

# Estructuras de datos

Training Camp Argentina 2013

Fidel I. Schaposnik (UNLP) - fidel.s@gmail.com  
17 de julio de 2013

- Introducción
- set de C++
- Mínimo en intervalos (RMQ)
  - caso estático: *sparse tables*
  - caso dinámico: *segment tree*
- Suma en intervalos
  - caso estático: sumas parciales uni- y bi-dimensionales
  - caso dinámico: *binary indexed tree*
- Operaciones más generales en intervalos
  - caso estático
  - caso dinámico: *square root trick*
- Problema adicional: ancestro común más bajo (LCA)

## ¿Qué es una estructura de datos?

- Una forma de almacenar datos y operar con ellos de modo de poder calcular eficientemente diversas magnitudes.
- Puede pensarse como una “caja negra” donde están los datos (posiblemente sujetos a modificaciones), que nos responderá cierto tipo de preguntas de modo eficiente.

## ¿Cómo se usa una estructura de datos en una competencia de programación?

Las etapas o pasos intermedios del algoritmo muchas veces pueden verse como subproblemas menores, que debemos resolver para hallar la respuesta final. Si bien los problemas completos rara vez se repiten, los subproblemas muchas veces son “clásicos”, y conocemos para ellos una estructura de datos adecuada.

El set de C++ representa un conjunto de elementos, que podemos pensar como ordenados de “menor” a “mayor”. Nos permite realizar distintas operaciones en  $\mathcal{O}(\log N)$ :

- insertar y borrar elementos (`insert` y `erase`, resp.);
- fijarnos si un elemento está presente (`find`);
- encontrar el primer elemento “mayor” o “mayor o igual” que otro (`upper_bound` y `lower_bound`, respectivamente).

Algunas observaciones:

- en un conjunto no puede haber repeticiones;
- no controlamos el funcionamiento interno de la estructura, solo la función de comparación;
- no podemos modificar elementos (pero sí eliminar una versión vieja de un elemento y reemplazarla por una nueva).

# Ejemplo del uso de set de C++

```
1 #include <iostream>
2 #include <set>
3 using namespace std;
4
5 int main() {
6     set<int> c;
7
8     for (int i=-15; i<17; i+=3) c.insert(i);
9
10    cout << c.size() << endl;
11    for (set<int>::iterator it=c.begin(); it!=c.end(); it++)
12        cout << *it << ' ';
13
14    cout << endl << (c.find(-9)!=c.end()) << endl;
15    c.erase(-9);
16    cout << (c.find(-9)!=c.end()) << endl;
17
18    cout<<*c.lower_bound(6)<<' ' <<*c.upper_bound(6)<<endl;
19    cout<<*c.lower_bound(7)<<' ' <<*c.upper_bound(7)<<endl;
20
21    return 0;
22 }
```

*Ejemplo del uso de set de C++*

# Ejemplo del uso de set de C++ (cont.)

Podemos utilizar nuestros propios elementos definiendo un orden

```
1 struct pt {
2     int x, y;
3     pt(int xx=0, int yy=0) {
4         x=xx; y=yy;
5     };
6 };
7
8 bool operator<(const pt &p1, const pt &p2) {
9     double a1=atan2(p1.y, p1.x), a2=atan2(p2.y, p2.x);
10    if (a1 != a2) return a1 < a2;
11    else return p1.x*p1.x+p1.y*p1.y < p2.x*p2.x+p2.y*p2.y;
12 }
13
14 [...]
15
16 set<pt> p;
17
18 p.insert(pt(1,1)); p.insert(pt(2,2));
19 p.insert(pt(2,1)); p.insert(pt(3,5));
```

*Uso del set de C++ con una estructura propia*

# Aplicación: Dijkstra

```
1 set< pair<int , int> > s;  
2  
3 void dijkstra(int start , int dest , int N) {  
4     for (int i=0; i<N; i++) n[i].d = INF;  
5     n[start].d = 0; s.insert(make_pair(0, start));  
6     while (!s.empty()) {  
7         int cur = s.begin()->second;  
8         if (cur == dest) break;  
9         for (int i=0; i<(int)n[cur].c.size(); i++) {  
10             int next = n[cur].c[i].first;  
11             if (n[next].d==INF || n[next].d>n[cur].d+n[cur].c[i].second){  
12                 if (n[next].d != INF)  
13                     s.erase(make_pair(n[next].d, next));  
14                 n[next].d = n[cur].d + n[cur].c[i].second;  
15                 s.insert(make_pair(n[next].d, next));  
16             }  
17         }  
18         s.erase(make_pair(n[cur].d, cur));  
19     }  
20 }
```

*Algoritmo de Dijkstra usando un set -  $\mathcal{O}(N \log N + E)$*

- *Template Library Management* (TJU - 3972): Algoritmo goloso acelerado usando dos set elegantemente.



# Mínimo en intervalos (RMQ)

Dado un arreglo de  $N$  números  $m_0, \dots, m_{N-1}$ , queremos responder preguntas de la forma

**¿Cuál es el menor número en el intervalo  $m_i, \dots, m_{j-1}$ ?**  
 $\equiv \text{RMQ } [i, j)$

- El algoritmo ingenuo recorre todos los elementos en  $[i, j)$ , y requiere  $\mathcal{O}(N)$  en el peor caso;
- Si no se van a modificar los valores del arreglo, podemos preprocesar en  $\mathcal{O}(N \log N)$  y responder preguntas en  $\mathcal{O}(1)$ ;
- Si los elementos del arreglo pueden cambiar, podemos inicializar en  $\mathcal{O}(N)$  y responder preguntas o modificar valores en  $\mathcal{O}(\log N)$

# RMQ - Sparse tables

Si los elementos del arreglo no pueden cambiar de valor, podemos **preprocesar** los datos para acelerar el cálculo de las respuestas:

- Hay  $\mathcal{O}(N^2)$  preguntas posibles, y podemos precalcular todas sus respuestas en  $\mathcal{O}(N^2)$  usando programación dinámica;
- No hace falta guardar todas las respuestas: basta con precalcular una cantidad suficiente como para poder responder cualquier pregunta.

Definimos

$$\text{st}_k(i) = \min_{i \leq j < i+2^k} \{m_j\} \equiv \text{RMQ}[i, i+2^k) \quad \text{para} \quad k = 0, 1, \dots$$

Observamos que podemos precalcular los  $\mathcal{O}(N \log N)$  valores de  $\text{st}$  usando programación dinámica

$$\text{st}_0(i) = m_i$$

$$\text{st}_k(i) = \min \left\{ \text{st}_{k-1}(i), \text{st}_{k-1}(i + 2^{k-1}) \right\}$$

# RMQ - Sparse tables (cont.)

Para calcular  $\text{RMQ}[i, j)$ , observamos que si  $2^k \leq j - i < 2^{k+1}$ ,

$$\text{RMQ}[i, j) = \min \left\{ \text{st}_k(i), \text{st}_k(j - 2^k) \right\}$$

```
1 void st_init(int *m, int N, int **st) {
2     for (int i=0; i<N; i++) st[0][i] = m[i];
3     for (int k=1; (1<<k)<=N; k++) {
4         for (int i=0; i+(1<<k)<=N; i++) {
5             if (st[k-1][i] < st[k-1][i+(1<<(k-1))])
6                 st[k][i] = st[k-1][i];
7             else st[k][i] = st[k-1][i+(1<<(k-1))];
8         }
9     }
10 }
11
12 int st_query(int **st, int s, int e) {
13     int k = 31 - __builtin_clz(e-s);
14     if (st[k][s] < st[k][e-(1<<k)]) return st[k][s];
15     else return st[k][e-(1<<k)];
16 }
```

*RMQ estático con sparse tables -  $\mathcal{O}(N \log N)$  init. y  $\mathcal{O}(1)$  query*

Si los elementos del arreglo pueden cambiar, el preproceso o **inicialización** debe ser complementado con el **mantenimiento** de la estructura.

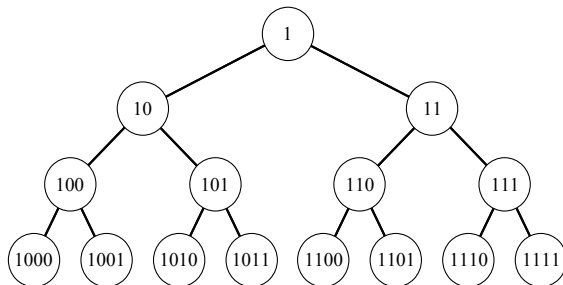
Vamos a usar un árbol binario del siguiente modo:

- Cada nodo representa un intervalo, en particular la raíz representa al  $[0, N)$
- El hijo izquierdo de un nodo representa la primera mitad del segmento de su padre, el derecho la otra
- Las hojas del árbol representan segmentos de longitud uno, y almacenan el correspondiente valor del arreglo
- En cada nodo interno se almacena el mínimo de los dos valores almacenados en sus dos hijos

# RMQ - Segment tree (cont.)

Una forma elegante de definir un árbol binario usando un arreglo

- El “nombre” de un nodo es su posición en el arreglo
- La raíz es el nodo 1
- El hijo izquierdo de  $i$  es el nodo  $2i$ , el derecho es el  $2i + 1$
- El padre del nodo  $i$  es el  $i/2$
- El valor almacenado en el nodo es el valor en la posición correspondiente del arreglo



## RMQ - Segment tree (cont.)

- Para inicializar, debemos llenar los valores de los nodos desde las hojas hasta la raíz:  $\mathcal{O}(N)$
- Para modificar un valor, debemos cambiar el valor de la hoja correspondiente y luego actualizar los valores de todos los nodos internos en el camino de la hoja hasta la raíz:  $\mathcal{O}(\log N)$
- Para realizar una consulta, debemos tomar el mínimo entre los valores almacenados en los nodos cuyos segmentos corresponden exactamente al intervalo deseado:  $\mathcal{O}(\log N)$
- La estructura tipo árbol sugiere una implementación recursiva de todas estas funciones

# RMQ - Segment tree (código)

```
1 #define LEFT(n) ( 2*(n) )
2 #define RIGHT(n) ( 2*(n)+1 )
3 #define NEUT 2147483647
4
5 int oper(int a, int b) {
6     return min(a, b);
7 }
8
9 void rmq_init(int n, int s, int e, int *rmq, int *m) {
10     if (s+1 == e) rmq[n] = m[s];
11     else {
12         rmq_init(LEFT(n), s, (s+e)/2, rmq, m);
13         rmq_init(RIGHT(n), (s+e)/2, e, rmq, m);
14         rmq[n] = oper(rmq[LEFT(n)], rmq[RIGHT(n)]);
15     }
16 }
```

*RMQ dinámico con segment tree - Inicialización*

# RMQ - Segment tree (código cont.)

```
1 void rmq_update(int n, int s, int e, int*rmq, int*m, int p, int
  v) {
2   if (s+1 == e) rmq[n] = m[s] = v;
3   else {
4     if (p < (s+e)/2)
5       rmq_update(LEFT(n), s, (s+e)/2, rmq, m, p, v);
6     else rmq_update(RIGHT(n), (s+e)/2, e, rmq, m, p, v);
7     rmq[n] = oper(rmq[LEFT(n)], rmq[RIGHT(n)]);
8   }
9 }
10
11 int rmq_query(int n, int s, int e, int *rmq, int a, int b) {
12   if (a >= e || b <= s) return NEUT;
13   else if (s >= a && e <= b) return rmq[n];
14   else {
15     int l = rmq_query(LEFT(n), s, (s+e)/2, rmq, a, b);
16     int r = rmq_query(RIGHT(n), (s+e)/2, e, rmq, a, b);
17     return oper(l, r);
18   }
19 }
```

*RMQ dinámico con segment tree - Mantenimiento y consulta*



# RMQ - Segment tree (observaciones)

- Todas las funciones se llaman con  $n = 1$ ,  $s = 0$  y  $e = N$
- En algunos casos conviene almacenar la *posición* del menor elemento, en lugar de su valor
- `oper` puede ser otras operaciones además de  $<$ : podemos usar  $>$ , pero también operaciones algebraicas como  $+$  y  $*$ , operaciones binarias como  $\vee$ ,  $\wedge$  y  $|$ , etc.
- NEUT debe ser el elemento neutro de la operación `oper`

# Suma en intervalos

Dado un arreglo de  $N$  números  $m_0, \dots, m_{N-1}$ , queremos responder preguntas de la forma

**¿Cuál es el valor de  $S[i, j) = m_i + \dots + m_{j-1}$ ?**

- El algoritmo ingenuo recorre todos los elementos en  $[i, j)$ , y requiere  $\mathcal{O}(N)$  en el peor caso;
- Si no se van a modificar los valores del arreglo, podemos preprocesar en  $\mathcal{O}(N)$  y responder preguntas en  $\mathcal{O}(1)$ ;
- Si los elementos del arreglo pueden cambiar, podemos inicializar en  $\mathcal{O}(N \log N)$  y responder preguntas o modificar valores en  $\mathcal{O}(\log N)$

# Suma en intervalos: sumas parciales

- Si los elementos del arreglo no pueden modificarse, podemos precalcular todas las respuestas en  $\mathcal{O}(N^2)$  y responder en  $\mathcal{O}(1)$ .
- Observamos que basta poder responder cuando  $i = 0$ , porque

$$S[i, j) = S[0, j) - S[0, i)$$

Luego podemos precalcular sólo los valores de  $S[0, i)$  en  $\mathcal{O}(N)$ , y responder en  $\mathcal{O}(1)$

```
1 void init(int *m, int *sm, int N) {  
2     sm[0] = 0;  
3     for (int i=1; i<=N; i++) sm[i] = sm[i-1]+m[i-1];  
4 }  
5  
6 int query(int *sm, int a, int b) {  
7     return sm[b]-sm[a];  
8 }
```

*Suma en intervalos usando sumas parciales*

# Suma en intervalos: sumas parciales bidimensionales

El problema anterior puede generalizarse para el caso de arreglos bidimensionales: si queremos calcular

$$S_2(a, b, c, d) = \sum_{i=a}^{b-1} \sum_{j=c}^{d-1} m_{ij}$$

para  $a < c$  y  $b < d$ , basta conocer  $S_2(0, 0, i, j)$ , dado que

$$S_2(a, b, c, d) = S_2(0, 0, c, d) - S_2(0, 0, a, d) - S_2(0, 0, c, b) + S_2(0, 0, a, b)$$

Luego el preprocesamiento será en  $\mathcal{O}(N^2)$  y las consultas en  $\mathcal{O}(1)$ .

El principio de inclusión exclusión nos permite generalizar esto a arreglos  $n$ -dimensionales, obteniendo consultas en  $\mathcal{O}(2^n)$  realizando un preprocesamiento en  $\mathcal{O}(N^n)$

# Suma en intervalos: binary indexed tree

Si los elementos del arreglo pueden modificarse, ya conocemos una estructura eficiente: usamos un segment tree con  $\text{oper} \mapsto +$  y  $\text{NEUT} \mapsto 0$ .

Una estructura más eficiente aprovecha la observación  $S[i, j) = S[0, j) - S[0, i)$  modificando los intervalos almacenados en cada posición del arreglo

- Definimos  $\text{bit}(i) = \sum_{j=i-k+1}^i m_j$  con  $k$  la mayor potencia de 2 que divide a  $i$  (o  $k = 1$  si  $i = 0$ ).
- Para calcular  $S[0, i + 1)$  sumamos los intervalos que lo componen sacando progresivamente los 1's de la expresión binaria de  $i$ .
- Para modificar una posición (sumar una cantidad en una posición dada) agregamos la cantidad deseada a todos los intervalos que contienen esa posición.

# Suma en intervalos: binary indexed tree (código)

P. Fenwick, "A New Data Structure for Cumulative Frequency Tables",  
Software - Practice and Experience, **24**(3), pp. 327-336 (1994)

```
1 int get_cf(int idx, int *bit) {
2     int cf = bit[0];
3     while (idx > 0) {
4         cf += bit[idx];
5         idx &= idx - 1;
6     }
7     return cf;
8 }
9
10 void upd_f(int idx, int f, int *bit) {
11     if (idx == 0) bit[idx] += f;
12     else while (idx < MAXN) {
13         bit[idx] += f;
14         idx += idx & (-idx);
15     }
16 }
```

*Suma en intervalos usando binary indexed tree*

# Suma en intervalos: binary indexed tree (observaciones)

- Las posiciones del arreglo acumulan los cambios, luego para modificar una posición llamamos a `upd_f` con

$$v = m_i^{(\text{nuevo})} - m_i^{(\text{viejo})}$$

- Las dos operaciones son  $\mathcal{O}(\log N)$
- La inicialización es  $\mathcal{O}(N \log N)$  porque debemos modificar todas las posiciones una por una
- También puede modificarse para ser utilizado con otras operaciones
- Se puede extender fácilmente esta estructura para operar sobre arreglos bidimensionales

# Operaciones más generales: caso estático

Dada una operación asociativa  $\otimes$  y un arreglo de  $N$  elementos  $m_0, \dots, m_{N-1}$  que no se pueden modificar, si queremos responder preguntas de la forma

**¿Cuál es el valor de  $O[i, j) = m_i \otimes \dots \otimes m_{j-1}$ ?**

Definimos

$$O_k(i) = \bigotimes_{j=i}^{i+2^k-1} m_j \equiv O[i, i+2^k)$$

y para calcular  $O[i, j)$  aplicamos la operación adelantando  $i$  a medida que encontramos 1's en la cantidad  $j - i$



# Operaciones más generales: caso estático (código)

```
1 void init(int *m, int **sm, int N) {
2     for (int i=0; i<N; i++) sm[0][i] = m[i];
3     for (int k=1; (1<<k)<=N; k++)
4         for (int i=0; i+(1<<k)<=N; i++)
5             sm[k][i]=oper(sm[k-1][i], sm[k-1][i+(1<<(k-1))]);
6 }
7
8 int query(int **sm, int a, int b) {
9     int RES = 0, l = b-a;
10    for (int i=0; (1<<i)<=l; i++) if (l&(1<<i)) {
11        RES = oper(RES, sm[i][a]);
12        a += (1<<i);
13    }
14    return RES;
15 }
```

*Operaciones más generales: caso estático -  $\mathcal{O}(N \log N)$  inicialización y  $\mathcal{O}(\log N)$  consulta*

# Operaciones más generales: caso dinámico

Si los elementos del arreglo pueden modificarse, podemos usar el siguiente truco:

- Partimos el intervalo  $[0, N)$  en  $\sqrt{N}$  segmentos  $s_i$  con  $i = 0, \dots, \sqrt{N}$
- Para mayor comodidad, definimos  $a_i = i\sqrt{N}$  y  $b_i = (i + 1)\sqrt{N}$ ; luego el  $i$ -ésimo segmento corresponde a  $[a_i, b_i)$ , luego
- Para cada segmento, guardamos el valor de

$$O_i = \bigotimes_{j=a_i}^{b_i-1} m_j \equiv O[a_i, b_i)$$

- Al realizar una modificación en la posición  $j$ , debemos recalcular el  $O_i$  tal que  $j \in [a_i, b_i) \implies \mathcal{O}(\sqrt{N})$
- Al realizar una consulta, debemos realizar la operación  $\otimes$  sobre los segmentos completos (a lo sumo  $\sqrt{N}$ ) y los elementos sueltos a ambos lados (a lo sumo  $2\sqrt{N}$ )  $\implies \mathcal{O}(\sqrt{N})$

## Problema adicional: ancestro común más bajo

El problema consiste en hallar el ancestro común más bajo entre pares de nodos de un árbol enraizado

# Problema adicional: ancestro común más bajo

El problema consiste en hallar el ancestro común más bajo entre pares de nodos de un árbol enraizado

- Puede reducirse a la aplicación de RMQ sobre el arreglo generado con el *pre- y post-visit DFS transversal* del árbol:  
**¿Cómo?**

# Problema adicional: ancestro común más bajo

El problema consiste en hallar el ancestro común más bajo entre pares de nodos de un árbol enraizado

- Puede reducirse a la aplicación de RMQ sobre el arreglo generado con el *pre- y post-visit DFS transversal* del árbol:  
**¿Cómo?**
- Puede pensarse como un caso particular de la estructura analizada para operaciones generales en arreglos estáticos:  
**¿Cómo?**

# Problema adicional: ancestro común más bajo

El problema consiste en hallar el ancestro común más bajo entre pares de nodos de un árbol enraizado

- Puede reducirse a la aplicación de RMQ sobre el arreglo generado con el *pre- y post-visit DFS transversal* del árbol:  
**¿Cómo?**
- Puede pensarse como un caso particular de la estructura analizada para operaciones generales en arreglos estáticos:  
**¿Cómo?**
- Puede utilizarse para calcular eficientemente distancias entre los nodos de un árbol: **¿Cómo?**