conjunto de elementos que tiene como hijos (directos o indirectos) en la última capa del arbol, ya que esta equivale al arreglo original.

Para verlo en un ejemplo, en la figura anterior (que es un "Segment Tree" con la operación suma), el nodo número 3, con valor 17, contiene la suma de sus dos nodos hijos (nodos 6 y 7 con valores 9 y 8 respectivamente), y representa la suma en el rango del arreglo original $[4, 7]$, ya que contiene como hijos en la última capa a los nodos 12, 13, 14 y 15 equivalentes en el arreglo original (restando n se obtiene la equivalencia) a las posiciones 4, 5, 6 y 7 respectivamente.

2.4. Las Operciones

Para implementar las operaciones, se puede hacer tanto con un enfoque iterativo o recursivo. El enfoque recursivo suele ser mucho más claro, aunque el enfoque iterativo suele ser más útil y fácil de programar en competencias. Por eso, comencemos describiendo el algoritmo recursivo, y veamos su código. Trabajaremos sobre el caso de suma, pero se puede fácilmente generalizar para cualquier otro Monoide.

2.4.1. Enfoque Recusivo

Para obtener la suma dado un "Segment Tree" válido, definiremos una función auxiliar *query(a: int, b: int, k: int, x: int, y: int)* que nos permitirá obtener la suma en el rango. a y b son el rango de la forma inclusiva-inclusiva sobre el cual nosotros queremos hacer la consulta. Luego, k indica el nodo sobre el que estamos parados, y x e y siempre van a mantener el rango que abarca a ese nodo. Incialmente, por eso, la función siempre va a ser llamada con los parámetros *query(l, r, 1, 0, n-1)*.

La función recursiva, entonces, evalúa los siguientes casos:

- Si el rango está fuera del intervalo que representa ese nodo es vacío (es decir, el rango $[a, b] \cap [x, y] = \emptyset$), se debe devolver el elemento neutro. En caso de la suma, el 0.
- Si el rango está totalmente contenido dentro del intervalo, debemos retornar el valor en $segTree[k]$, ya que ahí se encuentra la suma del rango $[x, y]$.
- Si el rango está parcialmente contenido, entonces se recurre a los dos subárboles izquierdo y derecho. Esto se repetirá hasta caer en alguno de los dos casos anteriores, que siempre ocurren al llegar al último piso del árbol.

Esto se puede implementar de la siguiente forma:

```
int query(int a, int b, int k, int x, int y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return segTree[k];
    int d = (x+y)/2;
    return query(a,b,2*k,x,d) + query(a,b,2*k+1,d+1,y);
}
```

Puede pensarse que esto puede tener una complejidad de $O(n)$, ya que se podría llamar muchas veces al subarbol de forma recursiva. Pero no es así, ya que, en el peor de los casos, se puede ver que para cada capa del Arbol Binario a lo sumo se van a tener cuatro llamadas a la función *query*. Esto se debe a que el intervalo sobre el cual uno hace las preguntas es continuo. Por ende, como el Arbol Binario tiene profundidad $\log_2 n$, la complejidad de este algoritmo será $O(\log n)$.

Ahora, vayamos a analizar la operación de *update*. Al igual que en el caso anterior, nombraremos una función auxiliar *update(pos: int, k: int, x: int, y: int, val: int)* que indicará en *pos* la posición del arreglo original que será modificada, k, x e y representan lo mismo que en el caso anterior (el nodo actual, el principio y el final del rango que representa ese nodo respectivamente), y por último *val*, que es el nuevo valor a asignar. La función se llamará inicialmente de la siguiente forma: *update(pos, 1, 0, n-1, val)*.

La función evalúa los siguientes casos:

- Si estoy en el útlimo nivel del Arbol Binario (entonces $x = y$), asigno *val* a esa posición y retorno ya que no hay más capas a ser recorridas del arbol.
- Caso contrario, reviso si *pos* está en el subarbol izquierdo o derecho, chequeando si es mayor o menor al promedio de x e y. Enonces, recurro al subarbol que corresponda, y una vez actualizado ese arbol (al salir de todas las llamadas recursivas), actualizo mi valor aplicando la operación correspondiente sobre los dos nodos hijos.

Esto se puede implementar de la siguiente forma:

```
void update(int pos, int k, int x, int y, int val) {
    if (x == y) { segTree[k] = val; return; }
    int d = (x + y) / 2;
    if (pos <= d) update(pos, k*2, x, d, val);
    else update(pos, k*2+1, d+1, y, val);
    segTree[k] = segTree[k*2] + segTree[k*2+1];
}
```

Nuevamente, la complejidad de esta operación es $O(\log n)$ ya que se recorre cada una de las capas del Arbol Binario Completo una sola vez.

2.4.2. Enfoque Iterativo

Analizaremos este enfoque de forma más breve, ya que no es tan importante comprenderlo (con el enfoque recursivo alcanzará para comprender el tema).

La operación *query(a: int, b: int)* lo que hace es ir iterando capa a capa del Arbol Binario, desde la inferior hasta la superior, siempre manteniendo a en el borde izquierdo del intervalo y b en el borde derecho del mismo. Inicialmente, se encuentra en el rango $[a + n, b + n]$ ya que de n hasta $2n$ es donde se encuentra el arreglo original. Cada paso que se mueve a una capa superior, los valores de los nodos que no pertenezcan a la capa del arbol son agregados a la suma, y ese índice a o b pasa a apuntar a otro que es candidato a estar dentro del rango. Luego, una vez que se "cruzan" los dos punteros, significa que el rango ya fue abarcado en su totalidad, por lo que en s ya se encuentra en la respuesta.

```
int query(int a, int b) {
    a += n; b += n;
    int s = 0;
    while (a <= b) {
        if (a%2 == 1) s += tree[a++];
        if (b%2 == 0) s += tree[b--];
        a /= 2; b /= 2;
    }
```

```
    return s;
}
```

La complejidad de esta operación es $O(\log n)$.

Para la operación *update(k: int, x: int)*, actualiza a la posición k del arreglo original con el valor x. Luego, itera todos los padres del nodo modificado recomputando la suma de cada uno de ellos. El código es sencillo, y es el siguiente:

```
void update(int k, int x) {
    k += n, tree[k] = x;
    for (k /= 2; k >= 1; k /= 2) {
        tree[k] = tree[2*k]+tree[2*k+1];
    }
}
```

La complejidad será $O(\log n)$ también.

2.5. Problemas Clásicos

Los siguientes problemas requieren únicamente la implementación de esta estructura de datos. El código de las soluciones se encontrará disponible en el Anexo de esta monografía.

2.5.1. CSES 1648 - Dynamic Range Sum Queries

Link al Problema: https://cses.fi/problemset/task/1648

> Dado un arreglo de n enteros, tu tarea es procesar q queries de los siguientes tipos:
>
> 1. Actualizar el valor en la posición k a un valor u
> 2. Cual es la suma de los valores en el rango $[a, b]$?
>
> Tanto q y n pueden ser como máximo $2 * 10^5$

Para resolver este problema, solo se necesita implementar la estructura de datos del "Segment Tree", ya que la suma es un monoide. Dejaré tanto la solución recursiva como la iterativa.

2.5.2. CSES 1649 - Dynamic Range Minimum Queries

Link al Problema: https://cses.fi/problemset/task/1648

> Dado un arreglo de n enteros, tu tarea es procesar q queries de los siguientes tipos:
>
> 1. Actualizar el valor en la posición k a un valor u
> 2. Cual es el mínimo de los valores en el rango $[a, b]$?
>
> Tanto q y n pueden ser como máximo $2 * 10^5$

Este problema lo resolveré utilizando un código genérico de "Segment Tree" que puede ser utilizado para cualquier Monoide. Para esto, es solo necesario redefinirse *Node* que tiene una función *identity()*, que devuelve la identidad del Monoide, y el *operator»*, que actúa al aplicar la operación entre dos nodos. En este caso, lo implementé con el mínimo, ya que es el lo que pedía el problema.

3. Técnicas Adicionales sobre Segment Tree

En esta parte aprenderemos algunas técnicas que nos permitirán aprovechar más esta estructura de datos.

3.1. Búsqueda Binaria

Además de la operación *query* que trabajamos anteriormente, podemos utilizar búsqueda binaria para poder encontrar elementos dentro del arreglo sobre el cual se hacen consultas, bajo un cierto criterio. Por ejemplo, podemos determinar la primera posición k que es menor o igual a un cierto valor x en el caso de un "Segment Tree" de mínimos, o también se puede encontrar el mínimo prefijo que sume al menos x para el caso de que se esté trabajando sobre la suma. Para poder aplicar esta técnica, uno comienza parado en el nodo raíz, y observando los valores del nodo padre del subárbol izquierdo y derecho, toma una decisión. De esta forma, se estaría aplicando una forma de recorrer el árbol similar a como se hace en un Árbol Binario de Búsqueda, pero sin mantener la invariante que tienen estos tipos de árboles. Veamos un ejemplo para entender más fácilmente esto.

3.1.1. CSES 1143 - Hotel Queries

Link al problema: https://cses.fi/problemset/task/1143

> Hay n hoteles en una calle. Para cada hotel conoces el número de habitaciones libres. Tu tarea es asignar habitaciones de hotel para grupos de turistas. Todos los miembros de un grupo quieren alojarse en el mismo hotel.
> Los grupos llegarán a usted uno tras otro y usted sabrá para cada grupo la cantidad de habitaciones que necesita. Siempre asignas un grupo al primer hotel que tenga suficientes habitaciones. Después de esto, el número de habitaciones libres en el hotel disminuye.
> Para cada grupo de turistas, imprima el número del hotel al que fueron asignados.

Para resolver el problema, vamos a usar el "Segment Tree" de máximos como una estructura para hacer Binary Search. La idea es encontrar el primer hotel que tenga suficientes habitaciones para el grupo de turistas en $O(\log n)$. Como base, utilizaremos un "Segment Tree" de máximos, que se realizará sobre el arreglo base de habitaciones disponibles de cada uno de los hoteles. Los hoteles estarán ordenados según cuál estará primero. Esto nos será de utilidad para hallar el primer hotel con habitaciones suficientes haciendo búsqueda binaria.

Para cada grupo de turistas, partimos desde el primer nodo (la raíz). Nos moveremos hacia el subárbol izquierdo si su padre contiene un número mayor o igual a la cantidad de habitaciones que necesitamos, o al de la derecha si el nodo contiene un número menor. De esta forma, cuando arribemos a la última capa del Árbol Binario Completo, nos encontraremos en el primer hotel que tenga la suficiente cantidad de habitaciones. Esto se debe a que siempre que había algún hotel con al menos esta cantidad de habitaciones disponibles, nos fuimos en la dirección con índice menor del arreglo. De esta forma, la respuesta va a ser $k - n$, siendo k el nodo de la última capa al cual llegamos. También, cabe aclarar que si no hay habitaciones suficientes en ningún hotel para este grupo (que alcanza con verificar que la raíz tenga un número más chico que la cantidad de habitaciones requerida por el grupo), la función devolverá 0.

El código para resolver este problema se encuentra en el Anexo. En este, se planteó una función *occupy* que busca el hotel a ocupar, ocupa las habitaciones y retorna el hotel que se ocupó, según lo solicitado por el enunciado. Como cada llamada a esta función tiene complejidad $O(\log n)$, la complejidad total del ejercicio es $O(n \log n)$. De esta forma, trabajamos cómo se puede usar Binary Search sobre un "Segment Tree".

3.2. Compresión de Coordenadas

Esta es una técnica muy utilizada en Programación Competitiva cuando se cumplen las siguientes hipótesis:

- Uno debe utilizar una estructura de datos basada en arreglos.
- Los índices pueden llegar a valores muy grandes.
- Se saben de antemano con cuales índices uno desea trabajar, es decir, cuáles índices son relevantes para la resolución del problema.

La idea de esto es tomar el conjunto de índices $C_i = \{I_1, I_2, ..., I_n\}$ que se utilizarán en el caso de prueba (que generalmente se pueden obtener analizando el input), y luego se define una función biyectiva $f(i)$ que mapea cada uno de los índices a un número entre $[0, n-1]$ siendo $n = \#C_i$ (cantidad de elementos del conjunto) manteniendo que, para dos índices $i, j \in C_i$, con $i \neq j$, si $i < j \implies f(i) < f(j)$. De esta forma, según la cantidad de distintos valores que reciba el input, indicará cuántos elementos tendrá el mapeo de índices. De esta forma, se podrán volver a utilizar distintas estructuras sobre arreglos que con tamaños muy grandes se dificultaba, como es el "Segment Tree".

Desde el punto de vista implementativo, para poder mapear desde un índice I_n a $f(I_n)$ se puede utilizar un *map* o un *unordered_map*. Para mapear la vuelta, es decir desde $f(I_n)$ a I_n (lo que se puede hacer porque la funcion es biyectiva), se puede usar un *array*. Ahora, veamos un problema para ver cómo se aplica esto.

3.2.1. CSES 1144 - Salary Queries

Link al Problema: https://cses.fi/problemset/task/1144

> Una compañía tiene n empleados con ciertos salarios. Tu tarea es procesar salarios, pudiendo preguntar cuales eran los salarios de los empleados en un rango, y actualizar el salario de un empleado. Los salarios pueden ir entre 0 y 10^9.

El gran problema que tiene este ejercicio es que los salarios pueden ser hasta 10^9, por lo que si queremos guardar un Segment Tree en el que cada posición guarde cuántas personas tienen ese salario (para hacer una query de rangos) requeriríamos muchísima memoria. Es por eso que debemos recurrir a compresión de coordenadas. Lo que debemos hacer es leer todo el input de entrada, y guardar todos los posibles valores de los salarios actuales y futuros en un arreglo, al cual llamaremos *compresion* Luego, se puede mapear, una vez ordenado el arreglo y eliminado los repetidos, cada valor a un índice entre $[0, c-1]$ siendo c la cantidad de salarios distintos. Luego, podemos armar un *map* que mapee cada salario posible a su posición en ese arreglo creado anteriormente.

Para poder procesar las queries, utilizaremos un "Segment Tree" de longitud igual (o su potencia de 2 más grande y cercana) a la cantidad de elementos de la lista de compresión de coordenadas (es decir, con la cantidad de salarios distintos). Cada posición del arreglo indica cuántas personas tienen el salario $compresion[i]$, siendo i el índice en el arreglo sobre el cual se realizan las consultas. Entonces, en el caso que se nos pida actualizar un salario, simplemente debemos restarle 1 a la posición con el salario anterior, y sumarle 1 al nuevo salario. Por ende, es de utilidad ir guardando en otro arreglo el valor del salario actual para un empleado e ir actualizándolo a medida que se van realizando los cambios.

Para realizar las consultas, entonces, se puede hacer una búsqueda binaria sobre el arreglo *compresion* para encontrar los índices en el arreglo de consultas sobre los cuales llamar al "Segment Tree", o, si se agregó a *compresion* los valores donde se hacen las consultas, se puede utilizar el *map* de salarios a índice en el arreglo de consultas para saber que valores pasarle al método *query* del "Segment Tree". De esta forma, pudimos resolver el problema con complejidad $O(n \log n)$, ya que cada consulta o actualización va a tener complejidad $O(\log n)$.

En el Anexo se encuentra disponible el código que resuelve este problema. Este es un ejemplo de como se puede usar compresión de coordenadas para utilizar estructuras de datos sobre arreglos.

3.3. Segment Trees Dinámicos

Existen algunos problemas donde uno debe utilizar esta estructura de datos sobre rangos con números muy grandes, pero uno no posee de entrada todos los valores sobre los cuales se deben realizar las consultas y actualizaciones, como es en el caso anterior. Este tipo de problemas es conocido como problemas "online", y suelen requerir responder una consulta o actualización antes de recibir la siguiente. Si nosotros alocáramos toda la memoria de entrada y representáramos el "Segment Tree" como hicimos anteriormente, nos quedaríamos sin memoria. Pero, para subsanar esto, podemos recurrir a la memoria dinámica y almacenar solo los nodos que tengan valores interesantes para nosotros.

De esta forma, la implementación del "Segment Tree" es muy similar a la recursiva que trabajamos anteriormente, a diferencia que cada nodo del árbol no va a ser representado más como una posición en un arreglo, sino más bien va a ser un *struct* que contenga un puntero tanto hacia su hijo izquierdo como su hijo derecho. Será *nullptr* si uno no tendrá más hijos, o si una rama no tiene ningún valor seteado. El *struct* se vería de la siguiente forma:

```
using tint = long long;
struct Node {
  tint value;
  Node *left, *right;
  Node(tint value): value(value) {}
  Node(): value(0) {}
};
```

De esta forma, en la operación *query*, antes de recurrir a cada uno de los hijos, debemos chequear si estos nodos existen. Si no existen, es como si estos tuvieran el valor identidad para el Monoide con el que se está trabajando.

Para la operación *update*, en cada paso se debe chequear si el nodo que contiene el valor a actualizar existe. En caso de que que no exista, es decir, que hacia la dirección que se quiere realizar la actualización sea *nullptr*, debe crearse el nodo y luego recurrir a ese subárbol. El proceso finalizará cuando se llegue a la última capa del árbol, donde el nodo represente al elemento del arreglo sobre el cual se hacían originalmente las consultas.

Cabe aclarar que es necesario al inicializar la estructura indicar el máximo n que esta puede llegar a soportar, ya que esto indica la profundidad que va a tener el árbol al momento de realizar las consultas. Por ejemplo, si $n = 10^9$, el árbol tendrá una profundidad de $\log_2(10^9) \approx 30$. Por ultimo, también hay que recordar que en C++ se debe liberar la memoria una vez destruida la estructura de datos. Aunque en programación competitiva no es totalmente necesario realizarlo (ya que el juez corre cada caso de prueba por separado o nos da mucha memoria disponible), es una buena práctica en el mundo del del desarrollo de software.

3.3.1. CSES 1144 - Salary Queries - Segment Tree Dinámico

Link al problema: https://cses.fi/problemset/task/1144

Este problema, trabajado en la sección anterior, puede también ser resuelto utilizando un "Segment Tree Dinámico". Simplemente se crea esta estructura con el máximo $n = 10^9$ y se puede llamar a las funciones *update* y *query* con los valores indicados por el enunciado. Al igual que en el desarrollo anterior, cada posición del arreglo original (que en este caso está implícito al usar una estructura dinámica) contiene la cantidad de personas que tienen ese salario. Luego, la estructura es modelada con la operación suma, y en $O(\log n)$ podemos saber cuántas personas tienen un salario entre a y b, dados por cada una de las consultas. Para la actualización de los valores, se le resta 1 a la posición que representa al salario que el empleado tenía anteriormente, y se le suma 1 al nuevo salario. De esta forma, quedaría modelado el problema. En el Anexo se encuentra el código para esta solución.

4. Lazy Propagation Segment Tree

Ahora, imaginemos que no solo nos interesa realizar actualizaciones de un único valor, sino que nos interesa hacer actualizaciones de un rango $[a, b]$. Si uno desea hacer este tipo de actualizaciones y solo interesa hacer una query del valor de una celda únicamente, se puede hacer usando un "Segment Tree" clásico almacenando la información como una arreglo de diferencias. Pero, esto nos limita también a no hacer queries en rangos. Es por eso que el "Segment Tree" con propagación perezosa nos va a permitir hacer actualizaciones en rango en $O(\log n)$ y también consultas en rango en $O(\log n)$. ¡La misma complejidad que un Segment Tree tradicional! De todas formas, tiene contras, ya que utiliza más memoria que el "Segment Tree" clásico y además no se puede usar para cualquier Monoide, sino que solo para operaciones que se puedan componer. A continuación, analizaremos el funcionamiento de esta estructura.

4.1. Operaciones de la Estructura

El "Lazy Segment Tree" soportará las siguientes dos operaciones principales, ambas con complejidad en $O(\log n)$:

- *query(l: int, r: int): T*, que dado un rango, nos permitirá obtener el resultado de aplicar una cierta operación en el rango $[l, r]$
- *update(l: int, r: int, val: T)* que actualizará los valores en el rango $[l, r]$ con el valor indicado en *val*. Se pueden definir distintas operaciones de actualización. Por ejemplo, en un "Lazy Segment Tree" de suma se puede crear un *update* para setear un segmento a un cierto valor *val*, u otra operación para incrementar ese segmento por *val*. Lo que se debe cumplir es que las operaciones se puedan componer.

Cabe aclarar que estas estructuras de datos no funcionan con todos los Monoides. Solo funcionan con operaciones que además de ser Monoides, se deben poder componer para poder ser aplicadas todas juntas. Por ejemplo, si se aplica la operación suma a un nodo con valores 2, 4 y 6 es equivalente a aplicar una sola vez la operación sumar 12. Por ende, no habría problema de utilizar esta estructura de datos en este caso.

4.2. Representación de la Estructura

La estructura se puede representar utilizando un arreglo de *nodos*, con las mismas consideraciones que un "Segment Tree" clásico. Lo que hay que tener en cuenta es que cada *Nodo* tendrá no solo el valor actual de este nodo, sino que también contendrá más valores que nos permitirán hacer una propagación perezosa. Esto se entenderá más al analizar la implementación de las operaciones.

4.3. Las Operaciones

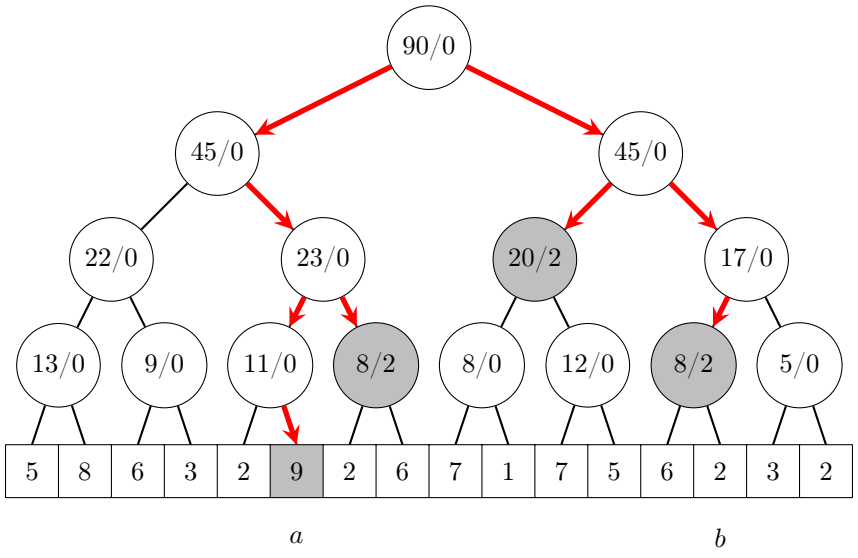
Para poder implementar las operaciones de un "Lazy Segment Tree", es más cómodo aplicar un enfoque recursivo. Para facilitar la explicación, trabajaré cada uno de los algoritmos dando como ejemplo al "Lazy Segment Tree" de suma, que nos permitirá hacer una consulta en rango de la suma en $[l, r]$ y una actualización en $[l, r]$ que consiste en incrementar o decrementar cada nodo de ese rango por un valor val. En este caso, cada *Nodo* almacenará dos valores: la suma de los elementos del subárbol (llamado a partir de ahora sum) y el valor por el cuál va a ser incrementado o decrementado todo el subárbol (el cual será llamado $lazy$). Veamos cada una de las operaciones, en este caso comenzando por *update* y luego yendo por *query*.

La operación de *update* irá recorriendo de forma recursiva cada subárbol según el rango que desea ser actualizado. Para ello, nos definiremos una función auxiliar muy similar a la del "Segment Tree" clásico, con los siguientes parámetros: *update(a: int, b: int, val: int, k: int, l: int, r: int)*, con $[a, b]$ el rango a actualizar, val el valor a aplicar, k el número de nodo actual que representa el rango $[l, r]$ del arreglo original. La función aplicará el siguiente algoritmo:

- Si el rango $[l, r]$ está afuera del rango del nodo $[a, b]$, es decir $[a, b] \cap [x, y] = \emptyset$, retornar, ya que no se debe hacer nada.
- Si el rango $[a, b]$ está totalmente contenido dentro de $[l, r]$, es decir $[a, b] \cap [l, r] = [a, b]$, setear el valor $lazy$ y retornar. En el caso de la suma, sería incrementar el valor $lazy$ por val. Eso significa que en $lazy$ quedará una operación pendiente a aplicar en los nodos inferiores, que se hará cuando sea necesario, ya que teniendo la suma de los subárboles en $suma$ y este valor $lazy$ se puede deducir cuál va a ser el valor total del nodo, mediante la *evaluación* del mismo.
- Si el rango $[l, r]$ está parcialmente contenido en $[a, b]$, que es cuando $[a, b] \cap [x, y] \neq \emptyset$, se deberá propagar los valores $lazy$ a los subárboles izquierdo y derecho, y borrar del nodo ese valor (setearlo a 0 en el caso de la suma). Con propagar se refiere a aplicar una función que le indique a los subárboles qué conjunto de operaciones deben aplicar. En el caso de la suma con operación de incrementar en val, este paso consiste en simplemente en sumar el valor $lazy$ a cada uno de los valores $lazy$ de sus subárboles. Luego, se llama a actualizar a cada uno de estos de forma recursiva, similar a como se hacía en el *query* del "Segment Tree" clásico.
- Por último, si se cayó en el tercer caso de los anteriores, se actualiza el valor de este nodo, evaluando cada uno de los nodos hijos. Para el caso de la suma, el resultado de evaluar los hijos se guardará en sum.

Este algoritmo, al tener un esquema muy similar al *update* de la estructura de datos original, tiene una complejidad de $O(\log n)$. Además, cabe aclarar que la función de evaluación para un nodo para el caso de la suma será $eval(k) = suma + lazy \times (r - l + 1)$, con $(r - l + 1)$ la cantidad de hijos que tiene este nodo. Esta operación hace sentido ya que $lazy \times (r - l + 1)$ es equivalente a el resultado que tendría el nodo de aplicar esa operación perezosa a cada uno de los elementos en el rango $[l, r]$. Y además, se le agrega $suma$, que indica los valores de suma para cada uno de los subárboles izquierdo y derecho.

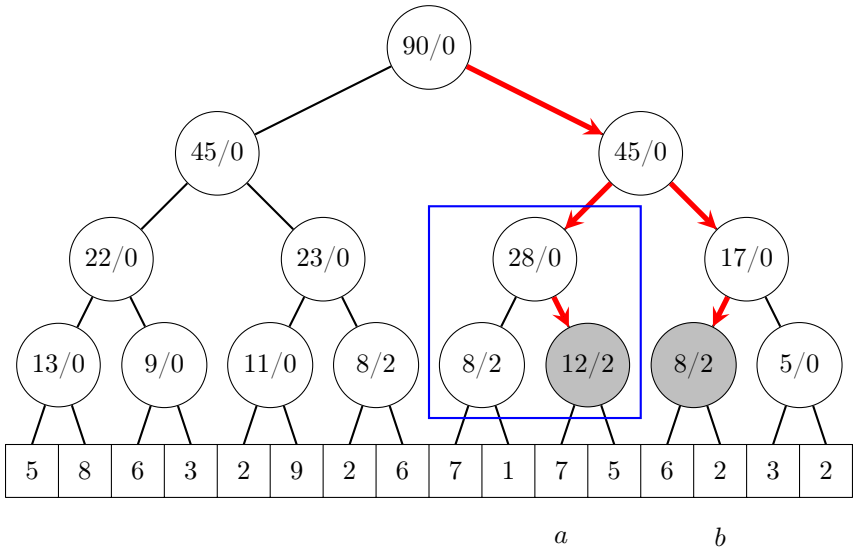
Por ejemplo, dado un "Lazy Segment Tree" genérico, sumarle 2 al rango $[a, b]$ del gráfico resultaría en el siguiente "Lazy Segment Tree":



Ahora, analicemos la función *query* para el "Lazy Segment Tree". Esta operación, nuevamente, tendrá una función auxuliar, que es similar a la de "Segment Tree" tradicional: *query(a: int, b: int, k: int, x: int, y: int)* con $[a, b]$ el rango sobre el cual se quiere hacer la *query*, k el número de nodo actual y el rango $[l, r]$, el rango que representa el nodo k en el arreglo original. El algoritmo sería de la siguiente forma:

- Si el rango $[l, r]$ está afuera del rango del nodo $[a, b]$, es decir $[a, b] \cap [x, y] = \emptyset$, retornar el valor identidad, como en el caso de la suma el 0 ya que no hay valores que estén en el rango solicitado para ese nodo.
- Si el rango $[a, b]$ está totalmente contenido en el nodo, es decir $[a, b] \cap [l, r] = [a, b]$, retornar el valor resultante de *evaluar* el nodo con la función definida para el problema. En el caso de la suma, con la función descripta anteriormente. Esto devolverá el valor que tiene el rango representado por el nodo al aplicar la operación.
- Si no ocurre ninguno de los dos casos anteriores, es decir que el nodo está parcialmente contenido en el intervalo a buscar, se debe realizar un paso de propagación idéntico al momento de actualizar. Esto se debe a que nosotros vamos a hacer consultas en los dos subárboles, y para ello necesitamos que todas las actualizaciones ya hayan sido hechas para cada uno de los subárboles al momento de consultarlos. Esto se logra a través del paso de propagación, que informará a los nodos hijos los valores que estos van a tener. Al igual que en el caso anterior, al momento de propagar las actualizaciones, se deben borrar de este nodo, ya que hasta este punto las operaciones fueron aplicadas.
- Luego, si se cayó en el caso anterior, se llama recursivamente a *query* en cada uno de los dos hijos. Esto nos va a dar la información que se debe devolver para esta consulta. De todas formas, resta actualizar el valor actual del nodo, ya que se propagó la actualización pero no se aplicó el resultado de realizarla en sus hijos. Por ende, se evalúan los nodos hijos y se aplica la operación al evaluarlos. Luego, finalmente, se retornan los valores.

Por ejemplo, dado un "Lazy Segment Tree", consultar al rango $[a, b]$ produciría una propagación en el árbol de la siguiente forma (además de retornar la suma):



Puede resultar un poco confuso a veces pensar el algoritmo de Lazy Propagation. Especialmente, es confuso cuándo se debe realizar la propagación, ya que uno intuitivamente solo piensa que debe hacerse en el *update*. Pero, notar que, cuando uno quiere acceder a un nodo, todas las actualizaciones de ese mismo ya debieron ser hechas. Por ende, es importante siempre realizar todas las propagaciones hacia ese nodo antes de accederlo, por lo que, como norma o regla memotécnica, hay que recordar hacer la propagación antes de recurrir a cualquier nodo inferior. También, al momento de propagar, hay que recordar actualizarse a uno mismo con el resultado de evaluar lo propagado en los nodos hijos.

4.4. ¿Cómo pensar un problema con Lazy Segment Tree?

Una vez ya analizado el algoritmo, este puede ser modificado para resolver distintos problemas de Programación Competitiva. Para facilitar esto, hay cuatro cuestiones clave que deben ser definidas al momento de resolver un problema con esta estructura de datos:

- **Operación matemática a elegir**: es la operación básica sobre la cual la estructura de datos trabaja. El resultado de aplicar una consulta en el rango $[l, r]$ será la consecuencia de aplicar esta operación en ese rango.
- **Valores a almacenar de cada nodo:** estos son los valores que nos permitirá guardar tanto el resultado de aplicar la operación trabajada a los nodos hijos como los valores "lazy" que contendrán la información de las operaciones aplicadas a ese nodo hasta ese momento pero que aún no se pasaron a sus hijos.
- **Función de evaluación de un nodo:** para armarla, uno debe preguntarse, ¿qué cálculo debo hacer para saber el valor actual del nodo sin que los hijos necesariamente hayan sufrido la actualización? Suele consistir en una cuenta matemática que combine los valores de los nodos hijos con los valores "lazy".
- **Paso de propagación**: establece cómo se deben trasladar los valores "lazy" de un nodo hacia sus hijos. Varía según el tipo de actualización con la que se está trabajando. Esto también está relacionado con cómo se debe recurrir a los dos nodos hijos al momento de hacer un *update*.

Ahora, pasaremos a resolver distintos problemas con esta estructura de datos. Para explicar las soluciones, explicaremos cada una de estas cuestiones para el problema que se esté trabajando.

4.5. Algunos Problemas

4.5.1. CSES 1735 - Range Updates and Sums

Link al problema: https://cses.fi/problemset/task/1735

> Tu tarea es mantener un vector de n valores y procesar eficientemente los siguientes tipos de consultas:
>
> 1. Aumente cada valor en el rango $[a, b]$ en x.
> 2. Establezca cada valor en el rango $[a, b]$ en x.
> 3. Calcule la suma de los valores en el rango $[a, b]$.

Este es un problema clásico de suma con "Lazy Segment Tree", pero con un pequeño *twist*: se puede tanto setear el valor de un rango como aumentarlo. Por ende, las operaciones se complejizarán un poco. Analicemos como debemos definir cada una de las cuatro variables mencionadas anteriormente para este "Lazy Segment Tree".

Comencemos estableciendo los valores a almacenar en cada nodo. Cada nodo tendrá tres valores: *suma*, que indica la suma de sus hijos; *sumaLazy*, que indica por cuánto se debe incrementar a cada uno de los elementos del arreglo original que representa el nodo y $setValueLazy$, que muestra a cuánto se debe saetear el valor de cada uno de los elementos del arreglo. Notemos que la operación de establecer cada uno de los valores del rango "reemplaza" a todas las operaciones de incremento que se hicieron anteriormente, ya que serán superpuestas (por ende, cuando se realice una de estas, se debe poner en 0 el valor de *sumaLazy* de los nodos que sean afectados). Pero, hacer un incremento o decremento va a mantener el valor de $setValueLazy$ como base, pero cambiaría *sumaLazy*, acumulando las operaciones de suma que se vayan realizando hasta la llamada de otro *seteo* de valores.

Para la función de evaluación, esta dependerá de los valores de *sumaLazy* y $setValueLazy$. Se pueden producir los siguientes escenarios:

- Si no hay updates por hacer, la función solo deberá retornar *suma*.
- En caso de que haya una actualización de *seteo* por ser realizada, es decir, cuando $setValueLazy > 0$ (ya que no se pueden asignar números menores o iguales a 0), la función evaluar va a devolver $setValueLazy \times (b - a + 1)$: el valor que se seteo multiplicado por la cantidad de elementos en el rango.
- Si hay una actualización de suma por realizarse, evaluar el nodo devolverá $sumaLazy \times (b - a + 1) + suma$: lo que se le va a sumar a cada uno de sus hijos por la cantidad de elementos en el rango, más la suma de los hijos.
- Si se debe hacer un *seteo* y luego sumarle valores, el resultado de evaluar el nodo será $(setValueLazy + sumaLazy) \times (b - a + 1)$: el valor que se *seteo* más el incremento de *sumaLazy* multiplicado por la cantidad de elementos en el rango.

Estos son los escenarios posibles de evaluación de la función. Luego, la operación a elegir, ya que lo indica el enunciado, será la suma. Y el paso de propagación no es muy difícil: simplemente se deben copiar los valores del nodo padre a

ambos de sus hijos, ya que evaluar ambos nodos hijos y sumar sus resultados, como lo único que hace es partir el rango en dos, daría el mismo número que evaluar al nodo padre de ellos. De esta forma, tenemos definida la estructura de datos para resolver el problema. ¡Solo queda implementarla! La implementación estará en el Anexo.

4.5.2. CSES 1736 - Polinomial Queries

Link al problema: https://cses.fi/problemset/task/1736

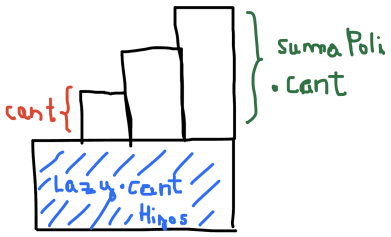
> Su tarea es mantener un vector de n valores y procesar eficientemente los siguientes tipos de consultas:
>
> 1. Aumente el primer valor en el rango $[a, b]$ en 1, el segundo valor en 2, el tercer valor en 3, y así sucesivamente.
> 2. Calcule la suma de los valores en el rango $[a, b]$.

Personalmente, este es uno de los problemas clásicos que más me ha costado resolver. Pero, veremos que si aplicamos un enfoque muy similar al anterior para resolverlo, su solución no es tan complicada. Como observación importante, notemos que la operación incrementar siempre suma una serie aritmética de razón 1 (la diferencia entre los términos es 1). Supongamos que aplicamos en un mismo intervalo $[a, b]$ la operación de actualización. La primera vez, la razón entre todos los valores del intervalo será 1, y la segunda será 2. Podemos entonces intuir que para ese intervalo se puede representar la suma sabiendo cuántas veces se aplicó la operación en ese rango. Pero, nos falta un poco de información: podemos almacenar la diferencia, pero no sabemos cuál es el valor del primer elemento del rango (ya que ese, con la cantidad de veces que se aplicó la operación, nos define cada uno de los valores). Entonces, podemos guardar ese valor también para cada nodo.

Consecuentemente, con estas ideas, podemos guardar tres variables para cada nodo del árbol:

- *suma*: la suma de los valores en el rango.
- *lazy*: valor que contiene cuanto se le suma al primer elemento del rango que denota el nodo.
- *cantidad*: cuántas veces se le aplicó una operación de actualización a ese nodo, antes de ser propagada.

Luego, para evaluar cada nodo, podemos plantear la siguiente fórmula: $(sumaPoli \times cantidad) + (lazy \times cantHijos) + suma$ donde $cantHijos = y - x + 1$, con $[x, y]$ el rango del nodo y $sumaPoli = \frac{cantHijos \times (cantHijos-1)}{2}$. Para entender la fórmula, pensemos cada uno de los términos por separado. El primer término nos aplica el resultado de aplicar *cantidad* veces la operación para el rango que representa el nodo. Eso se puede representar usando la clásica Suma de Gauss, y multiplicando eso por la razón de cambio, es decir, la cantidad de veces que se aplicó la operación. El segundo término aplica un "piso", ya que nos pone como incremento para cada valor del arreglo original a la cantidad que tiene el primer elemento del rango representado por el nodo. El tercer término simplemente agrega el resultado de evaluar los dos nodos inferiores, y estaremos usando como operación la suma, ya que es lo que indica el enunciado. Se puede pensar esto de forma gráfica de la siguiente forma (ignorando el tercer término):



Ahora que sabemos los valores del nodo, debemos pensar el paso de propagación, el cuál no es tan sencillo ya que no es igual para el nodo izquierdo y para el nodo derecho. Para el nodo izquierdo, simplemente le sumamos tanto los valores *lazy* como *cantidad* del padre. Esto se debe a que el comienzo de la secuencia para la el hijo izquierdo es igual al comienzo de la secuencia para el nodo padre. Por ende, este nodo hijo va a abarcar un rango menor al padre, pero va a comenzar igual.

Para el nodo derecho, se le va a sumar *cantidad* ya que ese es el número de operaciones (o la razón de la serie aritmética) que se le tiene que aplicar. Pero, *lazy*, que contiene el primer elemento de la secuencia, no se incrementa en *lazy* del padre. En cambio, se lo incrementa por $temp = lazy + cantidad * mitadRangoActual$, siendo $mitadRangoActual = (r - l + 1)/2$ y los tres valores haciendo referencia al nodo padre. Esto simplemente nos calcula

el término de la sucesión en la posición *mitadRangoActual*, lo que hace que sea el primer término del subárbol derecho. De esta forma, podemos incrementar por *temp* al valor *lazy* del hijo derecho.

Por último, debemos notar que, al momento de actualizar un rango, cuando se recurre a los dos subárboles izquierdo y derecho luego de propagar, se hace hacia el nodo derecho con exactamente la misma división que en el paso de propagación, pero con la diferencia que toma como primera posición a la cantidad de nodos del subárbol derecho (que se puede calcular con $[l, r]$ y $[a, b]$ en la función recursiva). No entraré mucho en detalle ya que es similar al paso de propagación, y se puede ver en el código del Anexo.

De esta forma, quedó diseñada la estructura de datos para resolver el problema, por lo que solo resta leer el input y utilizarla. El problema resuelto quedará en el Anexo.

5. Segment Tree Persistente

Ahora, una pregunta motivadora que podemos hacernos es, ¿podemos hacer que las estructuras de datos viajen en el tiempo? Es decir, ¿podemos ir actualizando la estructura de datos y poder volver a visitar copias pasadas de la misma de forma eficiente, sin generar una copia cada vez?

Una de las estructuras persistentes más simples son los *arreglos persistentes*, que permiten en $O(1)$ agregar un elemento al arreglo y en $O(\log n)$ acceder al mismo en un tiempo t determinado. Esto lo hace simplemente haciendo un arreglo de arreglo de pares, donde en cada arreglo se almacenan las tuplas $< tiempo, valor >$ para una posición determinada. Nosotros nos centraremos en "Segment Trees", que permitirán hacer consultas a lo largo de las distintas modificaciones. Muchas veces esto es explicado hablando de modificaciones a lo largo del tiempo, pero le daré un enfoque distinto: trabajaremos sobre hacer copias de la estructura de datos en tiempo y espacio $O(1)$, de tal forma que uno luego podrá modelar el "tiempo" de forma que sea necesaria, ya que podrá guardar las copias que uno quiera ya para los momentos que uno desee.

5.1. Operaciones de la Estructura

Esta estructura de datos soportará las siguientes operaciones:

- *query(l: int, r: int) ->int*: hacer una consulta de una operación en el "Segment Tree". Complejidad $O(\log n)$ tiempo, no modifica el espacio.
- *update(pos: int, val: int)*: actualizar la posición *pos* con el valor *val*. Complejidad $O(\log n)$ tiempo y espacio para cada llamada a la operación. Es decir, cada llamada a *update* asignará $\log n$. Nuevas posiciones en memoria.
- *copy()->PersistantSegmentTree*: genera una copia de la estructura de datos en $O(1)$. Modificar esta nueva estructura no modificará la anterior.

Para esta estructura de datos se puede utilizar cualquier Monoide.

5.2. Representación de la Estructura

Para representar la estructura, utilizaremos como base los "Segment Trees Dinámicos", es decir, los "Segment Trees" que viven en memoria dinámica. Cada estructura tendrá un nodo raíz *root* que permitirá recorrerla, y cada nodo tendrá el valor del mismo, el puntero al hijo izquierdo y el puntero al hijo derecho. Veremos como trabajar con la estructura el la siguiente sección.

5.3. Las Operaciones

La operación *query(l: int, r: int)* es idéntica a la del "Segment Tree Dinámico", no requiere ninguna modificación. Se utilizará la misma función auxiliar que el *query* recursivo para la estructura clásica.

La operación *update(pos: int, val: int)* también será idéntica a la operación del "Segment Tree Dinámico", pero no se realizará la actualización sobre los nodos ya creados para esa estructura. En cambio, para cada uno de los $\log n$ nodos que se modificarán al momento de realizar una actualización, se crearán nuevos nodos. Para las ramas ya existentes y no modificadas, de todas formas, se apuntará a los nodos originales ya creados anteriormente. De esta forma, en cada actualización, se creará siempre una nueva raíz (que siempre es modificada) que permitirá recorrer la estructura luego de realizar los cambios. Esta será la nueva raíz del "Segment Tree Persistente". Como los nodos previos a la actualización se mantuvieron intactos, recorrerlos desde la raíz anterior nos permitiría recorrer la estructura previa.

¡Y así con todas las versiones anteriores si guardamos sus raíces! Y lo mejor de todo, sumamos por la operación solo $O(\log n)$ de memoria. Esa es la magia de la operación.

Por último, definimos la operación *copy()->PersistantSegmentTree*, que lo único que hace es copiar la referencia al nodo raíz y devolver una nueva instancia de *PersistantSegmentTree*, con el fin de mantener la modularidad. Cabe aclarar que en buenas prácticas, habría que limpiar los nodos que pierden todas las referencias hacia él, ya que C++ no lo maneja de forma automática. De todas formas, esto no se suele hacer en programación competitiva, ya que nos dan mucha memoria y nunca es necesario liberarla.

Estas son las operaciones, que con breves modificaciones sobre el "Segment Tree Dinámico" nos agregan la capacidad de, por un poco de memoria más, guardar el historial a medida que se va modificando la estructura de datos. El manejo de esta estructura de verá de forma mucho más clara a partir de su código, que veremos a partir de un problema clásico, a continuación.

5.4. Problemas

5.4.1. CSES 1737 - Range Queries and Copies

> Tu tarea es mantener una lista de arreglos que inicialmente tiene un solo arreglo. Debes procesar los siguientes tipos de consultas:
>
> 1. Establecer el valor a en el arreglo k como x.
> 2. Calcular la suma de los valores en el rango $[a, b]$ en el arreglo k.
> 3. Crear una copia del arreglo k y agregarlo al final de la lista.

Para resolver el problema, podemos utilizar la estructura antes mencionada para el Monoide suma. Se puede mantener un arreglo *arr* de punteros a *PersistantSegmentTree*, donde $arr[k]$ representa al arreglo k dado por el enunciado. En la posición 1 se encuentra la estructura correspondiente al arreglo inicial. Luego, para hacer actualizaciones o consultas del tipo 1 o 2, se puede hacer la consulta en la estructura $arr[k]$. Por último, para hacer una copia, se puede llamar a *copy()*, y el puntero de la copia agregarlo al final de *arr*. Como las primeras dos operaciones se hacen en complejidad $O(\log n)$ y la tercera en $O(1)$, la complejidad en peor caso será de $O(q \log n)$, suficiente para resolver el problema. El código se encuentra en el Anexo.

6. Conclusiones

A lo largo de esta monografía pudimos desarrollar la estructura de datos "Segment Tree", que nos permite realizar distintos tipos tipos de consultas sobre un arreglo de información (que puede ser implícito o explícito), con actualizaciones sobre los datos que se almacenan. Además, analizamos distintas técnicas y variantes de esta estructura que nos permitirán obtener ciertas flexibilidades o ventajas para poder resolver problemas más complejos.

6.1. Problemas Adicionales

Estos son algunos problemas que pueden servir para trabajar los conceptos que estuvimos trabajando:

- Range Update Queries: https://cses.fi/problemset/task/1651
- List Removals: https://cses.fi/problemset/task/1749
- Distinct Value Queries: https://cses.fi/problemset/task/1734
- Prefix Sum Queries: https://cses.fi/problemset/task/2166
- Sumo: https://juez.oia.unsam.edu.ar/task/123
- Clima: https://juez.oia.unsam.edu.ar/task/35
- Mega Inversions: https://open.kattis.com/problems/megainversions
- Contando puentes: https://juez.oia.unsam.edu.ar/task/12

6.2. Bibliografía

Esta es la bilbliografía consultada para la realización de la monografía:

- Competitive Programmer Handbook: https://cses.fi/book/book.pdf
- Repositorio del Competitive Programmer Handbook: https://github.com/pllk/cphb
- OIA Wiki: http://www.oia.unsam.edu.ar/wp-content/uploads/2017/11/segment-tree.pdf
- Wikipedia: https://es.wikipedia.org/wiki/Monoide
- USACO Guide: https://usaco.guide
- CP Algorithms: https://cp-algorithms.com

Agredecimientos a Antti Laaksonen por los gráficos utilizados en el Competitive Programmer Handbook.

6.3. Temas Adicionales

Estos son algunos temas que se pueden consultar para ampliar el conocimiento del tema:

- Fenwick Tree
- Segment Tree y Fenwick Tree 2D
- Segment Tree Lazy Creation + Lazy Update (codigo de USACO Guide)

7. Anexo

7.1. CSES 1648 - Dynamic Range Sum Queries - Recursivo

Link al Problema: https://cses.fi/problemset/task/1648

```cpp
// AC - Tiempo 0.18 s

#include <iostream>
#include <vector>

#define forn(i, n) for(tint i = 0; i < tint(n); i++)

using namespace std;
using tint = long long;

struct SegmentTree {
  tint n, segN;
  tint segTree[1000000];

  SegmentTree (tint _n) {
    n = _n; segN = 1;
    while (segN < n) segN *= 2;
  }

  tint retrieve(tint a, tint b, tint k, tint x, tint y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return segTree[k];
    tint d = (x+y)/2;
    return retrieve(a,b,2*k,x,d) + retrieve(a,b,2*k+1,d+1,y);
  }

  tint retrieve(tint a, tint b) {
    return retrieve(a, b, 1, 0, n-1);
  }

  void update(tint pos, tint k, tint x, tint y, tint val) {
    if (x == y) { segTree[k] = val; return; }
    tint d = (x + y) / 2;
    if (pos <= d) update(pos, k*2, x, d, val);
    else update(pos, k*2+1, d+1, y, val);
    segTree[k] = segTree[k*2] + segTree[k*2+1];
  }

  void update(tint pos, tint val) {
    update(pos, 1, 0, n-1, val);
  }
};

int main () {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);

  tint n, q;
        cin >> n >> q;

  SegmentTree segTree (n);

  forn (i, n) {
    tint temp; cin >> temp;
```

```
    segTree.update(i, temp);
  }

  forn (i, q) {
    tint func, a, b;
    cin >> func >> a >> b;

    if (func == 1) segTree.update(a-1, b);
    else cout << segTree.retrieve(a-1, b-1) << "\n";
  }
}
```

7.2. CSES 1648 - Dynamic Range Sum Queries - Iterativo

Link al Problema: https://cses.fi/problemset/task/1648

```
// AC - Tiempo 0.13 s

#include <iostream>
#include <vector>

#define forn(i, n) for(tint i = 0; i < tint(n); i++)

using namespace std;
using tint = long long;

struct SegmentTree {
  tint segN;
  tint segTree[1000000]; // Hay que hacerlo de esta forma ya que sino con Vector crashea

  SegmentTree (tint n) {
    segN = 1;
    while (segN < n) segN *= 2;
  }

  tint retrieve(tint a, tint b) {
    a += segN; b += segN;
    tint s = 0;
    while (a <= b) {
      if (a%2 == 1) s += segTree[a++];
      if (b%2 == 0) s += segTree[b--];
      a /= 2; b /= 2;
    }
    return s;
  }

  void update(tint k, tint x) {
    k += segN, segTree[k] = x;
    for (k /= 2; k >= 1; k /= 2) {
        segTree[k] = segTree[2*k]+segTree[2*k+1];
    }
  }
};

int main () {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);

  tint n, q;
        cin >> n >> q;
```

```
  SegmentTree segTree (n);

  forn (i, n) {
    tint temp; cin >> temp;
    segTree.update(i, temp);
  }

  forn (i, q) {
    tint func, a, b;
    cin >> func >> a >> b;

    if (func == 1) segTree.update(a-1, b);
    else cout << segTree.retrieve(a-1, b-1) << "\n";
  }
}
```

7.3. CSES 1649 - Dynamic Range Minimum Queries

Link al Problema: https://cses.fi/problemset/task/1649

```
// AC - Tiempo 0.15 s

#include <iostream>
#include <vector>
#include <algorithm>

#define forn(i, n) for(tint i = 0; i < tint(n); i++)

using namespace std;
using tint = long long;

template<typename T>
struct SegmentTree {
 private:
  static tint nextPow2(tint x) {
    tint ans = 1;
    while (ans < x) ans *= 2;
    return ans;
  }

  static constexpr tint parent(tint x) { return x / 2; }
  static constexpr tint leftChild(tint x) { return 2 * x; }
  static constexpr tint rightChild(tint x) { return 2 * x + 1; }

 public:
  vector<T> st;

  SegmentTree(tint n) {
    n = nextPow2(n);
    st.resize(2 * n, T::identity());
  }

  tint size() const { return st.size() / 2; }

  void update(tint idx, T val) {
    idx += size();
    st[idx] = val;
    for (tint i = parent(idx); i > 0; i = parent(i)) {
      st[i] = st[leftChild(i)] >> st[rightChild(i)];
```

```
    }
  }

  T query(tint start, tint end, tint idx = 1, tint left = -1,
          tint right = -1) const {
    if (idx == 1) left = 0, right = size();

    if (start <= left and right <= end) return st[idx];
    if (end <= left or right <= start) return T::identity();

    tint mid = (left + right) / 2;
    return query(start, end, leftChild(idx), left, mid) >>
           query(start, end, rightChild(idx), mid, right);
  }
};

struct Node {
  tint val;
  Node(tint x): val{x} {}
  static Node identity() {return Node(2e15); }
  Node operator>> (const Node &other) const { return Node(min(val, other.val)); }
};

int main () {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);

  tint n, q;
  cin >> n >> q;

  SegmentTree<Node> segTree (n);

  forn(i,n) {
    tint temp; cin >> temp;
    segTree.update(i, Node(temp));
  }

  vector<tint> results;

  for (int i = 0; i < q; i++) {
    tint a, b, func;
    cin >> func >> a >> b;

    if (func == 1) segTree.update(a-1, Node(b));
    else results.push_back(segTree.query(a-1, b).val);
  }

  for (auto g: results) cout << g << "\n";
}
```

7.4. CSES 1143 - Hotel Queries

Link al problema: https://cses.fi/problemset/task/1143

```
// AC - Tiempo 0.11 s

#include <iostream>
#include <vector>

#define forn(i, n) for(tint i = 0; i < tint(n); i++)
```

```
using tint = long long;
using namespace std;

tint n, m, s = 1;
vector<tint> t;

void update (int a, int val) {
  a += s;
  t[a] = val; a /= 2;
  while (a >= 1) {
        t[a] = max(t[a*2], t[a*2+1]);
        a /= 2;
  }
}

tint occupy (int amount) {
  if (t[1] < amount) return 0;
  tint i = 1;
  while (i < s) {
        if (t[i*2] >= amount) i*= 2;
    else i = i * 2 + 1;
  }
  i -= s;
  update (i, t[i + s] - amount);
  return i+1;
}

int main () {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);

  cin >> n >> m;
  while (s <= n) s*=2;

  t = vector<tint> (2*s, -1);
  vector<tint> answers(m);

  forn (i, n) {
    tint val; cin >> val;
    update(i, val);
  }

  forn (i, m) {
    tint temp; cin >> temp;
    answers[i] = occupy(temp);
  }

  for (auto x: answers) cout << x << " ";
}
```

7.5. CSES 1144 - Salary Queries

Link al problema: https://cses.fi/problemset/task/1144

```
#include <algorithm>
#include <iostream>
#include <map>
#include <vector>
```

```cpp
#define forn(i, n) for (int i = 0; i < int(n); i++)
#define DBG(x) cerr << #x << " = " << x << endl

using namespace std;

vector<int> salaryValues;
vector<int> possibleSalaryValues;
vector<tuple<bool, int, int>> queries;
map<int, int> valueCompression;

int compressionN;

struct SegmentTree {
  int segN;
  vector<int> segTree;

  SegmentTree(int n) {
    segN = 1;
    while (segN < n) segN *= 2;
    segTree.resize(segN * 2);
  }

  void update(int pos, int val) {
    pos = valueCompression[pos] + segN;
    segTree[pos] += val;
    for (pos /= 2; pos >= 1; pos /= 2) {
      segTree[pos] = segTree[pos * 2] + segTree[pos * 2 + 1];
    }
  }

  int getNoCompression(int a, int b) {
    a += segN;
    b += segN;
    int suma = 0;
    while (a <= b) {
      if (a % 2 == 1) suma += segTree[a++];
      if (b % 2 == 0) suma += segTree[b--];
      a /= 2;
      b /= 2;
    }
    return suma;
  }
};

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);

  int n, k;
  cin >> n >> k;

  salaryValues.resize(n);
  queries.resize(k);

  forn(i, n) {
    cin >> salaryValues[i];
    possibleSalaryValues.push_back(salaryValues[i]);
  }

  forn(i, k) {
```

```
    char type;
    int a, b;
    cin >> type >> a >> b;
    if (type == '!') {
      possibleSalaryValues.push_back(b);
      queries[i] = make_tuple(false, a, b);
    } else {
      queries[i] = make_tuple(true, a, b);
    }
  }

  sort(possibleSalaryValues.begin(), possibleSalaryValues.end());

  vector<int> aux;
  forn(i, possibleSalaryValues.size())
    if (aux.empty() || possibleSalaryValues[i] != aux.back())
      aux.push_back(possibleSalaryValues[i]);
  possibleSalaryValues = aux;

  compressionN = int(possibleSalaryValues.size());
  forn(i, compressionN) valueCompression[possibleSalaryValues[i]] = i;
  SegmentTree segTree(compressionN);

  for (auto &e : salaryValues) {
    segTree.update(e, 1);
  }
  for (auto &q : queries) {
    bool type;
    int a, b;
    tie(type, a, b) = q;

    if (type == false) {
      a--;
      segTree.update(salaryValues[a], -1);
      salaryValues[a] = b;
      segTree.update(salaryValues[a], 1);
    }

    if (type == true) {
      int la = int(lower_bound(possibleSalaryValues.begin(),
        possibleSalaryValues.end(), a) - possibleSalaryValues.begin());
      int lb = int(upper_bound(possibleSalaryValues.begin(),
        possibleSalaryValues.end(), b) - possibleSalaryValues.begin());
      if (lb > 0) lb--;
      cout << segTree.getNoCompression(la, lb) << "\n";
    }
  }
}
```

7.6. CSES 1144 - Salary Queries Dinámico

Link al problema: https://cses.fi/problemset/task/1144

```
// AC - Tiempo 0.80s
#include <iostream>
#include <vector>

#define forn(i, n) for(tint i = 0; i < tint(n); i++)

using namespace std;
```

```
using tint = int;

struct Node {
  tint value;
  Node *left, *right;
  Node(tint value): value(value) {}
  Node(): value(0) {}
};

struct SegmentTree {
  tint n, segN;
  Node* root;

  SegmentTree (tint _n) {
    n = _n; segN = 1;
    while (segN < n) segN *= 2;
    root = new Node(0);
  }

  tint query(Node &curr, tint a, tint b, tint k, tint x, tint y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return curr.value;
    tint d = (x+y)/2;

    tint temp = 0;
    if (curr.left != nullptr) temp += query(*curr.left,a,b,2*k,x,d);
    if (curr.right != nullptr) temp += query(*curr.right,a,b,2*k+1,d+1,y);
    return temp;
  }

  tint query(tint a, tint b) {
    return query(*root, a, b, 1, 0, n-1);
  }

  void update(Node &curr, tint pos, tint k, tint x, tint y, tint val) {
    if (x == y) { curr.value += val; return; }
    tint d = (x + y) / 2;

    if (pos <= d) {
      if (curr.left == nullptr) curr.left = new Node(0);
      update(*curr.left, pos, k*2, x, d, val);
    }
    else {
      if (curr.right == nullptr) curr.right = new Node(0);
      update(*curr.right, pos, k*2+1, d+1, y, val);
    }

    curr.value = (curr.left != nullptr? curr.left->value: 0) +
        (curr.right != nullptr? curr.right->value: 0);
  }

  void update(tint pos, tint val) {
    update(*root, pos, 1, 0, n-1, val);
  }
};

int main () {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);
```

```
  tint n, q;
  cin >> n >> q;

  SegmentTree segTree (1e9+1);
  vector<tint> valores (n+1);

  forn (i, n) {
    cin >> valores[i+1];
    segTree.update(valores[i+1], 1);
  }

  forn (i, q) {
    char func;
        tint a, b;
        cin >> func >> a >> b;

        if (func == '?') {
    cout << segTree.query(a, b) << "\n";

  } else {
    segTree.update(valores[a], -1);
    valores[a] = b;
    segTree.update(valores[a], 1);
  }
}
```

7.7. CSES 1735 - Range Updates and Sums

Link al problema: https://cses.fi/problemset/task/1735

```
// AC - Time 0.36s

#include <iostream>
#include <algorithm>
#include <vector>

#define forn(i, n) for (int i = 0; i < int(n); i++)

using namespace std;

typedef long long tint;

struct Node {
  tint suma = 0, sumaLazy = 0, setValueLazy = 0;

  void propagate(Node &o) {
    if (o.setValueLazy > 0) {
      sumaLazy = o.sumaLazy;
      setValueLazy = o.setValueLazy;
    } else {
      sumaLazy += o.sumaLazy;
    }
  }

  void updateSuma(Node &o1, Node &o2, int l, int r) {
    int m = (l + r) / 2;
    suma = o1.evaluate(l, m) + o2.evaluate(m + 1, r);
  }

  void setValue(int val, bool setValue) {
```

```
    if (setValue) {
      setValueLazy = val;
      sumaLazy = 0;
    } else {
      sumaLazy += val;
    }
  }

  tint evaluate(int l, int r) {
    const tint abarca = tint(r - l + 1);
    if (setValueLazy > 0)
      return abarca * setValueLazy + abarca * sumaLazy;
    else
      return abarca * sumaLazy + suma;
  }

  void clearLazy() {
    sumaLazy = 0;
    setValueLazy = 0;
  }
};

struct LazySegmentTree {
  int segN;
  vector<Node> segTree;

  LazySegmentTree(int n) {
    segN = 1;
    while (segN < n) segN *= 2;
    segTree.resize(segN * 2);
  }

  void update(int a, int b, int val, bool setValue) {
    update(a, b, val, setValue, 1, 0, segN - 1);
  }

  void update(int a, int b, int val, bool setValue, int k, int l, int r) {
    if (a > r || b < l) return;

    if (a <= l && r <= b) {
      segTree[k].setValue(val, setValue);
      return;
    }

    segTree[k * 2].propagate(segTree[k]);
    segTree[k * 2 + 1].propagate(segTree[k]);
    segTree[k].clearLazy();

    tint m = (l + r) / 2;
    update(a, b, val, setValue, k * 2, l, m);
    update(a, b, val, setValue, k * 2 + 1, m + 1, r);
    segTree[k].updateSuma(segTree[k * 2], segTree[k * 2 + 1], l, r);
  }

  tint query(int a, int b) { return query(a, b, 1, 0, segN - 1); }

  tint query(int a, int b, int k, int l, int r) {
    if (b < l || a > r) return 0;
    if (a <= l && r <= b) return segTree[k].evaluate(l, r);
```

```
    segTree[k * 2].propagate(segTree[k]);
    segTree[k * 2 + 1].propagate(segTree[k]);
    segTree[k].clearLazy();

    tint m = (l + r) / 2;
    tint q1 = query(a, b, k * 2, l, m);
    tint q2 = query(a, b, k * 2 + 1, m + 1, r);

    segTree[k].updateSuma(segTree[k * 2], segTree[k * 2 + 1], l, r);
    return q1 + q2;
  }
};

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);

  int n, q;
  cin >> n >> q;
  LazySegmentTree tree(n);

  forn(i, n) {
    int temp;
    cin >> temp;
    tree.update(i, i, temp, true);
  }

  forn(i, q) {
    int op;
    cin >> op;
    if (op == 1) {
      int a, b, x;
      cin >> a >> b >> x;
      a--; b--;
      tree.update(a, b, x, false);
    } else if (op == 2) {
      int a, b, x;
      cin >> a >> b >> x;
      a--; b--;
      tree.update(a, b, x, true);
    } else {
      int a, b;
      cin >> a >> b;
      a--; b--;
      cout << tree.query(a, b) << "\n";
    }
  }
}
```

7.8. CSES 1736 - Polinomial Queries

Link al problema: https://cses.fi/problemset/task/1736

```
#include <vector>
#include <iostream>

#define forn(i, n) for (int i = 0; i < int(n); i++)

using namespace std;
using tint = long long;
```

```
int largo = 1;
int segN = 1;

struct node {
  tint suma = 0, lazy = 0, cantidad = 0;
};

vector<node> segTree;

// Logica para evaluar un nodo
tint eval(tint pos, tint x, tint y) {
  tint cantHijos = y - x + 1;
  tint sumaPoli = (cantHijos * (cantHijos - 1)) / 2;
  tint parcial =
      (sumaPoli * segTree[pos].cantidad) + (segTree[pos].lazy * cantHijos);
  return segTree[pos].suma + parcial;
}

void update(int pos, int a, int b, tint value, int x = 0, int y = segN - 1) {
  // Si esta en el rango, sumar al lazy y retornar
  if (a <= x && b >= y) {
    segTree[pos].lazy += value;
    segTree[pos].cantidad++;
    return;
  }

  // Si esta por fuera, retornar
  if (a > y || b < x) return;

  tint med = (y - x + 1) / 2;
  // Propagación izquierda
  segTree[pos * 2].lazy += segTree[pos].lazy;
  segTree[pos * 2].cantidad += segTree[pos].cantidad;
  // Propagación derecha
  segTree[pos * 2 + 1].lazy +=
      (segTree[pos].lazy + segTree[pos].cantidad * med);
  segTree[pos * 2 + 1].cantidad += segTree[pos].cantidad;
  // Reseteo
  segTree[pos].lazy = 0;
  segTree[pos].cantidad = 0;

  // Recursión
  tint prom = (x + y) / 2;
  tint toLeft = max(0ll, prom + 1 - max(a, x));
  update(pos * 2, a, b, value, x, prom);
  update(pos * 2 + 1, a, b, value + toLeft, prom + 1, y);

  // Actualización del nodo actual
  segTree[pos].suma = eval(pos * 2, x, prom) + eval(pos * 2 + 1, prom + 1, y);
}

tint query(int pos, int a, int b, int x = 0, int y = segN - 1) {
  // Returnear suma de los hijos
  if (a <= x && b >= y) return eval(pos, x, y);

  // Returnear nada porque no está en el rango
  if (a > y || b < x) return 0;

  tint med = (y - x + 1) / 2;
```

```
  // Propagación izquierda
  segTree[pos * 2].lazy += segTree[pos].lazy;
  segTree[pos * 2].cantidad += segTree[pos].cantidad;
  // Propagación derecha
  segTree[pos * 2 + 1].lazy +=
      (segTree[pos].lazy + segTree[pos].cantidad * med);
  segTree[pos * 2 + 1].cantidad += segTree[pos].cantidad;
  // Reseteo
  segTree[pos].lazy = 0;
  segTree[pos].cantidad = 0;

  // Si es que no está totalmente contenido en el rango, retornar los hijos
  int prom = (x + y) / 2;
  segTree[pos].suma = eval(pos * 2, x, prom) + eval(pos * 2 + 1, prom + 1, y);
  return query(pos * 2, a, b, x, prom) + query(pos * 2 + 1, a, b, prom + 1, y);
}

int main() {
  cin.sync_with_stdio(false);
  cin.tie(nullptr);

  int N, Q;
  cin >> N >> Q;

  while (segN < N) segN *= 2;
  largo = segN * 2;
  segTree.resize(largo);

  forn(i, N) {
    int temp;
    cin >> temp;
    update(1, i, i, temp);
  }

  forn(i, Q) {
    int t;
    cin >> t;
    if (t == 1) {
      // Update
      int a, b;
      cin >> a >> b;
      update(1, a - 1, b - 1, 1);
    } else {
      // Query
      int a, b;
      cin >> a >> b;
      cout << query(1, a - 1, b - 1) << "\n";
    }
  }
}
```

7.9. CSES 1737 - Range Queries and Copies

Link al problema: https://cses.fi/problemset/task/1737

```
// AC 0.58s

#include <iostream>
#include <vector>
#define forn(i, n) for(tint i = 0; i < tint(n); i++)
```

```cpp
using namespace std;
using tint = long long;

struct Node {
  tint val;
  Node *left, *right;

  // Instanciador sin hijos para valores base
  Node (tint _val): val(_val), left(nullptr), right(nullptr) {}

  // Instanciador que aplica operación sobre nodos hijos, en este caso la suma.
  Node (Node *_left, Node *_right) {
    left = _left; right = _right;
    val = 0;
    if (left) val += left->val;
    if (right) val += right->val;
  }
};

struct PSegmentTree {
  tint n;
  Node *root;

  PSegmentTree (tint _n): n(_n), root(new Node(tint(0))) {}
  PSegmentTree (PSegmentTree *copy): n(copy->n), root(copy->root) {}

  PSegmentTree *copy () {
    return new PSegmentTree(this);
  }

  Node *update(Node *node, tint val, tint pos, tint x, tint y) {
    if (x == y) { return new Node(val); }
    tint mid = (x + y) / 2;

    if (node == nullptr) {
      if (pos <= mid) return new Node(update(nullptr, val, pos, x, mid), nullptr);
      else return new Node(nullptr, update(nullptr, val, pos, mid+1, y));
    }

    if (pos <= mid) return new Node(update(node->left, val, pos, x, mid), node->right);
    else return new Node(node->left, update(node->right, val, pos, mid+1, y));
  }

  void update (tint pos, tint val) {
    root = update(root, val, pos, 0, n-1);
  }

  tint query (Node *node, tint a, tint b, tint x, tint y) {
    if (b < x || a > y) return 0;
    if (a <= x && y <= b) return node->val;

    tint mid = (x+y) / 2;
    tint tempAns = 0;

    if (node->left) tempAns += query(node->left, a, b, x, mid);
    if (node->right) tempAns += query(node->right, a, b, mid+1, y);
    return tempAns;
  }
```

```
  tint query (tint a, tint b) {
    return query(root, a, b, 0, n-1);
  }
};

int main () {
  ios_base::sync_with_stdio(false);
  cin.tie(nullptr);

  tint n, q;
  cin >> n >> q;

  PSegmentTree segTreeInicial (n+1);

  forn (i, n) {
    tint val; cin >> val;
    segTreeInicial.update(i+1, val);
  }

  vector<PSegmentTree*> segTrees;
  segTrees.push_back(nullptr);
  segTrees.push_back(&segTreeInicial);

  forn (i, q) {
    tint type; cin >> type;
    if (type == 1) {
      tint k, a, x;
      cin >> k >> a >> x;
      segTrees[k]->update(a, x);
    } else if (type == 2) {
      tint k, a, b;
      cin >> k >> a >> b;
      cout << segTrees[k]->query(a, b) << "\n";
    } else {
      tint k; cin >> k;
      segTrees.push_back(segTrees[k]->copy());
    }
  }
}
```