

Aux #8

2015年11月9日 (月)

[P1]

n elementos; $k = \lceil \log(n+1) \rceil$

↳ k arreglos ordenados A_0, \dots, A_{k-1}
pero no tienen orden entre sí.

$A_0 [1] 1$ elto
 $A_1 [xx] 2$
 $A_2 [\dots] 4$
 $A_3 [\dots] 8$
⋮
 $A_{k-1} [\dots] 2^{k-1}$ eltos

} completamente llenos o completamente vacíos.

• BÚSQUEDA:

→ búsqueda binaria - "secuencial", es decir, una por arreglo no vacío.

En el peor caso busco en todos los arreglos

$$\sum_{i=0}^{k-1} \log_2(2^i) = \sum_{i=0}^{k-1} i = \frac{k(k-1)}{2} = O(k^2) = O(\log^2 n)$$

Diremos que eso es suficientemente bueno... no es mucho más que $O(\log n)$

• INSERCIÓN:

Idea: la estructura "representa n en base 2", donde un arreglo es lleno si 1 o vacío si 0, (ni)

Luego insertar un elemento podría hacerse como "sumar 1",
obviamente sin perder las invariantes (arreglos ordenados, llenos
o vacíos)

→ Creamos $A' = [x]$

a) Si A_j está vacío, $A_j \leftarrow A'$ y fin

b) Si no, $A' \leftarrow \underbrace{A_j + A'}_{\text{merge}}$; $j \leftarrow j+1$, volver a a)

8 # xA

En este caso, el "+" es una op. de merge

Pior Caso: hago merge de todos los arreglos.

⇒ k merges. Cada merge es de 2^i elementos

$$\Rightarrow T(n) = \Theta\left(\sum_{i=0}^{k-1} 2 \cdot 2^i\right) = \Theta(2^k) = \Theta(n)$$

Mejor hagamos análisis amortizado.

Sean m operaciones de inserción sobre una estructura con n elementos.

Notemos que A_0 participa en un merge cada 2 inserciones.

$A_1 \dots$

$A_r \dots$

4 inserciones

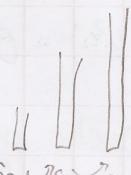
2^{r+1} inserciones

$$\Rightarrow \text{Costo total de merges: } \sum_{i=0}^{k-1} 2^{i+1} \left[\frac{m}{2^{i+1}} \right] \leq k \cdot m = m \cdot \Theta(k)$$

cuánto cuesta
mergear A_i y A'

$\Rightarrow m \cdot \Theta(\log n)$
en cuántos
merges participa A_i
 $\Theta(\log n)$
por op

P2



j-ésimo: tamaño 3^j
stack

$S_0 \cup S_1 \cup S_2$

• Pior caso: mover todo $\Theta(n)$

• n ops en un m.s. vacío

$$\Phi = 2 \sum_{j=0}^{t-1} n_j (\log_3 n - j) \quad (\text{dados...})$$

n_j de elementos en S_j

Vamos a calcular el costo de vaciar el stack Si en el Siti

Costo

$$\text{amortizado} = \Delta \Phi + 2 \cdot 3^i$$

$$\phi = 2 \sum_{j=0}^{t-1} n_j (\log_3 j) \quad \begin{cases} \text{Esto puede verse como que cada} \\ \text{elemento tiene un potencial } \phi_{elem}. \\ \phi_{elem} = 2(\log_3 n - j), \text{ donde } j \text{ es el índice} \\ \text{del stack en el que está.} \end{cases}$$

$$\Rightarrow \phi > 0 \quad ?$$

$$\text{Costo amortizado} = \Delta\phi + 2 \cdot 3^i$$

$$\phi_f = 2 \cdot 3^i (\log_3 n - (i+1))$$

$$\phi_i = 2 \cdot 3^i (\log_3 n - i)$$

solo considero los elementos que se están "moviendo"

$$\Rightarrow \Delta\phi = -2 \cdot 3^i$$

$$\Rightarrow \hat{C} = \Delta\phi + C = -2 \cdot 3^i + 2 \cdot 3^i = 0$$

\Rightarrow vaciar stacks es "gratis"

C Cuánto cuesta un push entonces?

por análisis anterior

$$\begin{aligned} \hat{C}_{total} &= C_{push_{S_0}} + \underbrace{\Delta_{vaciar}_{stacks}}_{stacks} + \Delta\phi_{push_{S_0}} + \Delta\phi_{vaciar_{stacks}} \\ &= 1 + (\phi_f - \phi_i) \quad \Delta\phi_{push_{S_0}}: \phi_f = 2 \cdot \log_3 n \\ &= 1 + 2 \log_3 n = \Theta(\log n) \quad \phi_i = 0 \\ &\quad = 2 \log_3 n \end{aligned}$$

y para n pushes $\rightarrow \Theta(n \log n)$

También este problema se puede hacer con un método parecido al problema anterior.

- n ops de push y pop.

C Cómo se hace un pop?

- $i = 0$

- Se hace pop de S_i

- Si se vacía, se llena con elementos de S_{i+1} (se puede repetir recursivamente)

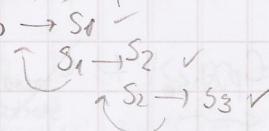
Consideremos un stack S_i . Un push o pop es **relevante** para S_i , si el mayor stack tocado es S_i .

Cada push relevante move $\sum_{j=0}^{i-1} 3^j = \Theta(3^i)$ elem al siguiente stack; Cada pop relevante move $\sum_{j=0}^{i-1} 3^j = \Theta(3^i)$ elem al stack anterior.

\Rightarrow cada op. come en tiempo $\Theta(3^i)$

Antes de un push relevante, todos los stacks antes de S_i están llenos y S_i está a lo mas $2/3$ lleno

Despues, todos excepto S_0 (que queda vacío) quedan en $1/3$ (justo) y S_i queda al menos $1/3$ lleno.



Para un pop relevante:

$S_0 \dots S_{i-1}$ vacío $\Rightarrow S_0 \dots S_{i-1} 2/3$

S_i al menos con $1/3$ $\Rightarrow S_i$ a lo más $2/3$

La primera op. relevante debe ser un push.

Antes, debe haber $\sum_{j=0}^{i-1} 3^j = \Theta(3^i)$ pustes (para llenar todo)

Entre 2 ops relevantes, debe haber al menos $\sum_{j=0}^{i-1} 3^{j-1} = \lceil 2(3^i) \rceil$ ops irrelevantes

Luego si pagamos $\Theta(1)$ por cada stack en cada op, el pago total es suficiente

\Rightarrow el costo total es $\Theta(\log n)$ (pues hay $\Theta(\log n)$ stacks)

Select: Posición de j-ésimo 1

rank: cuántos 1's hay hasta la j-ésima pos.

P3

Abriremos SelectNext \equiv SN

$$\rightarrow SN(j) = \text{Select}(1 + \text{rank}(j-1))$$

$$\rightarrow \text{bloques de tamaño } \frac{\log n}{2}, \text{ super bloques de tamaño } S = b \cdot \log n = \frac{\log^2 n}{2}$$

Para cada superbloque j, con $1 \leq j \leq \lceil \frac{n}{S} \rceil$ calculamos

(Nx son tablas precalculadas)

$$N_S[j] = SN(j \cdot s + 1)$$

\hookrightarrow selectNext del primer elto del superbloque.

Espacio: $\Theta\left(\frac{n}{\log n}\right)$ bits.

Para cada bloque de un superbloque,

$N_b[i] = SN$ del 1^{er} elto del bloque con respecto a ESE bloque

Espacio: $\Theta\left(\frac{n \log \log n}{\log n}\right)$ ~ los elemns de N_b son posiciones en un bloque de tamaño $< n$

Finalmente, para cada posible bloque S de tamaño b y posición i,

$N_p[S, i] = SN$ de i, para la secuencia S.

(notar que, por ejemplo, si $S = \Theta$ (solo 0's), $N_p[S, i] = b + 1$

$$\hookrightarrow 2^b \cdot b \cdot \log b = \sqrt{n} \log \log n \dots \text{el mismo truco de la clase aux anterior}$$

$\underbrace{\quad}_{\substack{\text{el más chico primero}}}$

Si fuera de rango (del futuro), voy a bloque más grande.

se sale de rango

Otro ejercicio de esto mismo, RMQ

range minimum query?

ALGORITMOS EN LÍNEA / ON LINE

2015年11月10日 (火)

Consideremos el siguiente problema:

Quiero jugar en cierto juego, pero no sé cuántos días jugaré antes de aburrirme

- Una tienda lo arrienda con costo de $\$A$ / día.
- Otra lo vende a $\$B$.

¿Qué hacer? Quiero minimizar mi gasto.

Si supiera el n° de días de uso del juego, d , la solución es comparar $A \cdot d$ con B , elijo comprar si $B < A \cdot d$ (y viceversa)
 \Rightarrow costo es $\min(A \cdot d, B)$

Si no conocemos d :

Idea: Estrategia "arriendo por \bar{d} días, luego lo compro"

- Si $d \leq \bar{d} \Rightarrow$ me aburro del juego mientras lo arrendaba
 \Rightarrow costo $A \cdot d$
- Si $A \cdot \bar{d} < B \Rightarrow$ el óptimo también arrienda
 \Rightarrow costo igual al óptimo $\Rightarrow \frac{\text{ALG(I)}}{\text{OPT(I)}} = 1$
- Si $A \cdot \bar{d} \geq B \Rightarrow$ el óptimo compraba $\Rightarrow \frac{\text{ALG(I)}}{\text{OPT(I)}} = \frac{A \cdot \bar{d}}{B}$

- Si $d > \bar{d} \Rightarrow$ voy a arrender \bar{d} y luego comprar
 \Rightarrow costo $A \cdot \bar{d} + B$

- Si $A \cdot \bar{d} < B \Rightarrow$ el óptimo arrendaba $\Rightarrow \frac{\text{ALG(I)}}{\text{OPT(I)}} = \frac{A \cdot \bar{d} + B}{A \cdot \bar{d}}$

- Si $A \cdot \bar{d} \geq B \Rightarrow$ el óptimo compraba
 $\Rightarrow \frac{\text{ALG(I)}}{\text{OPT(I)}} = \frac{A \cdot \bar{d} + B}{B} = \frac{A \cdot \bar{d}}{B} + 1$

$$\frac{\bar{d}}{d} + \frac{B}{A \cdot \bar{d}} \leq 1 + \frac{B}{A \cdot \bar{d}} \\ \leq 1 + \frac{B}{A \cdot \bar{d}}$$

El análisis se hace con respecto a un algoritmo óptimo que conoce "todo". Definimos "Competitividad de un algoritmo online A" como

$$C(n) = \max_{\{I|I|=n\}} \frac{A(I)}{OPT(I)} \leftarrow \text{algoritmo óptimo (y que sabe todo)}$$

Entonces se dice que A es $C(n)$ -competitivo.

Queremos limitar $\max \frac{ALG(I)}{OPT(I)} \Rightarrow$ nos conviene $\frac{A}{B} \approx 1$. Si $\frac{A}{B} = \frac{B}{A}$

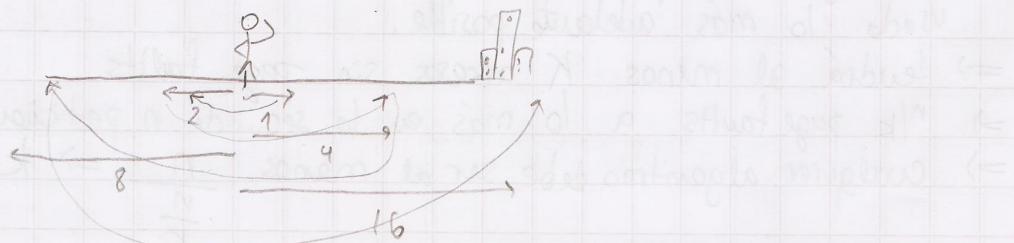
$\Rightarrow C_n(I) \leq 2 \Rightarrow$ la estrategia es 2-competitiva

- Otro problema Imagine despertar en una larguísima carretera
 $? \rightarrow ?$ \rightarrow quiero minimizar los pasos dados.

→ No sabemos en qué dirección está la ciudad
 El algoritmo óptimo es caminar en la "dirección correcta"
 $\Rightarrow OPT(n) = n$

Estrategias como "caminar hacia la derecha" tienen costo "máximo" no acotado (puedo irme en la dirección incorrecta)

Idea: caminamos en una dirección; giro al haber recorrido 2^i , vuelvo y recorro 2^{i+1} hacia la otra dirección.



Algoritmos EN LÍNEA (ON LINE)

Supongamos encontramos la ciudad entre 2^i y 2^{i+1}
La distancia recomendada es $\Rightarrow i = \lfloor \log_2 n \rfloor + 1$

$$d = 2 \sum_{i=0}^{\lfloor \log_2 n \rfloor + 1} 2^i + (n) = ALG(I)$$

↳ lo encontré, en el peor caso, en la dirección opuesta a la que partí.

$$\begin{aligned} &< 2 \cdot (2^{\lfloor \log_2 n \rfloor + 2}) + n \\ &\leq 2 \cdot (4n) + n = 9n \Rightarrow A(I) \leq 9n \\ &\Rightarrow C(n) \leq 9 \end{aligned}$$

Otro Problema: Paganado.

- La información que no sabemos es la secuencia de operaciones
- El costo se da ante la petición de una página que está en disco (page fault)
- Nos limitaremos a una memoria de tamaño K y $K+1$ páginas de información.

Pior Caso: el "adversario" siempre pide **la** página que tenemos en disco.

\Rightarrow Si existe un algoritmo que produce n page faults $\Rightarrow ALG(I) = n$

Cómo comparemos con el óptimo? El óptimo "conoce el futuro".
 \Rightarrow el algoritmo óptimo envía a disco la página que será usada lo más adelante posible.

\Rightarrow requerirá al menos K accesos sin page faults.

\Rightarrow n/K page faults a lo más en la sec. de n peticiones

\Rightarrow cualquier algoritmo debe ser al menos $\frac{n}{K}$ $\Rightarrow K$ -competitivo

(本) はい なぜなら

instacOPT global fail

¿Qué algoritmos hay?

- LRU: last recently used → k-competitivo
- FIFO → k-competitivo
- LFU last freq. used → NO es k-competitivo
- ⋮

LRU

Dividiremos la secuencia de ops en bloques que terminan cada k page faults de LRU. En cada bloque, OPT sufre un pf. al menos 1 vez.

- Sea p la página pedida antes de un bloque B (p es la MRU)
- Si uno de los pf's en B trae p a memoria, p fue el LRU en algún momento \Rightarrow todos los demás fueron consultados \Rightarrow OPT falló alguna vez.
 - Si una página p' se repite en los pf's
 - \Rightarrow entre los 2 pf's deben haberse pedido todos.
 - \Rightarrow OPT falla 1 vez entre las 2 peticiones.
 - Si los k fallos son diferentes \Rightarrow OPT siempre falla 1 vez por cada k fallos. de LRU.
 - \Rightarrow LRU es k-competitivo.

List Update Problem

2015年11月12日(木)



Accesar (i) \rightarrow costo i
Intercambiar ($i, i+1$) \rightarrow costo 1

Aplicación: compresión

codifico \rightarrow símbolos s_1, \dots, s_N (ordenados en tamaño)
elementos s_1, \dots, s_N

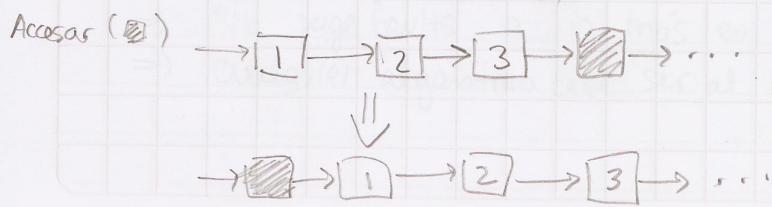
Para comprimir s : lo busco en la lista. Si está en la posición i , uso s_i
 \Rightarrow Si el costo de los accesos se minimiza voy a minimizar el
largo del texto comprimido. La idea es tener una estrategia que
logre hacer que las búsquedas resulten en el rango más cercano,
más al comienzo de la lista posible. De esa forma minimizo
bien, también, el largo del texto comprimido.

Al buscar puedo también intercambiar elementos s_j de acuerdo
a alguna estrategia bien definida (no aleatoria, p.e., para poder
usarlo al revés al descomprimir y tener resultados coherentes.)

Queremos una estrategia de reordenamiento (on-line) tal que dada
una secuencia de accesos s_1, s_2, \dots minimice el costo de acceso.

Recordemos que el óptimo conoce s_1, s_2, \dots, s_N y también puede
invocar intercambios.

Idea: mover el elemento accedido a la primera posición
(Move-to-Front)



Costo de acceder al k -ésimo
elemento: $2k-1$
(k para llegar,
 $k-1$ swaps)

La estructura cambia de forma dinámica según la secuencia de operaciones. \Rightarrow tiene sentido hacer un análisis amortizado

Mostraremos que el costo de MTF es a lo más 4 veces de OPT.

Def: Inversión

Una inversión en una lista A clr a una lista B es un par (x, y) tq:

- x está antes de y en A } o viceversa
- x está después de y en B

Usaremos la sgte. función potencial

$$\phi = 2 \cdot \# \text{ de inversiones de } L_{MTF} \text{ clr a } L_{OPT}$$

$$\phi_0 = 0 \text{ (al pp. } L_{MTF} \text{ y } L_{OPT} \text{ son iguales)}$$

$$\phi > 0 \checkmark$$

Demostraremos $C_{MTF} + \Delta\phi \leq 4 \cdot C_{OPT}$

Sea x un elemento en la secuencia de peticiones

$i =$ posición de x en L_{OPT}

$k =$ posición de x en L_{MTF}

Consideraremos a parte los swaps que OPT puede hacer.

$\Rightarrow C_{OPT} = i$ (Solo lo busca... el análisis de los swaps que hace los swaps veremos después)

- Hay $k-1$ elementos antes de x en L_{MTF} ; todos estos elementos quedan después de x después de la operación.

$$\Rightarrow \# \text{ de inv. creadas} + \# \text{ de inv. destruidas} = k-1$$



- Sea y un elemento antes de x en L_{MTF} antes de los swaps
 Para que se cree una nueva inversión, y debe estar antes de x en L_{OPT} \Rightarrow existen a lo más $(i-1)$ y 's
 \Rightarrow a lo más se crean $i-1$ inversiones
 \Rightarrow el resto deben ser destrucciones
 $\Rightarrow k-1-(i-1) = k-i$ deben ser destrucciones (a lo menos)
 $\Rightarrow \Delta\phi \leq 2(i-1 - (k-i))$
 $= 2(2i - k - 1)$
 $= 4i - 2k - 2$

$$\Rightarrow C_{MTF} \leq C_{MTF} + \Delta\phi \leq 2k-1 + 4i - 2k - 2 \\ = 4i - 3 \leq 4i = \boxed{4 \cdot C_{OPT}}$$

sobre secuencia de ops.

Queda ver qué pasa con los swaps que OPT pueda hacer

- A MTF le cuesta 0 (tr a este swap, MTF no está haciendo nada)
- A OPT le cuesta 1

Analizamos por partes

- Los alg. no dependen del otro
- 1) OPT retorna alto
 - 2) MTF retorna alto
 - 3) MTF hace swaps
 - 4) OPT hace swaps

Un swap de OPT crea o destro 1 inversión
 $\Rightarrow \Delta\phi = \pm 2$ ($\phi = 2 \cdot \# \text{ inversions}$)
 $\Rightarrow C_{MTF} \leq C_{MTF} + \Delta\phi \leq 2 = 2 \cdot 1$
 $= 2 \cdot C_{OPT}$
 $\leq 4 \cdot C_{OPT}$

\Rightarrow MTF es 4 competitivo

- Si se permite intercambiar pares de elementos arbitrarios con costo 1 = $\frac{1}{2}$ OPT "vence" a cualquier algoritmo outline por un factor de al menos $\frac{n}{2}$, sobre secuencias arbitrariamente largas y escogidas aleatoriamente ($n=|L|$)

Problema Tinder

- Tengo n participantes, quiero elegir a quien tenga el mejor puntaje.
- Se puede festejar a 1 participante a la vez.
- Luego de festejar y obtener su puntaje, pedo
 - a) "Me quedo con este participante"
 - b) "Adiós para siempre"

El algoritmo offline conoce n y el puntaje de cada participante; de antemano.

El online conoce n .

C) Qué quiero optimizar?

→ la probabilidad de encontrar al mejor participante
 $(\Rightarrow \text{OPT logra } P=1)$

Estrategia:

- Permuto aleatoriamente los participantes.
- Calibrámos usando las primeras t citas
 $(\text{elegimos "Adiós para siempre" en todas ellas})$
- Recordámos el mejor puntaje P
- En las siguientes $n-t$ citas, si vea un puntaje mayor que P , me caso.



Sean $1 \dots n$ los participantes y t es la persona con mayor puntaje, etc, luego de permutar usando una permutación Π , el problema es buscar el mínimo de $\Pi[1] \dots \Pi[n]$

El algoritmo es entonces.

$$\cdot P \leftarrow \min \{\Pi[1], \dots, \Pi[t]\}$$

$$\cdot j^* \text{ es el } 1^{\text{er}} \ j \geq t+1 \text{ tq. } \Pi[j] < P$$

$$\circ \mathbb{P}(\Pi[j] = 1) ?$$

$$\sum_{j=t+1}^n \mathbb{P}[\Pi[j] = 1] \text{ y nos quedamos con la persona } j$$

• Si $\Pi[j] = 1$ para algún $j > t$, ¿cuándo fallamos?

Fallamos si entre las citas $t+1$ y $j-1$ apareció alguien mejor que entre 1 y t .

$\Rightarrow \min \{\Pi[1], \dots, \Pi[j-1]\}$ debe estar en una posición $\in (t+1, j-1)$

Si este min está en una posición $\in (1, t)$, vamos a elegir a la persona correcta.

$$\Rightarrow \mathbb{P}[\Pi[j^*] = 1]$$

$$= \sum_{j=t+1}^n \mathbb{P}[\Pi[j] = 1] \text{ y } \min(\{\Pi[1], \dots, \Pi[j-1]\}) \in \{\Pi[1], \dots, \Pi[t]\}$$

$$= \sum_{j=t+1}^n \frac{1}{n} \frac{t}{j-1}$$

¿Por qué?

$$\mathbb{P}[\Pi[j] = 1 \text{ y } (\min \dots)] \Rightarrow t = \frac{n}{\mathbb{E}}$$