Estructuras para operadores asociativos en intervalos

Sebastián Marín

Facultad de Matemática, Física, Astronomía y Computación
Universidad Nacional de Córdoba

Training Camp 2019

Contenidos

Contenidos

¿Qué es un operador asociativo?

Definición

Un operador binario $\triangleright$ se dice **asociativo**, si siempre se cumple que $(a \triangleright b) \triangleright c = a \triangleright (b \triangleright c)$

Ejemplos:

- **Suma $+$ de enteros (o reales)**
- Producto $\cdot$ de enteros (o reales)
- **Máximo / Mínimo de enteros (o reales)**
- Operadores lógicos “or” y “and”

¿Qué es un operador asociativo?

Definición

Un operador binario $\triangleright$ se dice **asociativo**, si siempre se cumple que $(a \triangleright b) \triangleright c = a \triangleright (b \triangleright c)$

Ejemplos:

- **Suma $+$ de enteros (o reales)**
- Producto $\cdot$ de enteros (o reales)
- **Máximo / Mínimo de enteros (o reales)**
- Operadores lógicos “or” y “and”

Más ejemplos

Notar que en general el operador podría no ser conmutativo:

- Producto de matrices
- Composición de funciones
- Concatenación de Strings

Elemento neutro

Definición

Un **elemento neutro** para un operador $\triangleright$ es un e con la propiedad de que para cualquier x siempre se cumple $x \triangleright e = e \triangleright x = x$

Ejemplos:

- 0 para la suma
- 1 para el producto
- $+\infty$ / $-\infty$ para Min / Max
- *True* / *False* para And / Or
- I (matriz identidad) para el producto de matrices
- *id* (función identidad) para la composición de funciones
- "" (string vacío) para la concatenación de Strings

Elemento neutro

Definición

Un **elemento neutro** para un operador $\triangleright$ es un e con la propiedad de que para cualquier x siempre se cumple $x \triangleright e = e \triangleright x = x$

Ejemplos:

- 0 para la suma
- 1 para el producto
- $+\infty$ / $-\infty$ para Min / Max
- *True* / *False* para And / Or
- I (matriz identidad) para el producto de matrices
- *id* (función identidad) para la composición de funciones
- "" (string vacío) para la concatenación de Strings

Elemento neutro (cont)

Asumiremos por comodidad siempre que sea necesario que existe un elemento neutro.
Notar que si no existiera uno para la operación con la que estemos trabajando, podemos siempre simular uno con un valor “NULL” y “poniendo un if”, de forma tal que una operación de NULL con x deja al x intacto como resultado.

Contenidos

Problema de consulta de prefijos

Problema:

Dado un arreglo v de n posiciones, numeradas de 0 a $n-1$, y un operador asociativo $\triangleright$, es posible definir para $0 \leq i \leq n$, la **consulta** $P(i)$, cuyo resultado es $P(i) = v_0 \triangleright v_1 \triangleright \cdots \triangleright v_{i-1}$

Es decir, se consulta el resultado del operador $\triangleright$ sobre los primeros i elementos.

- Notar que cuando el rango es vacío, la respuesta es el elemento neutro (definición).
- Cuando el arreglo inicial nunca se modifica, se dice que el problema es **estático**.
- Si el arreglo se va modificando (generalmente, cambiando el valor en una cierta posición) con el paso del tiempo entre consultas, se dice que el problema es **dinámico**.

Problema de consulta de prefijos

Problema:

Dado un arreglo v de n posiciones, numeradas de 0 a $n-1$, y un operador asociativo $\triangleright$, es posible definir para $0 \leq i \leq n$, la **consulta** $P(i)$, cuyo resultado es $P(i) = v_0 \triangleright v_1 \triangleright \cdots \triangleright v_{i-1}$

Es decir, se consulta el resultado del operador $\triangleright$ sobre los primeros i elementos.

- Notar que cuando el rango es vacío, la respuesta es el elemento neutro (definición).
- Cuando el arreglo inicial nunca se modifica, se dice que el problema es **estático**.
- Si el arreglo se va modificando (generalmente, cambiando el valor en una cierta posición) con el paso del tiempo entre consultas, se dice que el problema es **dinámico**.

Problema de consulta de intervalos (rangos)

Problema:

Dado un arreglo v de n posiciones, numeradas de 0 a $n-1$, y un operador asociativo $\triangleright$, es posible definir para $0 \leq i \leq j \leq n$, la **consulta** $R(i,j)$, cuyo resultado es $R(i,j) = v_i \triangleright v_{i+1} \triangleright \cdots \triangleright v_{j-1}$

Es decir, se consulta el resultado del operador $\triangleright$ sobre el rango de elementos $[i,j)$.

- Nuevamente cuando el rango es vacío, definimos la respuesta como el elemento neutro.
- Al igual que antes, el problema puede ser estático o dinámico según se realicen modificaciones en el arreglo.

Problema de consulta de intervalos (rangos)

Problema:

Dado un arreglo v de n posiciones, numeradas de 0 a $n-1$, y un operador asociativo $\triangleright$, es posible definir para $0 \leq i \leq j \leq n$, la **consulta** $R(i,j)$, cuyo resultado es $R(i,j) = v_i \triangleright v_{i+1} \triangleright \cdots \triangleright v_{j-1}$

Es decir, se consulta el resultado del operador $\triangleright$ sobre el rango de elementos $[i,j)$.

- Nuevamente cuando el rango es vacío, definimos la respuesta como el elemento neutro.
- Al igual que antes, el problema puede ser estático o dinámico según se realicen modificaciones en el arreglo.

Problema de consulta de intervalos (cont)

Una observación importante es que cuando el operador tiene inverso (Ejemplo: **la resta**, para el $+$), es posible resolver este problema utilizando estructuras que resuelvan consulta de prefijos, mediante:

$R(i,j) = inv(P(i)) \triangleright P(j)$ (por ejemplo, $P(j) - P(i)$ para el operador +)

Las estructuras de datos

- Veremos estructuras que resuelven cada uno de los 4 problemas planteados, en todos los casos para un operador asociativo completamente arbitrario.

	Prefijos	Intervalos
Estático	**Tabla Aditiva**	**Sparse Table**
Dinámica	Fenwick Tree	**Segment Tree**

- "Orden de dificultad subjetivo":
TablaAditiva < **SparseTable** < FenwickTree < **SegmentTree**

Contenidos

Problema

- Dado un arreglo v de elementos, queremos encontrar una estructura eficiente que permita calcular $v_0 \triangleright v_1 \triangleright v_2 \triangleright \cdots \triangleright v_{i-1}$ para cualquier i.
- Supondremos desde ahora en toda esta parte que el operador es la suma: $\triangleright \equiv +$
- Todo lo que hagamos para cálculos de prefijos se traslada idéntico para cualquier otro operador asociativo.
- Además, para cualquier operador con inversos, podremos calcular intervalos de la misma manera que lo haremos.

Arreglo acumulado

Definición

Dado un arreglo unidimensional v de n elementos $v_0, v_1, \cdots, v_{n-1}$, definimos el **arreglo acumulado** de v como el arreglo unidimensional A de $n+1$ elementos $A_0, \cdots, A_n$ y tal que:

$$A_i = \sum_{j=0}^{i-1} v_j$$

Notar que A es muy fácil de calcular en tiempo lineal utilizando programación dinámica ("un for sumando"):

- $A_0 = 0$
- $A_{i+1} = v_i + A_i, \forall\ 0 \leq i < n$

Respuesta de los queries

- Una vez computada la tabla, las consultas $P(i)$ son directamente los valores A_i calculados.
- La respuesta a un query $R(i, j)$ de intervalo $[i, j)$ se obtiene directamente como $A_j - A_i$
- Luego podemos responder cada query en tiempo constante, computando una sola resta entre dos valores del arreglo acumulado.

Contenidos

Arreglo acumulado

Definición

Dado un arreglo bidimensional v de $n \times m$ elementos $v_{i,j}$ con $0 \leq i < n, 0 \leq j < m$, definimos el **arreglo acumulado** de v como el arreglo bidimensional A de $(n+1) \times (m+1)$ elementos tal que:

$$A_{i,j} = \sum_{a=0}^{i-1} \sum_{b=0}^{j-1} v_{a,b}$$

¿Podremos calcular fácilmente A en tiempo lineal como en el caso unidimensional? **¡Sí!** Utilizando programación dinámica.

- $A_{0,j} = A_{i,0} = 0 \quad \forall\, 0 \leq i \leq n, 0 \leq j \leq m$
- $A_{i+1,j+1} = v_{i,j} + A_{i,j+1} + A_{i+1,j} - A_{i,j} \quad \forall\, 0 \leq i < n, 0 \leq j < m$

Respuesta de los queries

- Las queries de rectángulos generales a la tabla aditiva vendrán dadas por rectángulos $[i_1, i_2) \times [j_1, j_2)$ con $0 \leq i_1 \leq i_2 \leq n, 0 \leq j_1 \leq j_2 \leq m$.
- La respuesta al query $[i_1, i_2) \times [j_1, j_2)$ sera

$$Q(i_1, i_2, j_1, j_2) = \sum_{a=i_1}^{i_2-1} \sum_{b=j_1}^{j_2-1} v_{a,b}$$

- Pero de manera similar a como hicimos para calcular el arreglo acumulado, resulta que:

$$Q(i_1, i_2, j_1, j_2) = A_{i_2,j_2} - A_{i_1,j_2} - A_{i_2,j_1} + A_{i_1,j_1}$$

- Luego podemos responder cada query en tiempo constante computando sumas y restas de cuatro valores del arreglo acumulado.

Contenidos

Arreglo acumulado

El caso general es completamente análogo... Dejamos las cuentas escritas rápidamente.

Definición

Dado un arreglo v de $n_1 \times \cdots \times n_r$ elementos $v_{i_1,\ldots,i_r}$ con $0 \leq i_k < n_k, 1 \leq k \leq r$, definimos el **arreglo acumulado** de v como el arreglo A de $(n_1 + 1) \times \cdots \times (n_r + 1)$ elementos tal que:

$$A_{i_1,\ldots,i_r} = \sum_{k_1=0}^{i_1-1} \cdots \sum_{k_r=0}^{i_r-1} v_{k_1,\ldots,k_r}$$

Arreglo acumulado (Cálculo usando programación dinámica)

El cálculo de A mediante programación dinámica viene dado por:

- $$V_{i_1,\cdots,i_r} = 0 \quad \text{si para algún } k \text{ resulta } i_k = 0$$
- $$A_{i_1+1,\cdots,i_r+1} = v_{i_1,\cdots,i_r} + \sum_{d_1=0}^{1} \cdots \sum_{d_r=0}^{1} (-1)^{r+1+\sum_{k=1}^{r} d_k} A_{i_1+d_1,\cdots,i_r+d_r}$$

Respuesta de los queries

- Las queries a la tabla aditiva vendrán dadas por subarreglos $[i_{10}, i_{11}) \times \cdots \times [i_{r0}, i_{r1})$
- La respuesta al query $[i_{10}, i_{11}) \times \cdots \times [i_{r0}, i_{r1})$ será:
- $$Q(i_{10}, i_{11}, \cdots, i_{r0}, i_{r1}) = \sum_{d_1=0}^{1} \cdots \sum_{d_r=0}^{1} (-1)^{r+\sum_{k=1}^{r} d_k} A_{i_{1\,d_1}, \cdots, i_{r\,d_r}}$$
- Luego podemos responder cada query en tiempo constante computando sumas y restas de 2^r valores del arreglo acumulado.

Contenidos

Aplicaciones prácticas de las tablas aditivas

- Se usan en procesamiento digital de imágenes:
  - Crow, Franklin (1984). “Summed-area tables for texture mapping” SIGGRAPH ’84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques. pp. 207-212.
  - Lewis, J.P. (1995). “Fast template matching.” Proc. Vision Interface. pp. 120-123.
  - Viola, Paul; Jones, Michael (2002). “Robust Real-time Object Detection”. International Journal of Computer Vision.
- Sirven para manipular distribuciones de probabilidad multivariadas, por ejemplo $P(A_x \leq X \leq B_x \wedge A_y \leq Y \leq B_y)$.
- Podemos encontrarlas en inglés como “Summed area table”

Contenidos

Motivación (range minimum query)

- Si no trabajamos con un operador con inverso, las tablas aditivas no sirven para calcular intervalos arbitrarios.
- Por ejemplo si queremos el mínimo de un intervalo en lugar de la suma, ya no podemos “restar” y hacer lo mismo.
- Notación: Usaremos en esta sección el operador Min, y llamaremos $RMQ(i, j)$ a la consulta de rango $[i, j)$
- Idea: Guardar prefijos es poco. Guardemos el acumulado de más intervalos del arreglo.
- Guardaremos el acumulado de todos los intervalos del arreglo con **tamaño potencia de 2**

Contenidos

Estructura : Queries en $O(\lg N)$ ("estilo skiplist")

- Definimos el siguiente arreglo M (**Sparse Table**):

$$M(i,k) = RMQ(i, i+2^k), 0 \leq i \leq n-2^k$$

- Una vez computado el arreglo M, podemos consultar cualquier rango potencia de 2.
- Como en binario todo número N se escribe con $O(\lg N)$ potencias de 2, basta usar $O(\lg L)$ queries para formar un intervalo de longitud L.
- Por ejemplo el intervalo $[2, 13)$, tiene longitud $11 = 8 + 2 + 1$
- Luego podemos formar $RMQ(2, 13) = min(M(2, 3) + M(10, 1) + M(12, 0))$
- Esto funciona con **cualquier operador asociativo**

Estructura : Queries en $O(\lg N)$ ("estilo skiplist")

- Definimos el siguiente arreglo M (**Sparse Table**):

$$M(i,k) = RMQ(i, i+2^k), 0 \leq i \leq n-2^k$$

- Una vez computado el arreglo M, podemos consultar cualquier rango potencia de 2.
- Como en binario todo número N se escribe con $O(\lg N)$ potencias de 2, basta usar $O(\lg L)$ queries para formar un intervalo de longitud L.
- Por ejemplo el intervalo $[2, 13)$, tiene longitud $11 = 8 + 2 + 1$
- Luego podemos formar
  $RMQ(2, 13) = min(M(2, 3) + M(10, 1) + M(12, 0))$
- Esto funciona con **cualquier operador asociativo**

Estructura : Inicialización

- Utilizaremos programación dinámica para computar el arreglo M:

$$M(i,k) = RMQ(i, i+2^k), 0 \leq i \leq n-2^k$$

- Para esto basta notar:
  - $M(i,0) = RMQ(i, i+1) = v_i$
  - $M(i,k+1) = RMQ(i, i+2^{k+1}) = \min(M(i,k), M(i+2^k,k))$
- La tabla M contiene $\Theta(n \lg n)$ elementos, luego la inicialización requiere $\Theta(n \lg n)$ tiempo y memoria.

```
1   for (int i = 0; i < N; i++) M[i][0] = v[i];
2   for (int k = 0; k < ULTIMO_NIVEL; k++) {
3       const int LARGO = (1<<k);
4       for (int i = 0; i <= N - 2 * LARGO; i++)
5           M[i][k+1] = min(M[i][k], M[i + LARGO][k]);
6   }
```

Estructura : Queries en $O(1)$

- Una vez computado el arreglo M, para responder un query basta notar que:

$$RMQ(i,j) = \min(M(i,k), M(j-2^k,k)), 2^k \leq j-i < 2^{k+1}$$

- Luego podemos responder queries en $O(1)$, siempre y cuando podamos computar k en $O(1)$.
- Esto se puede lograr al trabajar con enteros de 32 bits haciendo:
- `k = 31 - __builtin_clz(j-i)` en C++,
- `k = 31 - Integer.numberOfLeadingZeros(j-i)` en Java.
- Notar que esto **no funciona** con cualquier operador asociativo: se debe cumplir $x \triangleright x = x$ para todo x

Contenidos

El problema

LCA

Dado un árbol de n nodos con raíz, una estructura de LCA permite responder rápidamente consultas por el **ancestro común más bajo** entre dos nodos (es decir, el nodo más alejado de la raíz que es ancestro de ambos).

- ¡Sorprendentemente se puede reducir una consulta de LCA a una de RMQ!

Recorrido de DFS

- Utilizando un recorrido de DFS, podemos computar un arreglo v de $2n - 1$ posiciones que indica el orden en que fueron visitados los nodos por el DFS (cada nodo puede aparecer múltiples veces).
- Aprovechando este recorrido podemos computar también la **profundidad** (distancia a la raíz) de cada nodo i, que notaremos P_i.
- También guardaremos el índice de la primera aparición de cada nodo i en v, que notaremos L_i (Cualquier posición servirá, así que es razonable tomar la primera).

Utilizacion del RMQ

- La observación clave consiste en notar que para dos nodos i, j con $i \neq j$:

$$LCA(i, j) = RMQ(\min(L_i, L_j), \max(L_i, L_j) + 1)$$

- Tomamos mínimo y máximo simplemente para asegurar que en la llamada a *RMQ* se especifica un rango válido ($i < j$).
- Notar que en la igualdad anterior, $RMQ(i, j)$ compara los elementos de v por sus valores de profundidad dados por P.
- La complejidad de la transformación de la query de LCA a RMQ es $O(1)$, con lo cual la complejidad final de las operaciones será la del *RMQ* utilizado.

Contenidos

Aplicaciones prácticas

- Mediante LCA con Sparse Table, es posible computar en $O(N \lg N + Q)$, Q consultas de distancia entre nodos en un árbol.
- Sparse Table puede combinarse también con estructuras de Strings (Suffix Array) para responder queries de “prefijo común más largo” entre subcadenas arbitrarias de cadenas dadas. Esto se puede usar en bioinformática para hacer comparaciones de ADN y buscar genes.

Contenidos

Segment Tree

- En esta sección seguiremos llamando “mínimo” a la operación $\triangleright$
- Segment Tree funciona bien con cualquier operador asociativo, en todos los casos

Estructura : Descripción

- Para trabajar con el segment tree asumiremos que n es potencia de 2. De no serlo, basta extender el arreglo v con a lo sumo n elementos neutros adicionales para que sea potencia de 2.
- Utilizaremos un árbol binario completo codificado en un arreglo:
- A_1 guardará la raíz del árbol.
- Para cada i, los dos hijos del elemento A_i serán A_{2i} y A_{2i+1}.
- El padre de un elemento i será `i/2`, salvo en el caso de la raíz.
- Las hojas tendran índices en $[n, 2n)$.
- La idea será que cada nodo interno del árbol (guardado en un elemento del arreglo) almacene el mínimo entre sus dos hijos.
- Las hojas contendrán en todo momento los elementos del arreglo v.

Estructura : Inicializacion

- Llenamos $A_n, \cdots, A_{2n-1}$ con los elementos del arreglo v.
- Usando programación dinámica hacemos simplemente:

$$A_i = \min(A_{2i}, A_{2i+1})$$

- Notar que se debe recorrer i en forma descendente para no utilizar valores aún no calculados.
- El proceso de inicialización toma tiempo $O(n)$, y la estructura utiliza $O(n)$ memoria.

Estructura : Modificaciones

- Para modificar el arreglo, utilizaremos nuevamente la fórmula:

$$A_i = \min(A_{2i}, A_{2i+1})$$

- Notemos que si cambiamos el valor de v_i por x, debemos modificar A_{n+i} haciéndolo valer x, y recalcular los nodos internos del árbol...
- Pero sólo se ven afectados los ancestros de A_{n+i}.
- Luego es posible modificar un valor de v en tiempo $O(\lg n)$, recalculando sucesivamente los padres desde A_{n+i} hasta llegar a la raíz.

Estructura : Queries

Consideremos el siguiente algoritmo recursivo:

- $$f(k,l,r,i,j) = \begin{cases} A_k & \text{si } i \leq l < r \leq j; \\ +\infty & \text{si } r \leq i \text{ o } l \geq j \\ \min(f(2k, l, \frac{l+r}{2}, i, j), & \\ , f(2k+1, \frac{l+r}{2}, r, i, j)) & \textit{sino.} \end{cases}$$
- La respuesta vendrá dada por $RMQ(i,j) = f(1,0,n,i,j)$
- La complejidad temporal de un query es $O(\lg n)$

Ejemplo de código (Segment Tree)

```
1  struct STree { // [cerrado-abierto), defininir "oper" y "NEUT"
2    vector<ll> st;int n;
3    STree(int n): st(4*n+5,NEUT), n(n) {}
4    void upd(int k, int s, int e, int p, ll v){
5      if(s+1==e){st[k]=v;return;}
6      int m=(s+e)/2;
7      if(p<m)upd(2*k,s,m,p,v);
8      else upd(2*k+1,m,e,p,v);
9      st[k]=oper(st[2*k],st[2*k+1]);
10   }
11   ll query(int k, int s, int e, int a, int b){
12     if(s>=b||e<=a)return NEUT;
13     if(s>=a&&e<=b)return st[k];
14     int m=(s+e)/2;
15     return oper(query(2*k,s,m,a,b),query(2*k+1,m,e,a,b));
16   }
17   void upd(int p, ll v){upd(1,0,n,p,v);}
18   ll query(int a, int b){return query(1,0,n,a,b);}
19 }; // usage: STree st(n);st.upd(i,v);st.query(s,e);
```

Contenidos

Binary Indexed Tree

- También llamados “Fenwick Tree”
- Introducidos en la literatura en:
  Peter M. Fenwick (1994). “A new data structure for cumulative frequency tables”. Software: Practice and Experience
- Se utilizan para implementar eficientemente algoritmos de compresión adaptativa mediante Arithmetic Encoding (motivación original de Fenwick)
- $\frac{\text{Fenwick}}{\text{Tabla Aditiva}} = \frac{\text{Segment Tree}}{\text{Sparse Table}}$
- Un Segment Tree es estrictamente más poderoso como estructura de datos. El Fenwick Tree tiene la misma complejidad, pero suele ser más fácil de programar y tener mejores constantes.

Ejemplo de código (Fenwick Tree)

```
1  int ft[MAXN+1];
2  void upd(int i0, int v){ // add v to i0th element (0-based)
3    for(int i=i0+1;i<=MAXN;i+=i&-i)ft[i]+=v;
4  }
5  int get(int i0){ // get sum of range [0,i0)
6    int r=0;
7    for(int i=i0;i;i-=i&-i)r+=ft[i];
8    return r;
9  }
10 int get_sum(int i0, int i1){ // get sum of range [i0,i1) (0-based)
11   return get(i1)-get(i0);
12 }
```

Las estructuras de datos (múltiples dimensiones)

- Todas estas estructuras generalizan adecuadamente para resolver los correspondientes problemas en múltiples dimensiones.
- Por ejemplo para el caso 2D, podemos usar un Segment Tree donde cada hoja sea un nuevo Segment Tree.
- Incluso se podráan combinar de maneras diferentes, teniendo por ejemplo una tabla aditiva en donde cada celda guarda una Sparse Table.