Estructuras de datos avanzadas

Training Camp Argentina 2014

Lucas Tavolaro Ortiz (UBA) - tavo92@gmail.com

4 de agosto de 2014

Contenidos

- Introducción
- STL 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*
- Lowest Common Ancestor
- Ejercicio Adicional: Segment tree con lazy propagation

Introducción

¿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.

STL de C++

La STL es la Standard Template Library de C++ que contiene varias estructuras de datos que ya vienen hechas y listas para usar. Es importante estar familiarizadas con ellas ya que suelen usarse mucho en la competencia. A modo de repaso:

- vector: Inserción atras en $\mathcal{O}(1)$ amortizado, insercion en cualquier lado $\mathcal{O}(N)$, acceso a cualquier elemento en $\mathcal{O}(1)$, busqueda de un elemento en $\mathcal{O}(N)$
- set: Inserción, eliminación y busqueda de un elemento en $\mathcal{O}(\log N)$
- map: Relaciona una clave con un valor. Los tiempos son iguales que el set
- list: Inserción y eliminación en $\mathcal{O}(1)$. Acceder y buscar un elemento $\mathcal{O}(N)$
- queue: Inserción atras y eliminar el primer elemento en $\mathcal{O}(1)$
- unorder_set, unorded_map (C++ 11): Utilizando una buena función de hashing pueden lograr $\mathcal{O}(1)$ en todas las operaciones.

Mínimo en intervalos (RMQ)

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

¿Cuál es el menor número en el intervalo $m_i, \ldots, m_{j-1}$?

$$\equiv \text{RMQ}\ [i, j\,)$$

- 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

$$\mathrm{st}_k(i) = \min_{i \le j < i+2^k} \{m_j\} \equiv \mathrm{RMQ}[i, i+2^k) \qquad \text{para} \qquad k = 0, 1, \ldots$$

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

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

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

RMQ - Sparse tables (cont.)

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

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

```
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              st[k][i] = min(st[k-1][i],
                   st[k-1][i+(1<<(k-1))]);
6      }
7    }
8  }
9
10 int st_query(int **st, int s, int e) {
11     int k = 31 - __builtin_clz(e-s);
12     return min(st[k][s], st[k][e-(1<<k)]);
13 }
```

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

RMQ - Segment tree

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

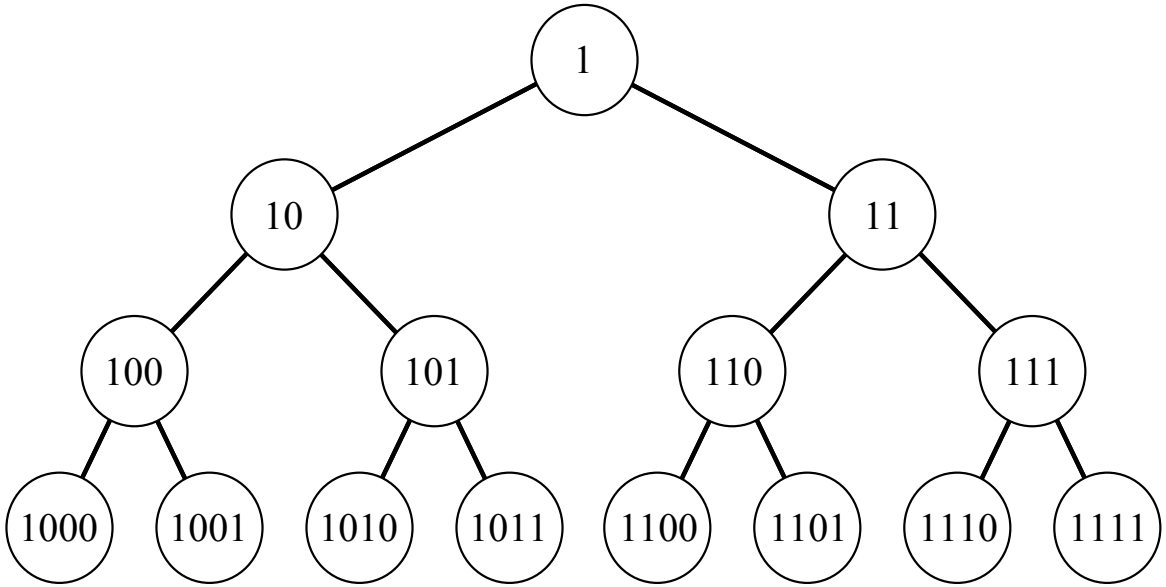
Vamos a usar un árbol binario del siguiente modo:

- 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 $m_0, \ldots, m_{N-1}$, queremos responder preguntas de la forma

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

- 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 `oper` $\mapsto +$ y `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 $\mathrm{bit}(i) = \sum_{j=i-k+1}^{i} m_j$ 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

Lowest Common Ancestor

Dado un arbol y dos nodos, queremos saber cual es el ancestro común mas bajo entre ellos dos.Nos sirve de algo lo visto hasta ahora?

Lowest Common Ancestor

Dado un arbol y dos nodos, queremos saber cual es el ancestro común mas bajo entre ellos dos.Nos sirve de algo lo visto hasta ahora?

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

Lowest Common Ancestor

Dado un arbol y dos nodos, queremos saber cual es el ancestro común mas bajo entre ellos dos.Nos sirve de algo lo visto hasta ahora?

- 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?**

Lowest Common Ancestor

Dado un arbol y dos nodos, queremos saber cual es el ancestro común mas bajo entre ellos dos.Nos sirve de algo lo visto hasta ahora?

- 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?**

Lowest Common Ancestor - Idea

- Utilizando un recorrido de DFS, podemos computar un arreglo v de 2*n-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.
- Notaremos el indice de la primera aparación de un nodo i en v como L_i.

Se les ocurre como usar esto para calcular el LCA?

Lowest Common Ancestor - Observación clave

$\text{LCA(i, j)} = \text{RMQ}(\min(L_i, L_j), \max(L_i, L_j))$

- El min y max se utilizan para asegurar que $i < j$.
- Para comparar se utilizan las profundidas de cada nodo definida anteriormente.

Ejercicio Adicional: Segment tree con lazy propagation

Volvamos al problema del RMQ, qué sucede si las actualizaciones no son solo de un solo nodo sino que puede ser un intervalo de valores [i, j) ?

- Actualizar nos costaria $\mathcal{O}(N \log N)$ si usamos lo que definimos antes.
- Habra alguna manera de hacerlo mejor?

Ejercicio Adicional: Segment tree con lazy propagation

Volvamos al problema del RMQ, qué sucede si las actualizaciones no son solo de un solo nodo sino que puede ser un intervalo de valores [i, j) ?

- Actualizar nos costaria $\mathcal{O}(N \log N)$ si usamos lo que definimos antes.
- Habra alguna manera de hacerlo mejor?
- Es necesario actualizar **todos los nodos del arbol** por cada intervalo? Podremos aprovechar el hecho de que cada nodo del arbol representa un intervalo?

Ejercicio Adicional: Segment tree con lazy propagation

Lazy propagation: Solo propagamos lo estrictamente necesario, el resto lo dejamos anotado en el nodo como valor que debe ser propagado.

- Ahora cada actualización nos cuesta $\mathcal{O}(\log N)$: Si un nodo esta en el intervalo entonces frenamos ahi y marcamos que "mas adelante" se debe propagar el resto.
- Tener cuidado al recorrer el arbol ya que se debe ir propagando mientras se visitan los nodos.

Se animan a implementarlo? Qué habria que cambiar del código mostrado anteriormente?

El Final :(

Se entendio todo? Alguna pregunta? Te parecio que faltaba alguna estructura de datos que no se dio en esta charla? Fueron muy voladas ciertas cosas?