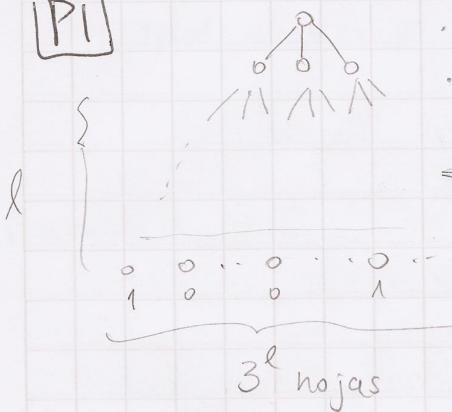


# Aux # 5

2015年10月19日 (A)

PI



- Árbol binario
- valor del padre es el mismo que la mayoría de los hijos  
⇒ Cuál es el valor del padre?
  - El algoritmo pide hojas en orden ALEATORIO  
⇒ debo trabajar en función del orden en que me las preguntan.

Adv: La idea es que si el ALG no pregunta todos,  $A_0$  y  $A_1$  son respuestas

- Mantenemos 2 Árboles  $A_0$  y  $A_1$   
    †'s pero ambas
- Inicializo todas las hojas de  $A_0$  con 0's  
    " " " "      $A_1$  con 1's  
    Son VÁLIDAS

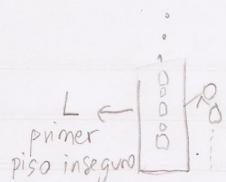
• Ante cada pregunta

- Si es el primer hijo de su trío en ser preguntado, respondo 0.  
    y actualizo  $A_1$
- Si es el segundo hijo de su trío en ser preguntado, respondo 1  
    y actualizo  $A_0$

- Si es el tercer hijo, subir un nivel y repetir

↳ nunca  
llega a la raíz si hay menos de 3<sup>l</sup>  
preguntas

\* Luego si el algoritmo responde 0, muestro  $A_1$  y viceversa.  
Luego un algoritmo que haga < 3<sup>l</sup> preguntas, no puede ser correcto.



**P2** Buscar primer piso inseguro para tirar un huevo por la ventana

a) Si tengo 1 solo huevo

→ Se intenta desde el primer piso:

Se deja caer.

→ Si se rompe, ese piso era L

→ Si no, subo 1 piso

$$\left. \begin{array}{l} \\ \\ \end{array} \right\} \Theta(L)$$

b) Definamos  $f(i)$  el piso desde el que el algoritmo bota el huevo en el paso  $i$

Adversario:

Si en el paso  $i$ ,  $f(i) > i$  → se rompió

Si no, no.

$$\left. \begin{array}{l} \\ \\ \end{array} \right\} \Omega(L)$$

Si  $f(i) > i$ , el algoritmo no puede distinguir si  $L = f(i)$  o  $L = f(i) - 1$

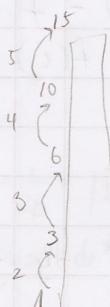
⇒ El algoritmo no puede ser correcto,

c) 2 huevos  $\Rightarrow \Theta(\sqrt{L})$

• para el primer huevo:

for  $i=1\dots$  hasta que se rompe:

bota el huevo desde el piso  $\frac{i(i+1)}{2}$



• Para el 2º huevo:

Sea  $k$  la iteración en que se rompió el primer huevo.

for  $i = \frac{k(k-1)}{2} + 1 \dots \frac{k(k+1)}{2} - 1$ :

bota el huevo desde  $i$

• Si el 2º huevo se rompe en un piso, ése será  $L$ .

• Si no,  $L = \frac{k(k+1)}{2}$

$$\boxed{\begin{array}{|c|} \hline \text{ } \\ \hline \end{array}} \quad \binom{t}{2} + t = \binom{t+1}{2}$$

nivel anterior  
 $\frac{(k-1)(k-1+1)}{2}$

Por def. del algoritmo,  $L > \frac{k(k-1)}{2} \rightarrow k = \Theta(\sqrt{L})$

el 1º huevo se bota  $k$  veces  $\Rightarrow \Theta(\sqrt{L})$

$$\text{el 2º huevo: } \frac{k(k+1)-1}{2} = \frac{k(k-1)+1}{2}$$

$$= \frac{k^2+k}{2} - \frac{k^2}{2} + \frac{k}{2} - 2$$

$$= k-2$$

$$\Rightarrow \underline{\Theta(\sqrt{L})}$$

$$\binom{t}{2} = \frac{t!}{2!(t-2)!} + t$$

$$= \frac{t! + 2t(t-2)!}{2(t-2)!}$$

$$= \frac{t! \cdot t-1 + 2t!}{2(t-1)!} = \frac{t!(t+1)}{2(t-1)!}$$

d) El adversario elige un valor "grande" de  $L$

De nuevo,  $f(i)$  son los pisos testeados por el algoritmo en el piso  $i$   
(definimos  $f(0) := 0$ )

Dividimos el análisis en 2 casos:

$$\bullet f(i) - f(i-1) \leq i \forall i$$

$$\Rightarrow f(i) \leq \underbrace{i}_{2} \underbrace{(i+1)}_{2} + i \text{ (Veo si } f(i) \geq L \text{ )}$$

$$\sum_{j=0}^i j$$

$$i = \lceil 2(\sqrt{L}) \rceil$$

$\Rightarrow$  el algoritmo hace  $\Omega(\sqrt{L})$  tests

$$\bullet \exists i^* \text{ tq } \underbrace{f(i^*) - f(i^*-1)}_t > i^*$$

El adversario declara que el huevo se rompió en  $f(i^*)$   
y define  $L = f(i^*)$

De la parte 2, sabemos que entre  $f(i^*-1)$  y  $f(i^*)$  hay  
que hacer  $t-2$  tests

$$L = f(i^*) = f(i^*-1) + t, \text{ pero } f(i^*-1) \geq \frac{i^*(i^*-1)}{2}$$

$$= \frac{i^*(i^*-1)}{2} + t$$

$$< \frac{t(t+1)}{2} + t = \binom{t}{2} + t = \binom{t+1}{2} \Rightarrow t = \Omega(\sqrt{L})$$

$\Rightarrow$  el 2º huevo se bota  $\Omega(\sqrt{L})$  veces.

## Union Find

[P3]

Stacks en memoria secundaria.



LIFO

Naive:

↳ buffer de tamaño  $B$  en memoria.

• Pushes y pops van al buffer

→ si se llena, envío el bloque a disco

→ si se vacía, pido un bloque de disco

PROBLEMA: una secuencia de pushes y pops con buffer lleno resultan en muchos I/O

Idea:  $2B$  en vez de  $B$  para el buffer.

• Si se llena, escribo solo  $B$

• Si se vacía, pido solo  $B$ .

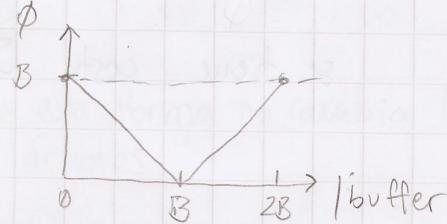
ANÁLISIS: fn. potencial

$\Phi$ : • siempre positiva ( $\Phi_k \geq \Phi_0 \forall k$ )

$$\Phi: \begin{cases} |buffer| - B & \text{si } |buffer| \geq B \\ B - |buffer| & \text{si } |buffer| < B \end{cases}$$

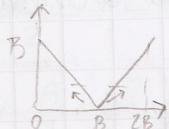
Estado ideal de  $B$

$$\Rightarrow \dot{\Phi} = |buffer| - B$$



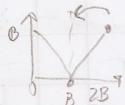
## Costos

Push y Pop sin tocar disco  
 $\Rightarrow \pm 1$  (operaciones en memo no cuentan, pero sí cuenta el cambio en la fn potencial,  $\Delta\phi$ )



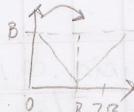
Push con overflow

$$\text{Costo: } B + \Delta\phi = B + -B = \emptyset$$



Pop con underflow

$$\text{Costo: } B + \Delta\phi = B + -B = \emptyset$$



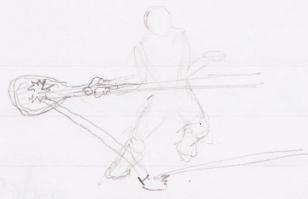
$\Rightarrow$  Costo amortizado constante. Cuesta  $B$  mover un bloque

$\rightarrow$  debe ser

$$\phi = |\text{buffer}| - B$$

se tiene costo  $\mathcal{O}\left(\frac{1}{B}\right)$  cor op.

# Union Find



2015年10月20日 (X)

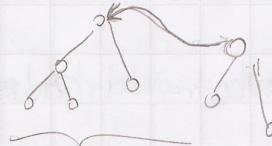
Ops: Make-set( $x$ )  $\rightarrow$  crea conjunto  $\{x\}$

Union( $x, y$ )  $\rightarrow$  une los conjuntos de  $x$  e  $y$

Find( $x$ )  $\rightarrow$  retorna "label" del conjunto de  $x$ .

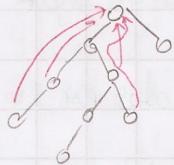
- Find retorna la raíz

- Union une 2 árboles



$\rightarrow$  Link-by-rank: uno el de "menor altura"  
( $\text{rank}(x) = h(x)$ ) como hijo del de mayor  
se actualiza

## Path Compression:



Importante: al hacer compresión de caminos NO vamos a actualizar  $\text{rank}(x)$  en el momento de la compresión  $\Rightarrow \text{rank}(x) \geq h(x)$ . (No como en el caso anterior)

## Propiedades:

0: Agregar p.c. (path compression) de esta forma no cambia los ranks, raíces o elementos de los árboles

1: Si  $x$  no es una raíz,

$$\text{rank}(x) < \text{rank}(\text{padre}(x))$$

2: Si  $x$  no es una raíz,  $\text{rank}(x)$  no cambia.

3: Si  $\text{padre}(x)$  cambia,  $\text{rank}(\text{padre}(x))$  crece

4: Toda raíz de  $\text{rank}(k)$  tiene  $\geq 2^k$  nodos en su árbol.

5: El rank más alto es  $\leq \lfloor \lg n \rfloor$

6: Si  $r \geq 0$ , hay  $\leq \frac{n}{2^r}$  nodos con rank  $r$ .

PROBLEMAS

①, 2, 4 y 5 son triviales

Le asigno un padre con rank mayor o igual  
no es que le cambie el rank al nodo

1: P.C. solo incrementa el rank de los padres de los nodos, pues los enlaza a sus antecesores.

3: P.C. solo asigna un ancestro del parente original  $\Rightarrow$  por prop 1 se tiene

6: Si no hay p.c., la prop 4 también se cumple para nodos no-raíz.  
(el rank solo aumenta cuando uno 2 árboles con raíces de igual rank... se "duplica" la cantidad de nodos, al menos)  
(pues alguna vez fisionan raíces, y sus hijos quedan fijos al dejar de ser raíz)

$\Rightarrow$  si no hay p.c. todo nodo de rank k tiene  $\geq 2^k$  nodos en su árbol.

A demás, nodos distintos de igual rank no pueden tener descendientes comunes (por 1) (uno no puede ser antecesor del otro)  $\uparrow$

$\Rightarrow$  debe haber  $\leq \frac{n}{2^k}$  nodos de rank k para cada k

(la propiedad se conserva cuando hay p.c., pues no cambia los ranks)

"Definimos"  $\lg^* n$  como cuantas veces debo aplicar  $\log$  para llegar a un  $n^o \leq 1$ .

$$1 \mapsto 0$$

$$2 \mapsto 1$$

$$2^2 \mapsto 2$$

$$2^2 \mapsto 3$$

$$2^{2^2} \mapsto 4$$

$$2^{2^2} = 2^{65536} \mapsto 5$$

$$\lg^* n = \begin{cases} n & \text{si } n \leq 1 \\ 1 + \lg^*(\lg(n)) & \text{si } n > 1 \end{cases}$$

Consideremos los ranks de la estructura y hagamos grupos:

$$[\lg^{*-1}(i), \lg^{*-1}(i+1)]$$

$$\rightarrow 1 (0, 1]$$

$$\rightarrow 2 (1, 2]$$

$$\rightarrow 3 (2, 2^2]$$

$$\rightarrow 4 (2^2, 2^{2^2}]$$

$$\rightarrow 5 (2^{2^2}, 2^{2^{2^2}}]$$

Prop: todo rank  $> 0$  cae en los primeros  $\lg^* n$  grupos.

(pues ranks  $\leq \lfloor \lg n \rfloor$ )

### • Contabilidad

- Cuando un nodo deja de ser raíz, le asignamos dulces  $\ddagger$   
Le damos  $2^k$  si cae en el grupo  $k$   
 $\Rightarrow$  pagamos  $\leq n \lg^* n$

¿Por qué? por (b), el nº de nodos con rank  $\geq k+1$  es

$$\frac{n}{2^{k+1}} + \frac{n}{2^{k+2}} + \dots \leq \frac{n}{2^k} \Rightarrow \text{los nodos del grupo } k \text{ necesitan a lo más } n \text{ dulces}$$

Luego, como hay  $\leq \lg^* n$  grupos, pago un total de  $n \lg^* n$  en este punto.

$n$  = cantidad de nodos de la estructura.

¿Cómo Analizar Find?

- Find está acotado por el  $n^o$  de punteros que se siguen hasta llegar a una raíz.

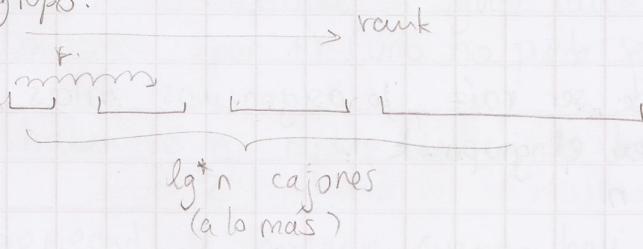
Dado  $\text{Find}(x)$ , hay 3 casos sobre  $x$ :

- 0:  $\text{padre}(x)$  es una raíz (solo pasa 1 vez por Find)
- 1:  $\text{rank}(\text{padre}(x))$  está en un grupo  $\neq$  al de  $\text{rank}(x)$
- 2:  $\text{rank}(\text{padre}(x))$  está en el mismo grupo que  $\text{rank}(x)$

1: en este caso, a lo más  $\lg^* n$  nodos pueden estar en un grupo más alto.

2: Se nos pide 1 para subir al padre. Cada vez que subimos, rank del "nodo presente" debe crecer

$\Rightarrow$  Antes de pagar  $2^k$ , vamos a llegar a un nodo de otro grupo.



- todos los saltos dentro del mismo cajón son GRATIS
- Si cambio de cajón, me cuesta 1.

$\Rightarrow$  Luego el costo está acotado por  $\lg^* n$  cajones.

• Una vez que estoy en un nodo con un rank de un grupo más alto, esto se puede mantener así, pues  $\text{rank}(\text{padre})$  no baja

$\text{rank}(x)$  queda constante

$\Rightarrow x$  puede pagar hasta ser un caso 1.

Partiendo de una estructura vacía,  $m \geq n$  finds y  $n-1$  unions toman

$\Theta(m \lg^* n)$  ops.

$\leq 5$  cualquier cosa es  $\ll 2^{65,522}$