Data Structures II: Sparse Table

Sparse Table

- Estructura de datos que nos permite contestar queries en rango bajo algún operador binario o función que cumpla con la propiedad **asociativa.**

- Sin embargo, solo se puede aplicar sparse table cuando **no hay updates**. Es decir, sobre un arreglo inmutable.

Definamos el operador binario ★ que cumple con la propiedad asociativa. Entonces las queries en rango serán de la siguiente forma :

$$\rightarrow query\,(l,r) = A[l] \bigstar A[l+1] \bigstar A[l+2] \bigstar \cdots \bigstar A[r-1] \bigstar A[r]$$

Si lo ponemos en términos de una función, podemos expresarlo de la siguiente forma. Sea la función asociativa $f\,(a,b)$

$$\rightarrow query\,(l,r) = f\,(A[l], A[l+1], A[l+2], \ldots, A[r-1], A[r])$$

Sparse Table – Idea clave

Para poder responder las queries eficientemente usaremos la propiedad de que cualquier número puede expresarse como la suma de distintas potencias de 2.

Entonces podemos dividir un rango en rangos más chicos de longitud de potencias de 2.

Por ejemplo, supongamos que tuviéramos el rango [5 , 17]. La longitud de ese rango es de tamaño 13. $13 = 1101_2 = 8 + 4 + 1$. Así que podemos dividirlo en rangos de tamaño 8 , 4 , 1.

Algunas formas posibles de dividirlo serían :

- $[5\,,17] = [5\,,5] \cup [6\,,9] \cup [10\,,17]$
- $[5\,,17] = [5\,,12] \cup [13\,,16] \cup [17\,,17]$

Sparse Table – Preprocesamiento

La ventaja de dividir el rango en potencias de 2, es que podríamos preprocesar todas las respuestas para cualquier rango que tenga tamaño igual a una potencia de 2.
Aparentemente esto suena demasiado costoso en memoria y tiempo, ya que tendremos que precalcular todos los rangos de tamaño 1 , 2 , 4 , 8 , 16 , … Pero hagamos los cálculos de cuántos rangos puede haber de este tipo.

La cantidad de rangos de tamaño x para cualquier x, es menor o igual que n

$$\Longrightarrow \#\ de\ rangos\ de\ tamaño\ potencia\ de\ 2 \leq \underbrace{n + n + n + \ \ldots + n}_{\lfloor \log n \rfloor + 1 \text{ veces}} \leq n\ (log\ n + 1)$$

Vemos que esa cantidad sí podría entrar en memoria. Lo que faltaría sería ¿cómo calcularlos eficientemente?

Sparse Table – Preprocesamiento

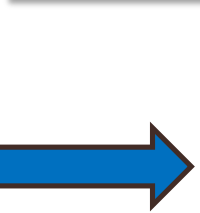
Para poder calcular la respuesta de cada rango de potencia de 2 lo haremos al estilo de programación dinámica.

Definamos la matriz $st[i][j]$ como la respuesta para el rango que empieza en i y tiene tamaño 2^j , es decir en el rango $[i\ , i + 2^j - 1]$

Donde, $0 \leq i < n$, $0 \leq j \leq [\log_2 n]$

Podemos hallar alguna fórmula recursiva dividiendo un rango de tamaño 2^j en 2 rangos de tamaño 2^{j-1}

Índices	1	2	3	4	5	6	7	8	9
A[ ]									



$$st[i][j] = st[i][j-1] \bigstar st\left[i + 2^{j-1}\right][j-1]$$

Sparse Table – Preprocesamiento

Con esa fórmula podemos armar la matriz fácilmente con dos for anidados. Utilicemos una función asociativa cualquiera denominada f , cuya complejidad es $O(|f|)$.

```
void build(vector<Long> &A){ // O(|f| n log n )
    Long n = A.size();
    for(Long i = 0; i < n; i++){
        st[i][0] = A[i]; //base case
    }

    for(Long j = 1; (1 << j) <= n; j++){
        for(Long i = 0; i + (1 << j) <= n; i++){
            st[i][j] = f(st[i][j-1], st[i + (1 << (j-1))][j-1]);
        }
    }
}
```

Comentario : Normalmente $O(|f|) = O(1)$, a excepción de algunas funciones como el gcd

Sparse Table – Query

Luego, para contestar las queries, lo único que hay que hacer es dividir el rango en potencias de 2. Lo podemos hacer de distintas formas , como se mencionó en una diapositiva anterior. Por ejemplo, el rango $[5\,,17] = [5\,,5] \cup [6\,,9] \cup [10\,,17]$

Podemos implementarlo en código de una forma similar a como lo hicimos con BIT, quitando en cada iteración el bit menos significativo del rango.

Para el ejemplo, tendríamos podemos dividir el rango de tamaño 13 de la siguiente forma:

$query(5{,}17) = st[5][0] + st[6][2] + st[10][3]$

$13 = 1101_2$

Podemos notar que , así como en BIT, nosotros estamos recorriendo los bits del rango (en este caso, el rango de tamaño 13). Por lo tanto, en el peor caso estaríamos recorriendo todos los bits, lo que nos daría una complejidad $O(\log n)$

Sparse Table – Query

```
Long query(Long l, Long r){ //O(|f| log n)
    Long ans = st[l][0];
    if(l == r) return ans;
    l++;
    Long range = r - l + 1;

    while( range > 0){
        Long step = range & -range;
        Long j = __builtin_ctz(step);
        ans = f(ans , st[l][j]);
        l += step;
        range -= step;
    }

    return ans;
}
```

Sparse Table – Queries especiales

Sin embargo, la verdad es que la mayoría de personas no usa la query anterior. Esto es porque ya existe una estructura de datos más general para poder responder queries (y también updates) de funciones asociativas y con la misma complejidad. Esa estructura se llama **Segment Tree** y la veremos en posteriores clases.

Pero la verdadera potencia del sparse table está en ciertas funciones especiales : las **funciones (u operadores) idempotentes**.
Las funciones idempotentes son aquellas en las cuales aplicar la función (o el operador) varias veces no cambia el resultado.

Dicho de una manera formal :

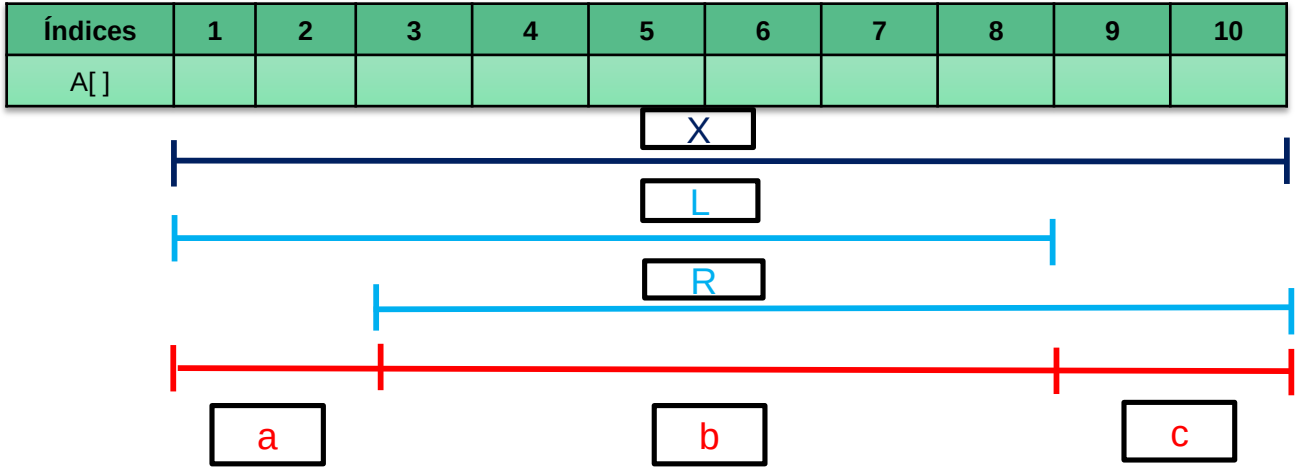
Son aquellas funciones f tal que $f(x, x) = x$

En caso de un operador se cumpliría que : $x \bigstar x = x$

Ejemplos Funciones idempotentes
max
min
gcd
and
or

Sparse Table – Queries especiales

¿Qué tienen de especial las funciones idempotentes con el sparse table?
Que podemos superponer los rangos sin alterar la respuesta.
Por ejemplo, supongamos que queremos hallar la respuesta del rango **X** en base los rangos **L** y **R**



Del gráfica, es claro que se cumple que $X = a \star b \star c$
Por propiedad de idempotencia tenemos que $X = a \star b \star b \star c = L \star R$

Sparse Table – Queries especiales

Lo anterior nos sirve de la siguiente forma: En vez de dividir el rango en $O(\log n)$ rangos más pequeños. Podemos dividirlo solo en 2 rangos superpuestos. Claro que para que esto funcione, tendríamos que encontrar una potencia de 2 que supere o sea igual a la mitad del tamaño del rango original. Nuestro mejor intento sería usar el **bit más significativo**, pero ¿será suficiente para cubrir la mitad del rango original?

Sí. Demostrémoslo de esta forma :

Supongamos que la posición del **bit más significativo** de la longitud del rango es k.

Podemos afirmar que $longitud\ del\ rango < 2^{k+1}$. Ya que el peor caso sucede cuando todos los bits están prendidos.

$$\Rightarrow \frac{longitud\ del\ rango}{2} < 2^{k}$$, que es lo que queríamos demostrar

Sparse Table – Queries especiales

Por ejemplo, supongamos que queremos hallar la query en el rango $[1,15]$
$(15 = 1111_2)$

Si la función NO fuera idempotente, tendríamos que checar 4 rangos $(O(\log n))$: $[1,1], [2,3], [4,7], [8,15]$

Pero si la función SÍ es idempotente, podemos aprovechar y solo checar 2 rangos de tamaño 8 que serían $[1,8]$ y $[9,15]$. Lo que nos daría una poderosa query en $O(1)$ (a menos que la función asociativa tuviera complejidad extra)

En código, sería lo siguiente :

```
Long query(Long l, Long r){ //O(|f|)
    //special cases : idempotent(  min, max, gcd, and , or)
    Long len = r - l + 1;
    Long lg = 31 - (__builtin_clz(len));
    return f(st[l][lg], st[r - (1 << lg) + 1][lg]);
}
```

Problemas

SPOJ – RMQSQ

Referencias

- cp-algorithms: https://cp-algorithms.com/data_structures/sparse-table.html