

## CC4102 - Diseño y

## Análisis de Algoritmos.

Prof: Gonzalo Navarro

af. 312

gnavarro@dcc.uchile.cl

1 [ Cotas inferiores  
Experimentación

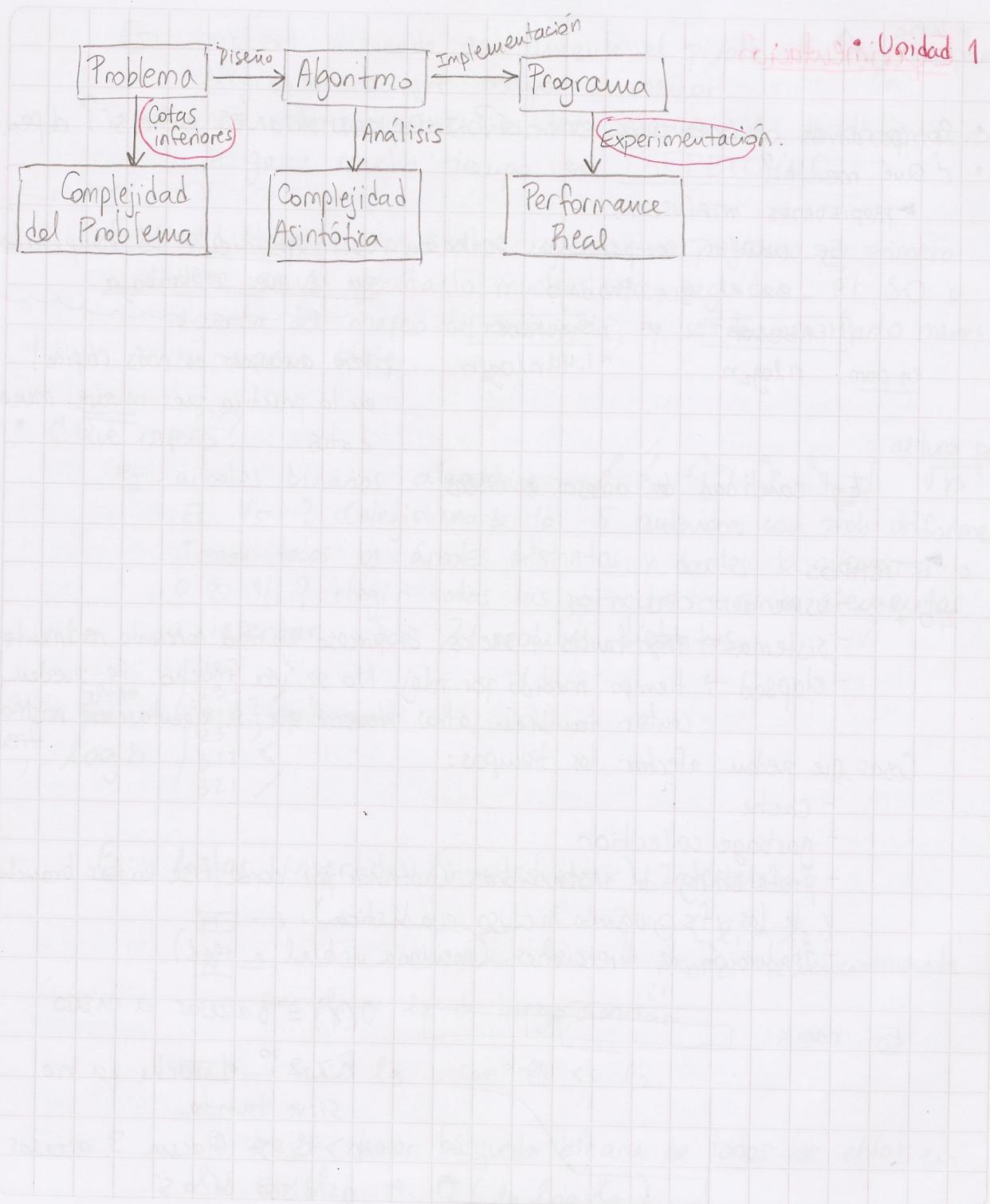
2 { Memoria Secundaria  
- ordenar  
- colas de prioridad  
- hashing

~3 { Algoritmos avanzados  
- amortización  
- universos discretos  
- competitividad

A { Algoritmos no convencionales  
- probabilísticos / aleatorizados  
- aproximados  
- paralelos  
Examen

- 2/3 • 2 controles y 1 Examen
- 1/3 • 3 Tareas (en pareja o de a 3)  
Si reprobaban tareas  
→ Habrá una tarea recuperativa  
(Haciendo hecho de nuevo las tareas reprobadas)
- Exención con 5.0

2015年9月1日 (火)



2015年9月3日(木)

## Experimentación

- Comparamos ALGORITMOS entre sí? IMPLEMENTACIONES entre sí? depende
- ¿Qué medir?

### ► PROPIEDADES INTRÍNSECAS:

Ej: cuántas comparaciones se hacen. Es independiente del computador en el que se trabaja.

MERGESORT vs QUICKSORT  
en prom  $n \log_2 n$        $\approx 1.44n \log_2 n$

; pero quicksort es más rápido!  
en la práctica, pues mueve menos datos.

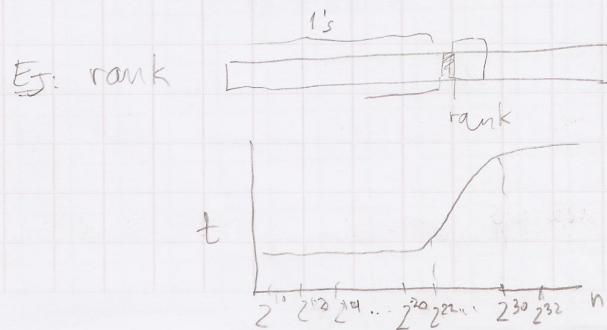
Ej: cantidad de accesos a discos.

### ► TIEMPOS

- I/O +
- usuario  $\rightarrow$  CPU
  - sistema  $\rightarrow$  page-faults y seeks. El proceso no está corriendo realmente
  - elapsed  $\rightarrow$  tiempo medido por reloj. No se usa mucho. Se pueden contar también otros procesos que se ejecutan al mismo tiempo

Cosas que pueden afectar los tiempos:

- Cache
- garbage collection
- prefetching de instrucciones (adivinar por donde irá en las branches de los ifs, cuando incluso esta distíctica)
- traducción de direcciones (memoria virtual a real)



$$\Theta(1) = 3 \text{ accesos a disco}$$

$n > 2^{20}$  el cache ya no sirve tanto.

$n > 3^0$  se hacen 3 accesos a disco si o si

Ej: quicksort eligiendo astutamente el pivote  
→ eligiendo siempre mayor o menor  
hay más comparaciones, más movimiento de datos, pero  
se gana mucho tiempo en PREFETCHING

Ej: cold state / warm state : ejecutar un programa por primera vez, versus ejecutarlo muchas veces seguidas. El SO se acuerda del mapeo de memoria, y se demora mucho menos en warm state.

- ¿Qué inputs?

Es  $\frac{1}{5}$ ? (elegir uno de los 5 anteriores con prob uniforme.)

Tomar todos los árboles distintos y dárles la misma prob.

O es  $\frac{1}{6}$ ? elegir todas las permutaciones de elementos a insertar. Son 2 modelos distintos.

$$\log h \quad \left\{ \begin{array}{l} 123 \\ 132 \\ 213 \\ 231 \\ 312 \\ 321 \end{array} \right\} \xrightarrow{\text{3}} \frac{1}{3} \text{ prob.}$$

Ej: listas invertidas (inverted index), Intersección

$$P_1 \rightarrow \text{unseen} \quad l_1 \quad ? \quad \theta(l_1 + l_2)$$

[p2] → unum l2 ) recorrer secuencialmente

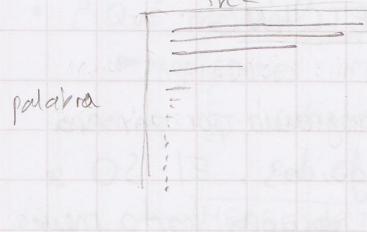
con la y la de largo similar

P3  $\rightarrow$  nro l3 con  $l_3 \ll l_2$

es mejor hacer búsqueda binaria de todos los  $l_3$  en  $l_2$ .  $\rightarrow \mathcal{O}(l_3 \log l_2)$



Experimentalmente como elijo cual de los 2 métodos usar?  
Aleatoriamente? pero pocas palabras aparecen MUCHO y  
muchas palabras aparecen POCO (Zipf's law).



En este caso aleatoriamente no es muy buena idea experimentar para decidir qué método elegir, pues en la realidad la acción no es nada aleatoria!

Otro ejemplo es elegir una buena función de hashing para palabras, p.e. suma de carac ascii, tomar los primeros 4 carac... problema: comparten prefijos, palabras que tienen las mismas letras, etc.  
Probar permutaciones de letras aleatoriamente es pésima idea, porque el desempeño de mis funciones de hashing pararía que se comportan bien, pero en la maldad NO!

Otro ejemplo: elegir nodos al azar en un árbol... 50% de prob. de elegir una hoja! a veces eso puede ser muy poco realista.

2015年9月10日 (木)

## Cotas inferiores de problemas

- Adversario (combate naval contra alguien que pone los barcos al final)
- Reducción (transformar un problema a otro)
- Teoría de la información (permite incluso deducir cotas inf promedio)  
(analiza redundancia de la información)

- BÚSQUEDA EN UN ARREGLO DESORDENADO.  $A[1, n]$ 
  - min # accesos en el peor caso =  $n$   
¿por qué? hay que razonar en base a cualquier algoritmo. No importa el orden en que lo haga, si reviso  $n-1$  celdas, en el peor caso siempre el número va a estar en la celda que falta.  
(como el ejemplo de combate naval)
  - min # accesos en arreglo ordenado? con búsq. binaria sabemos que el peor caso es logn  $\rightarrow$  el adversario tiene más restricciones con respecto a donde poner el elemento maliciosamente.

- ENCONTRAR EL MÍNIMO DE  $A[1, n]$ .

# comparaciones  $n-1$

$m \leftarrow A[1]$

for  $i \leftarrow 2$  to  $n$

if  $A[i] < m$

$m \leftarrow A[i]$

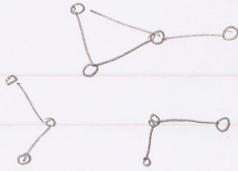
return  $m$ .

return  $m$ .

También podemos pensar en el TORNEO...  
se ve lindo, es un buen algoritmo paralelo  
pero también se realizan  $n-1$  comp.

¿Podemos hacer menos de  $n-1$  comp?

podríamos usar un argumento similar al del problema anterior  
pero... ¡en el primer paso del torneo TODOS los elementos  
ya se compararon con otro!



los arcos son comparaciones

Argumentamos con grafos. Debemos hacer suficientes comparaciones para producir un grafo CONEXO  $\rightarrow$  si tengo  $n$  nodos, necesito  $n-1$  arcos para producir un grafo conexo.

Lo importante es que el adversario puede seguir siendo consistente cuando coloca el mínimo en una componente conexa que he descartado (pues el adversario solo dice respuestas SÍ - NO)

Vamos a modelar el conocimiento del algoritmo sobre el input de otra forma

$(a, b, c)$

- $a = \#$  elementos nunca comparados.
- $b = \#$  elementos comparados alguna vez y siempre menores
- $c = \#$  elementos comparados alguna vez y alguna vez mayores

Cualquier algoritmo correcto parte de  $(n, 0, 0)$  y llega a  $(0, 1, n-1)$

¿Cómo es la evolución  $(n, 0, 0) \rightarrow (0, 1, n-1)$ ?

	a	b	$\stackrel{=}{c}$
a	$(a-2, b+1, c+1)$ para los 2 casos	$(a-1, b, c+1)$	$(a-1, b+1, c)$
b	lo mismo	$(a, b-1, c+1)$	$(a, b, c)$ $(a, b-1, c+1)$
c	lo mismo	lo mismo	$(a, b, c)$

en el peor caso el adv. Puede hacer que ocurra eso sin entrar en inconsistencias

→ NO SIRVE, no me conviene

¡Es directo!

Para pasar de  $c=0$  a  $c=n-1$ , como en cada paso  $c$  aumenta a lo sumo en 1, necesito hacer  $n-1$  comparaciones.

Otra cosa interesante es que deduzco las comparaciones que más me convienen!  $\rightarrow$  Puedo deducir un algoritmo

Alternativa 1: comparar  $a$  con  $a \rightarrow (n-2, 1, 1)$  y luego hacer el resto de las comparaciones  $a$  con  $b$   
↳ algoritmo for!

Alternativa 2: comparar a con a  $\frac{n}{2}$  veces  $(n, 0, 0) \rightarrow (0, \frac{n}{2}, \frac{n}{2})$   
y luego seguir comparando b con b.

- ENCONTRAR EL MÁX Y EL MÍN DE  $A[1, n]$ .  
 $n-1 + n-2 = 2n-3$  comp?

Extendamos el modelo anterior...

$$(h, 0, 0, 0) \rightarrow (0, 1, 1, n-2)$$

a	b	c	d
a	(a-1, b+1, c+1, d) si "a" gana (a-1, B, c+1, d)	(a-1, b, c, d+1) si "c" gana (a-1, b+1, c, d)	(a-1, b+1, c, d) (a-1, b, c+1, d)
b	(a, b-1, c, d+1) "b" gana (a, b-1, c-1, d+2)	(a, b, c, d) si "b" gana (a, b-1, c, d+1)	(a, b, c, d) si "b" gana (a, b-1, c, d+1)
c		(a, b, c-1, d+1) <del>(a, b, c, d)</del>	(a, b, c, d) <del>(a, b, c, d)</del>
d			(a, b, c, d)

- Notemos que la gran masa va a d.
  - d crece 'a lo sumo' de a 1 (adversario) y en ese caso a NO decrece.  
→ n - 2 comp necesarias.

Para hacer pasar a de n a 0, hago  $\left[ \frac{n}{2} \right]$  comp.

$\Rightarrow$  cota inferior  $\lceil \frac{3}{2}n \rceil - 2$

¿Podemos derivar un algoritmo que nos lleva a la cota inferior?

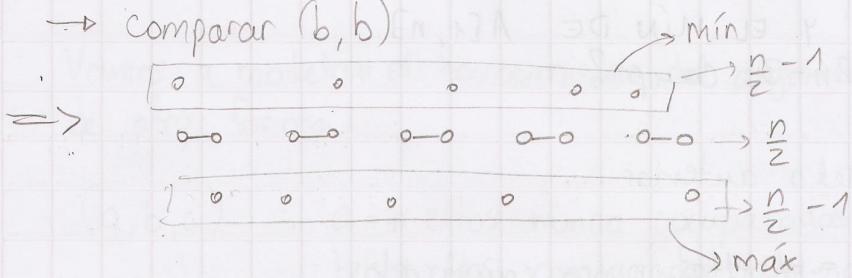
Veamos...

→  $(a, b)$  no me conviene mucho... es como el faro.

→ Es mejor empezar a comparar pares desconocidos  $(a, a)$

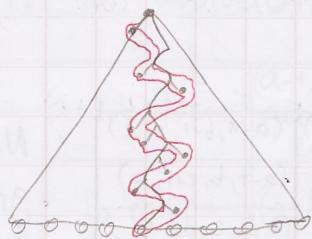
$$(n, 0, 0, 0) \xrightarrow{\eta_2} (0, \frac{n}{2}, \frac{n}{2}, 0) \hookrightarrow \text{más rápido}$$

→ Comparar  $(b, b)$



Como encontré un algoritmo, ahora SÉ que no hay cota inferior más alta. (A veces hay gap)

• MÁX y  $2^{\text{MÁX}} - 1 + \lceil \log_2 n \rceil$



¿Es el óptimo? Aux.

Hacer la tablita del adversario es bien complicado.

Esto fue ADVERSARIO.

## ★ Técnicas de reducción:

Si A tiene una cota inferior de  $f(n)$  y se puede reducir a B en tiempo  $\Theta(f(n))$  entonces B tiene una cota inferior  $\Omega(f(n))$

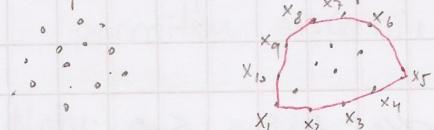


Ej. Ordenar es  $\Omega(n \log n)$

→ Cápsula convexa "convex hull"

input

output

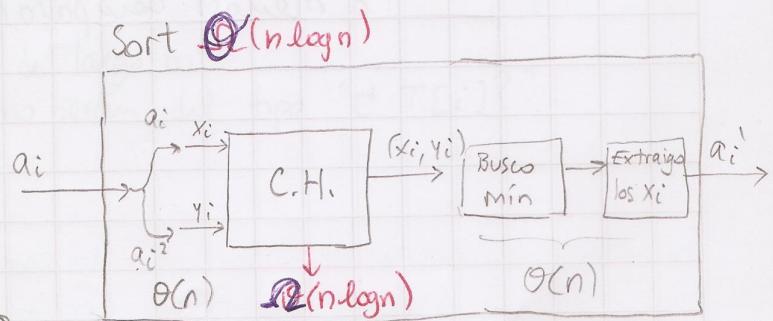
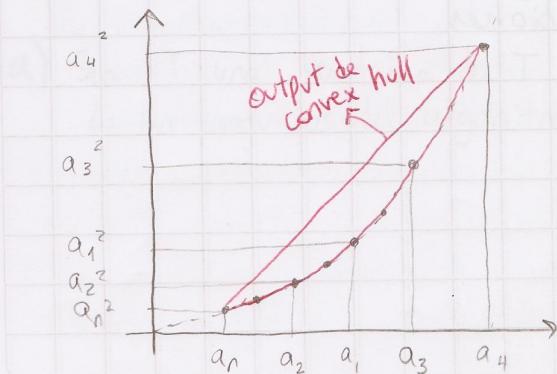


$x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, x_9, x_{10}$

minimo (en área) polígono que contiene a todas las puntos. Entrega una lista ORDENADA de los puntos (recorre los puntos en sentido antihorario)

Tomemos un problema de ordenamiento y lo transformaremos a un problema convex hull. Luego veremos cómo, de la solución de convex hull podemos traducir "fácilmente" a la solución del problema de ordenamiento.

$$(a_1, a_2, a_3, \dots, a_n) \xrightarrow{\quad} (a_1, a_1^2), (a_2, a_2^2), (a_3, a_3^2), \dots, (a_n, a_n^2).$$



Si C-H. se demorara menos, contradicción con SORT.

# Auxiliar 1 - Cotas Inferiores

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro      Auxiliar: Jorge Bahamonde

14 de Septiembre del 2015

1. Demuestre que se requieren al menos  $2n - 1$  comparaciones, en el peor caso, para fusionar dos listas ordenadas de tamaño  $n$  en una de tamaño  $2n$  ordenada.
2. El problema de *búsqueda aproximada en texto* consiste en, dado un string  $T[1, n]$ , llamado *texto*, otro string más corto  $P[1, m]$ , llamado *patrón*, y un entero  $0 \leq k < m$ , determinar si  $P$  ocurre en  $T$  permitiendo a lo más  $k$  *errores* (inserciones, borrados o reemplazos de caracteres en  $P$  (o  $T$ ). Ambos  $T$  y  $P$  son secuencias de caracteres de un alfabeto  $[1, \sigma]$ .  
Yao demostró en 1976 que no es posible resolver el problema de búsqueda exacta ( $k = 0$ ) en menos de  $\Omega(n \log_\sigma(m)/m)$  inspecciones de caracteres de  $T$  en promedio (el promedio supone que los caracteres de  $T$  se eligen al azar, uniformemente en  $[1, \sigma]$ ).
  - (a) Demuestre que un adversario impide resolver el problema general inspeccionando menos de  $k + 1$  caracteres en cualquier ventana posible de  $T$  de largo  $m$ .
  - (b) Deduzca de lo anterior una cota inferior de la forma  $\Omega(kn/m)$  para el problema, incluso en el caso promedio.
  - (c) Deduzca la cota para el problema  $\Omega(n(k + \log_\sigma m)/m)$  en promedio, demostrada por Chang y Marr en 1994. Ellos también diseñaron un algoritmo con esa complejidad promedio,  $O(n(k + \log_\sigma m)/m)$ . ¿Qué puede decir entonces de la complejidad promedio del problema?
3. Considere el problema de encontrar el máximo y el segundo máximo de un arreglo de  $n$  números. Se sabe que en el peor caso,  $n + \lceil \lg n \rceil - 2$  comparaciones son suficientes. Utilizando la técnica del adversario, demuestre que  $n + \lceil \lg n \rceil - 2$  comparaciones también son necesarias en el peor caso (en el modelo de comparaciones).
4. Demuestre que determinar si un grafo no dirigido es o no conexo toma  $\binom{n}{2}$  consultas del tipo “¿existe un arco entre los nodos  $u$  y  $v$ ?”, si la cantidad total de nodos es  $n$ .

2015年9月14日(月)

## AUX # 1

### "Cotas inferiores"

- $\Theta(g(n)) = \{f(n) : \exists n_0, c > 0, \forall n \geq n_0, 0 \leq f(n) \leq g(n)\}$
- $\Omega(g(n)) = \{f(n) : \exists n_0, c > 0, \forall n \geq n_0, 0 \leq g(n) \leq f(n)\}$
- $\Theta(g(n)) = \Theta(g(n)) \cap \Omega(g(n))$
- $\Theta(g(n)) = \{f(n) : \forall c > 0, \exists n_0 \geq 0, \forall n \geq n_0, 0 \leq f(n) \leq c \cdot g(n)\}$   
↳  $f(n) \in \Theta(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{f}{g} = 1$
- $\omega(g(n)) = \{f(n) : \forall c > 0, \exists n_0 \geq 0, \forall n \geq n_0, 0 \leq c \cdot g(n) \leq f(n)\}$   
↳  $f(n) \in \omega(g(n)) \Leftrightarrow \lim_{n \rightarrow \infty} \frac{g}{f} = 0$

PI

Tenemos 2 listas:

Sea  $l_1 := u_1, u_2, \dots, u_n$  siendo  $u_1 \leq u_2 \leq \dots \leq u_n$

$l_2 := v_1, v_2, \dots, v_n$  siendo  $v_1 \leq v_2 \leq \dots \leq v_n$

- Adversario responde todas las preguntas (comparar  $(a, b)$ ) como si:

$u_1 < v_1 < u_2 < v_2 < u_3 < \dots < u_n < v_n$

↳ el intercalamiento es un mal caso para merge.

Contradicción: Supongamos que el algoritmo hizo  $2n - 2$  (o menos) comparaciones.

Consideremos los pares:

$(u_1, v_1), (v_1, u_2), (u_2, v_2), (v_2, u_3), \dots, (u_n, v_n)$  }  $2n-1$  pares

Pues:  $(u_1 < v_1) < u_2 < v_2 < u_3 < v_3 < \dots < u_n < v_n$

Par 1      Par 2

hay  $2n-1$  pares.

Luego existe un par que no fue comparado, por ejemplo,  $(v_k, u_{k+1})$  sin pérdida de generalidad, supongamos que fue puesto así: (No hizo comp)

Adversario anuncia que  $u_{k+1} < v_k$

... |  $v_k | u_{k+1} | \dots$

↳ las respuestas del adversario no permiten discernir este caso  $\Rightarrow$  El algoritmo NO es correcto.

Nota: Como existe un par no compartido, el algoritmo debió situar dichos elementos en la lista. Basta que el adversario diga que el orden no era el correcto (El inverso) para que el algoritmo falte.  
↳ Además debe ser de la forma  $(u, v)$  o  $(v, u)$  pues no pueden ser situados 2 "u", o 2 "v" ya que ya están ordenados, y contradice el input.

[P2]

$T[1, n]$  texto

$0 \leq k \leq m$  errores permitidos

$P[1, m]$  patrón

$\Sigma$ : alfabeto

a) Sea  $T$  una ventana de  $T$  de largo  $m$ .

↳ las preguntas del algoritmo serán del tipo "¿  $T[i]?$ "

## AUX # 1

Adversario: Dado  $m$  y una ventana de largo  $m$  (igual al patrón). A las primeras " $k$ " consultas distintas responderá con "errores" (como por ejemplo, caracteres que no están en el patrón).

- Si el algoritmo responde que encontró un "match" el adversario "rellena" la ventana con errores  $\Rightarrow$  Algoritmo incorrecto.
- Análogamente, si responde que no hay match, rellena con caracteres del patrón.

Supongamos que tenemos PIZARRA, y hasta 3 errores.

$\hookrightarrow \underline{\text{n}}\text{i}\underline{\text{z}}$   $\underline{\text{a}}$   $\underline{\text{n}}\text{a}$   
↑      ↑    ↑

El adversario tira errores a las 3 consultas del algoritmo.

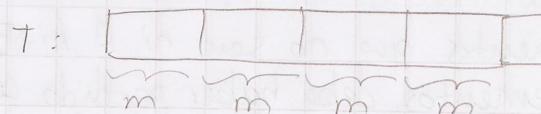
Entonces:

① Algoritmo dice que hay match:  $\underline{\text{n}} \text{ } \underline{\text{n}} \text{ } \underline{\text{n}} \text{ } \underline{\text{h}} \text{ } \underline{\text{h}} \text{ } \underline{\text{n}}$   
Rellena con  $\text{n}$ 's  
 $\hookrightarrow$  Resultado incorrecto.

② Algoritmo dice que no hay match:  $\underline{\text{n}} \text{ } \underline{\text{i}} \text{ } \underline{\text{z}}$   $\underline{\text{n}} \text{ } \underline{\text{n}} \text{ } \underline{\text{a}}$   
Rellena con las letras del patrón. Por ende sí hay match con tolerancia de 3 errores.  
 $\hookrightarrow$  Resultado incorrecto.

Reducción

- b) Tenemos P inicial :  $P_i = \text{"encontrar patrones en cualquier lugar"}$   $\leftarrow$  difícil  
 Patrón :  $P_o = \text{"encontrar patrones que coincidan en i.m, } i \geq 0$



Dividimos el texto en fragmentos de largo " $m$ ". Nos limitaremos a buscar en cada ventana de largo " $m$ ".

Cualquier solución de  $P_i$ , es transformable en una de  $P_o$ . Luego, si toda solución de  $P_o$  toma tiempo  $T \in \Omega\left(\frac{kn}{m}\right)$ , entonces toda solución de  $P_i$  también.

Para  $P_o$ : (adversario)

- Se divide  $T$  en bloques de largo " $m$ " (hay  $\frac{n}{m}$  bloques)
- En cada uno necesita al menos  $(k+1)$  "checks". (Los bloques son disjuntos)  $\Rightarrow$  el total toma al menos  $\Omega\left(\frac{kn}{m}\right)$ ,  $k > 0$ .

en realidad es  $k+1$ , pero la notación  $\Omega$  permite borrar esas constantes.

Lo ideal es tener un texto con ventanas, que es un problema fácil en cada ventana. Luego, con esto podría haber una mejor solución para el problema general, lo que no puede ser. Por ende, si el problema fácil toma al menos  $\Omega\left(\frac{kn}{m}\right)$ , luego el problema difícil se hace al menos en  $\Omega\left(\frac{kn}{m}\right)$ .

- c) Cualquier algoritmo demora al menos:  $\Omega\left(\frac{kn}{m}\right)$  con  $k$  errores y  $\Omega\left(\frac{n \log_2(m)}{m}\right)$  con  $0$  errores por enunciado.  
 $\Rightarrow$  En particular demora al menos el máximo de los 2 tiempos. Además:  
 $(f+g) \in \Omega(\max(f, g)) \Rightarrow \Omega\left(\frac{n \log_2(m)}{m}\right) + \Omega\left(\frac{kn}{m}\right) = \Omega\left(\frac{n(k + \log_2(m))}{m}\right)$   
 ↳ concluimos que es:  $\Theta\left(\frac{n(k + \log_2(m))}{m}\right)$

P3

Sean "n" números con A el máximo y B el segundo máximo.

- Para que un algoritmo responda bien debe saber:

- 1)  $A > B$

- 2) Identificar los  $n-2$  elementos que no son ni A ni B.

$\Rightarrow$  cada uno de estos elementos debe haber perdido contra B, o contra otro de este conjunto.

Por qué? En caso contrario, un adversario puede tomar un elemento que no haya perdido así, e intercambiárselo con B.  
 $\Rightarrow$  hay  $n-2$  comparaciones que no involucran A.

Para encontrar "k", construimos el siguiente adversario:

Adv: crea un arreglo L, donde  $\forall$  elemento del conjunto, se le asigna un conjunto con  $\leq x$  elementos.

$\Rightarrow L[x] \leftarrow \{x\}$ , en donde  $L[x]$  son los elementos que el algoritmo sabe que son  $\leq x$ .

Compare (a, b):

- Si:  $a \in L[b]$  ó  $b \in L[a]$

responder correctamente.

- Si no: si:  $|L[a]| \geq |L[b]|$

$$L[a] \leftarrow L[a] \cup L[b]$$

else:

$$L[b] \leftarrow L[a] \cup L[b]$$

Queremos que las listas vayan creciendo lo más lento posible, pues sabemos que cuando  $|L[x]| = n$ , entonces sabremos que  $x$  es el máximo, por la definición de  $L[x]$  (todos los elementos  $\leq x$ )

• En cada comparación que involucra a  $x$ ,  $|L[x]|$  a lo más se duplica. Claramente  $|L[x]| \leq 2^k$ , con  $k$  número de comparaciones en que ha participado  $x$ . Si el algoritmo es correcto, al final de la ejecución  $|L[A]| = n$  (máximo).

$\Rightarrow$  A estuvo en  $k$  comparaciones, con  $2^k \geq |L[A]| = n$

$\Rightarrow k \geq \lceil \log_2(n) \rceil$

• # de comparaciones  $\geq n - 2 + \lceil \log_2(n) \rceil$

2015年9月15日 (K)

## Cotas inferiores - Reducción

Ej: Colas de prioridad

heapify	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	-
insert	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
extract min	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(1)$
	Heap binario	Fibonacci Heap	?	?
			no existe	

Simplemente usando argumentos de reducción, esas 2 alternativas no pueden existir pues en tal caso se podría ordenar en tiempo lineal.

Ej: 3 SUM (puedo deducir complejidades no conocidas)

Dados  $n$  números  $x_1, \dots, x_n$ , ¿puedo elegir  $a, b, c$  tq  $a + b + c = 0$ ?  
 →  $n^3$  fuerza bruta  
 →  $n^2 \log n$  buscar todos los pares y ordenar y buscar.  
 →  $n^2$  buscar  $a + b = 0$ ? →  $\underbrace{\quad\quad\quad}_{\text{tao tao}} \uparrow$   
 para  $a + b + c = 0$ , hacer el algoritmo anterior  $n$  veces para  $-x_1, -x_2, \dots$

Conjetura: 3 SUM es  $\Omega(n^2)$

Pero este año se demostró que 3SUM es  $\Omega(n^{2-\varepsilon})$   $\forall \varepsilon > 0$   
 Granlund & Pettie

$$\frac{n^2}{(\log n / \log \log n)^{2/3}}$$

Y 3SUM se ocupaba para reducir problemas 3SUM-hard  $\circlearrowleft$

Ej: 3 Colinear: Dados  $n$  puntos en el plano, existen 3 sean colineales? (no en vertical)

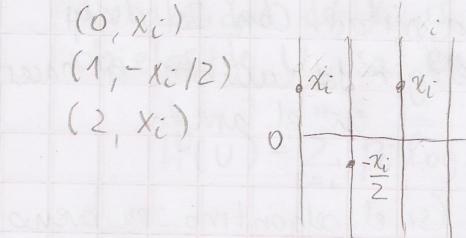
Sea  $x_1, \dots, x_n$  un problema de 3SUM

Por cada  $x_i$  creamos 3 puntos. (transformación en tiempo lineal)

$(0, x_i)$

$(1, -x_i/2)$

$(2, x_i)$



Sean 3 puntos colineales

$(0, a) (1, b) (2, c)$

$a = \frac{a+c}{2}$   $b$  es el punto medio (Thales)

$$\frac{-x_j}{2} = \frac{x_i + x_k}{2} \Rightarrow x_i + x_j + x_k = 0.$$

¶ al resolver 3-colinear soluciono 3SUM.

→ no es lineal, por adversario.

Teoría de la información sirve incluso para encontrar cotas a casos promedios

Ej: concurso de 20 preguntas binarias para adivinar persona.

Ej: buscar en base a comparaciones < en arreglo ordenado es  $\lceil \log_2 n \rceil$  al menos

Ej: sort



Para 2 permutaciones distintas, las secuencias de intercambios deben ser distintas, pues todos los intercambios son la permutación inversa para llegar a la "identidad". Y la inversa es ÚNICA.

Hay  $n!$  permutaciones distintas, luego  $n!$  transformaciones distintas.  
 Pues si hay 2 permutaciones  $\neq$ 's con la misma transformación, una de ellas no llega a la identidad.

El algoritmo (determinista) debe ser capaz de distinguir entre  $n!$  respuestas. Con 1 pregunta, descarta la mitad. Con 2,  $\frac{3}{4}$ ,  
 $\Rightarrow \log_2 n!$  preguntas binarias para elegir los cambios a hacer en el anejo.

$\lceil \log_2 n! \rceil$  preguntas son NECESARIAS (si el algoritmo es bueno siempre va a descartar la mitad del conjunto que me queda) por adversario.

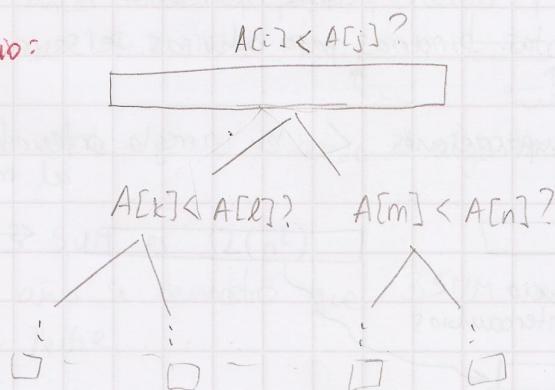
Recordar: STIRLING.

↳ analogía de cortar y elegir la torta.  
 Minimizar el peor caso

$$n! = \frac{n^n}{e^n} \sqrt{2\pi n} \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

$\Rightarrow \lfloor \log n! = n \log n - \Theta(n) \rfloor$  Luego ordenar es al menos  $\Theta(n \log n)$

Caso promedio:



Solo tengo información suficiente para entregar output ordenado en las hojas. Si lo codifico con bits, tengo una combinación ÚNICA para cada permutación (si no es única, el algoritmo entrega el mismo resultado para 2 perm  $\neq$ 's, y una de ellas no estaría ordenada).

## Ajax # 2

Si tengo un algoritmo que hace q preguntas, codifico con q bits.

(Shannon 1948)

Sea  $U$  un conjunto de  $n$  objetos  $U = \{x_1, \dots, x_n\}$  donde la probabilidad de  $x_i$  es  $p_i$ . Entonces cualquier codificación de los objetos de  $U$  usa en promedio, al menos

$$H(U) = \sum_{i=1}^n p_i \log_2 \left( \frac{1}{p_i} \right) \text{ bits. (a los ellos más probables les asigno codificaciones más pequeñas)}$$

En particular, si todos los ellos tienen la misma probabilidad, se necesitan  $\log_2 n$  bits.

Supongamos transmiso una permutación al azar (uniforme entre los  $n!$ )

$$H = \sum_{i=1}^{n!} \frac{1}{n!} \log_2 \left( \frac{1}{1/n!} \right) = \log_2 n! = \Theta(n \log n)$$

Es decir, si uso un algoritmo para codificar permutaciones, cada una con igual prob, necesito en promedio  $n \log n$  bits.

En teoría de la información, en general la cota inferior para el caso promedio es igual a la cota inferior para el peor caso, con distribución uniforme.

En cambio,

$$p_i = \frac{1}{2^{i+1}}$$

$$H = 2 \text{ bits}$$

Cota inferior,  
independiente de la  
forma de codificar

lanzar moneda muchas veces

$$\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}$$

1 01 001 0001

Si al tener un arreglo con ese sesgo y quiero ~~buscar~~, me conviene ordenar por sus  $p_i$  en orden decreciente y hacer búsqueda secuencial.

Si la probabilidad es uniforme, es conveniente usar un ABB.

¿ Y cuando tengo un sesgo entre medio? ☺ ↗ ↘ ↙ !

## Auxiliar 2 - (Más) Cotas Inferiores

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro      Auxiliar: Jorge Bahamonde

21 de Septiembre del 2015

### Adversarios

- Demuestre que determinar si un grafo no dirigido es o no cíclico toma  $\binom{n}{2}$  consultas del tipo “¿existe un arco entre los nodos  $u$  y  $v$ ?”, si la cantidad total de nodos es  $n$ .

### Reducciones

- Demuestre que el problema de determinar si, dados  $n$  puntos sobre tres líneas horizontales en  $y = 0, 1, 2$ , existe una línea no horizontal que pase por tres puntos es 3SUM-hard.
- Demuestre que el problema de, dado un conjunto de rectas en el plano, determinar si existe un cruce de al menos 3 es 3SUM-hard.

### Árboles de decisión

- Se tiene un arreglo  $A$  ordenado de tamaño  $n$ . Muestre una cota inferior sobre el número de comparaciones necesarias para encontrar la posición de un valor  $y$  en  $A$  (o retornar  $-1$  si no se encuentra). Considere que cada comparación puede retornar una de tres posibles respuestas ( $<$ ,  $>$  o  $=$ ). Realice esto para el peor caso y para el caso promedio.
- Se tiene un arreglo  $A$  ordenado de tamaño  $n$ . Suponga que sólo tiene disponible la función  $\text{TEST}(i, j, k, y)$ , que dados  $0 < i < j < k \leq n$  entrega una de las siguientes respuestas:

$$\begin{array}{ll} y < A[i] & A[j] < y < A[k] \\ y = A[i] & y = A[k] \\ A[i] < y < A[j] & y > A[k] \\ y = A[j] & \end{array}$$

Muestre una cota inferior sobre el número de llamados a  $\text{TEST}$  para encontrar la posición de un valor  $y$  en  $A$  (o retornar  $-1$  si no se encuentra).

2015年9月21日(月)

## Aux # 2

### [P1] Adversario.

- Idea: construiremos un adversario que mantiene 2 grafos, C y A.
  - 1) ambos serán consistentes con las respuestas dadas al algoritmo
  - 2) C tiene un ciclo
  - 3) A no tiene ciclo.

Estas props se mantienen ante  $\leq^*$  consultas

Adv:

$$C \leftarrow K_n \quad // \text{clique de } n \text{ nodos}$$
$$A \leftarrow \emptyset \quad // \text{n nodos, } 0 \text{ arcos.}$$

adyacente ( $u, v$ ):

Si  $A + (u, v)$  es acíclico

$$A \leftarrow A + (u, v)$$

return sí

else

$$C \leftarrow C - (u, v)$$

return No.

Veremos varias props de A y C para demostrar que C tiene 1 ciclo

a)  $A \subseteq C$

b)  $C - A$  es el conjunto de los arcos no preguntados

(todo arco preguntado fue agregado a A o fue quitado de C)

c) Si  $A$  es desconexo,  $C$  tiene todos los arcos que conectan las componentes conexas de A

zotmeng (n) word of zotmeng is zotmeng (s)

- Sea un arco de este tipo. Ese arco no puede crear un ciclo en A, luego no puede haber sido quitado de C.

d) C es conexo.

Dem: Por contradicción. Supongamos en un paso se quita el arco e y C queda desconexo. Luego A

- sería cíclico de tener e.

Pero C queda desconexo  $\Rightarrow$  e no es parte de un ciclo en C

Pero por a),  $A \subseteq C$

e sería parte de un ciclo en A

$\Rightarrow \Leftarrow$  luego C es conexo.

e) Si  $A \neq C$ , C tiene un ciclo.

Dem: Por contradicción sea C acíclico.

- Es conexo por d)  $\Rightarrow$  es árbol.

- Como  $A \neq C$  y  $A \subseteq C \Rightarrow A$  es desconexo.

- Sea  $e \in G(C-A)$

- e conecta 2 componentes de A.

- como C es árbol, e debe ser el único que conecta estas 2 componentes.

- Para  $n \geq 3$ , en  $K_n$  existe al menos un  $e'$  que las une,  $e \neq e'$ .

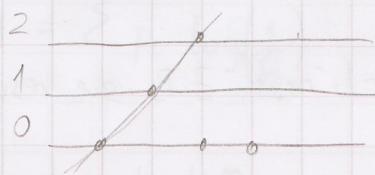
Por b),  $e' \in C-A \Rightarrow \Leftarrow$

$\Rightarrow$  Si  $A \neq C$ , C tiene ciclo.

\* La gracia es que el algoritmo da las mismas respuestas para 2 instancias  $\neq$ 's del problema. El algoritmo no sabe en cuál de los 2 está si no hasta hacer  $\binom{n}{2}$  preguntas.

[P2]

a)



No es trivial modelar

b) Reduciremos 3-colineal a esto. (idea, cambiar palabra "recta" por "punto")

• Tenemos una instancia de 3-colineal: n puntos, buscamos 1 recta tq

$$y_1 = ax_1 + b \quad (x_i, y_i) \rightarrow (a, b)$$

$$y_2 = ax_2 + b$$

$$y_3 = ax_3 + b \quad | \quad |$$

buscamos convertir puntos en líneas y viceversa

• Definimos la i-ésima recta como:  $(a_i, b_i) \rightarrow (x, y)$

$$y = (x_i)x + (-y_i)$$

Supongamos un alg encuentra (o determina que #) solución  $(\bar{x}, \bar{y})$

$(\exists$  pto  $(\bar{x}, \bar{y}))$

$$\text{en 3 rectas} \Leftrightarrow \bar{y}^* = a_1 \bar{x} + b_1$$

$$\bar{y}^* = x_1 \bar{x} - y_1$$

$$\bar{y}^* = x_2 \bar{x} - y_2$$

$$\bar{y}^* = x_3 \bar{x} - y_3$$

$\Leftrightarrow$

$$y_1 = \bar{x} \cdot x_1 - y^*$$

$\Leftrightarrow$

los puntos

$(x_1, y_1)$

$(x_2, y_2)$

$(x_3, y_3)$

son colineales.

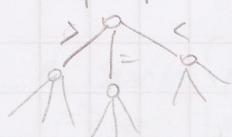
$$y_2 = \bar{x} \cdot x_2 - y^*$$

$$y_3 = \bar{x} \cdot x_3 - y^*$$

\* Hay que verificar que la reducción tome  $\Theta(n^{2-\epsilon})$ ,  $\epsilon > 0$   
 Bueno, el mapeo es lineal ( $y \cdot -1$ )

Luego para cualquier  $A$  que resuelva 3-líneas,  $\exists B$  que resuelve  
 3 colineal, tq:  $\sim$  reducción  
 $T(B) = T(A) + \Theta(n)$

[P3] Todo algoritmo correcto debe tener al menos tantas hojas  
 (a) como resp. posibles.



En este caso, los  $n$  elem. de  $A$  → posiciones  
 deben estar en las hojas.  
 $\Rightarrow$  al menos  $n$  hojas.

- Cada nodo interno produce a lo más 2 nodos internos  
 $\Rightarrow$  hay a lo más  $2^k$  nodos internos en el piso  $k$ .  
 $\Rightarrow$  hay a lo más  $\sum_{i=0}^{k-1} 2^i$  nodos internos en los  $k$  pisos  
 $(= 2^k - 1)$
- Como debe haber al menos  $n$  hojas.

$$2^k - 1 \geq n \Rightarrow k \geq \log_2(n+1) \quad (\text{peor caso})$$

Supongamos que solo busco los elem. que estén  
 $\Rightarrow$  la nta me da un código para el elemento.  
 $A[i] \rightarrow <>><>>$   $\Rightarrow$  puedo usar 1 bit por signo  
 $\Rightarrow k_i$  bits para cada elem

Pero  $H = \sum p_i \log_2 \frac{1}{p_i}$

Suponiendo una entrada uniforme,  $p_i = \frac{1}{n} \Rightarrow H = \log_2 n$   
 $\Rightarrow k_i \geq \log_2 n$  es el caso promedio.

2015年9月22日 (K)

## Búsqueda Binaria con probabilidades de acceso.

$$H = \sum p_i \log\left(\frac{1}{p_i}\right)$$

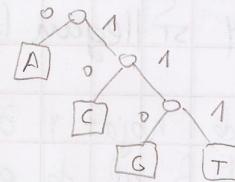
mín. número de bits por símbolo que puede usar una codificación si las prob. de los símbolos son  $p_1, \dots, p_n$

- Algoritmo de Huffman

A	00	0.5
C	01	0.25
G	10	0.125
T	11	0.125



induce a errores. Un código no puede ser prefijo de otro.



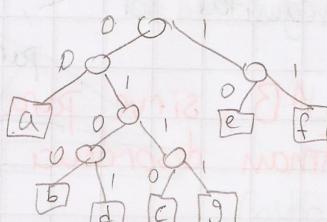
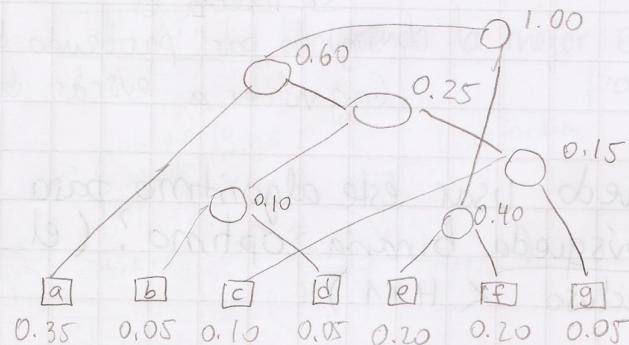
$$H \leq \sum_{i=1}^n p_i l_i < H + 1$$

largo del código

largo promedio de un código  $\left( \left( \sum_{i=1}^n p_i l_i \right) \cdot n = \text{largo total del texto} \right)$

$$\begin{aligned} 00 &= P(a) = 0.35 \\ 0100 &= P(b) = 0.05 \\ 0110 &= P(c) = 0.10 \\ 0101 &= P(d) = 0.05 \\ 10 &= P(e) = 0.20 \\ 11 &= P(f) = 0.20 \\ 0111 &= P(g) = 0.05 \end{aligned}$$

Árbol de peso mínimo



¿En qué tiempo se construye el árbol de peso mínimo?

**Heapify:** armo un heap, luego saco los 2 mínimos, los uno, los sumo, y el nuevo nodo lo reinserto en el heap.

$\Theta(n \log n)$

Otra manera? y si llegan los datos ordenados?

- tener lista de hojas y otra de nodos, e ir agregando los nodos al final de esa lista (ya queda ordenado)
- En una sola lista



se inserta el segundo par partiendo de la última inserción!  
(no volver a buscar del comienzo)

¿Puedo usar este algoritmo para crear un árbol de búsqueda binaria óptimo? (el costo esperado sea mínimo, incluso  $< H+1$ )

¿Qué hago? no puedo preguntar por  $<$ ,  $=$ ,  $>$ .

No es un ABB! es AB, sirve para codificar, pero no para buscar! Huffman desordena las hojas.

Entonces cómo puedo construir el mejor ABB con un algoritmo que no ordene las hojas?

# Memoria Secundaria

- Algoritmo de Hu-Tucker

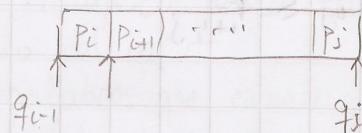
$\Theta(n \log n)$  → encuentra el mejor ABB ( $< H+2$ )  
 $< H+2$  (Garsia-Wachs) (con hojas ordenadas)

## PROBLEMA MÁS GENERAL

$p_1$	$p_2$	$p_3$	$p_4$	$p_5$	$p_6$	$p_7$	
↑	↑	↑	↑	↑	↑	↑	
$q_0$	$q_1$	$q_2$	$q_3$	$q_4$	$q_5$	$q_6$	$q_7$
							3.1

$$\sum p_i + \sum q_j = 1$$

$$P_{ij} = q_{i-1} + p_i + q_i + p_{i+1} + q_{i+1} + \dots + p_j + q_j$$



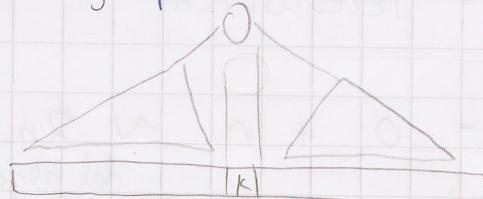
$C_{ij}$  = costo de buscar en  
 (cuantos a accesos)  
 $C_{ii} = 1$



usando la mejor estrategia  
 posible.

$$C_{ij} = 1 + \min_{1 \leq k \leq j} \left( \frac{P_{i,k-1}}{P_{ij}} C_{i,k-1} + \frac{P_{k+1,j}}{P_{ij}} C_{k+1,j} \right)$$

donde poner  
 raíz  $r_{ij}$  óptima



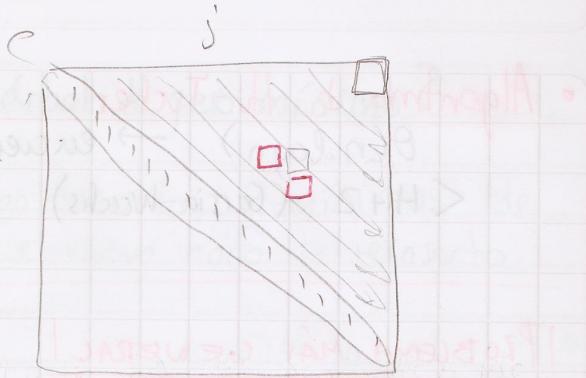
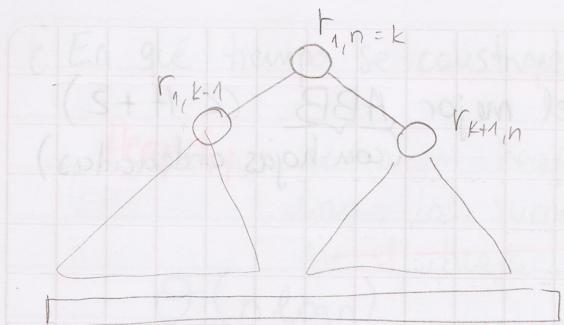
"dado que estoy  
 buscando en  $[i, j]$ "

Luego de hacer todos estos  
 cálculos, sabemos

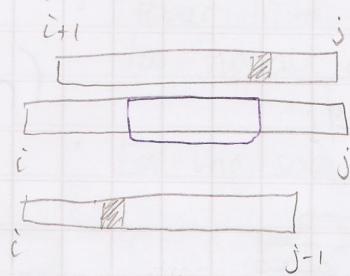
Costo promedio de búsqueda:

$$C_{1,n}$$

¿cuál es el costo de acceso?



¿y  $O(n^2)$ ?



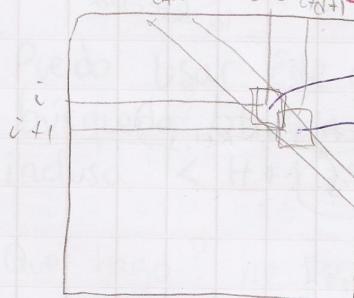
$$r_{i,j-1} \leq r_{ij} \leq r_{i+1,j}$$

con programación dinámica, el trabajo es rellenar esa Matriz

$$\sum_{i=1}^n \sum_{j=i}^n j-i+1 = \frac{n^3}{6} = \Theta(n^3)$$

vas de  $j$  a  $i$  buscando el mínimo

Ahora  $\rightarrow C_{ij} = 1 + \min \left( \frac{P_{i,k-1}}{P_{ij}} C_{i,k-1} + \frac{P_{k+1,j}}{P_{ij}} C_{k+1,j} \right)$



$$r_{i+d, j+d} - r_{i, j+d-1} + 1$$

$$r_{i+2, j+d+1} - r_{i+1, j+d} + 1$$

! TELESCÓPICA

$$r_{i+1,j} - r_{i,j-1} + 1$$

Sobrevive algún  $r \leq n$

$- 0 + n \sim 2n$  costo del algoritmo

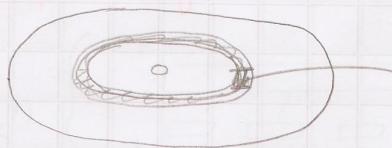
por diagonal

y son  $n$  diagonales  
 $\Rightarrow O(n^2)$

Y si en vez de probabilidades de acceso tengo COSTO de acceso?

# Memonia Secundaria

2015年9月24日(木)



tiempo:

- seek → mover puntero a pista
- latencia → tiempo para que el bloque pase de bajo del cabezal.
- transferencia

Para algoritmos en mem secundaria:

- minimizar los accesos a disco
- aprovechar localidad de bloques y entre bloques  
(pasadas secuenciales)

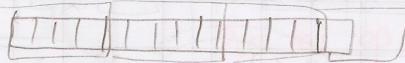
I/O model: costo = # bloques leídos y escritos (no toma en cuenta que un alg. secuencial es mejor que uno salta. Tampoco toma en cuenta trabajo en CPU.)

- n = tamaño del input
- B = n de palabras que caben en un bloque de mem secundaria
- M = n de palabras que caben en RAM.

El costo en I/O model se puede definir/medir en fn. de n, B y M.

Ej: Si leo todos los n enteros de un arreglo en disco, el costo es,  
 $\Theta(\frac{n}{B})$  secuencial

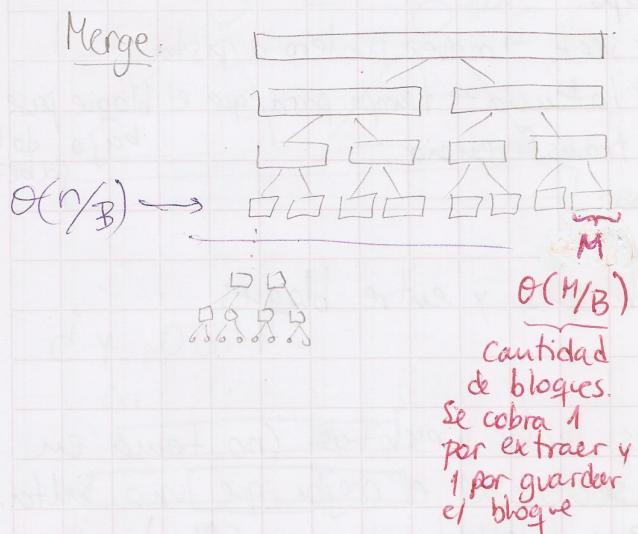
$\Theta(n)$  al azar.



Propri Resposta x3

## Ordenar en Disco:

Merge:



$\log n$

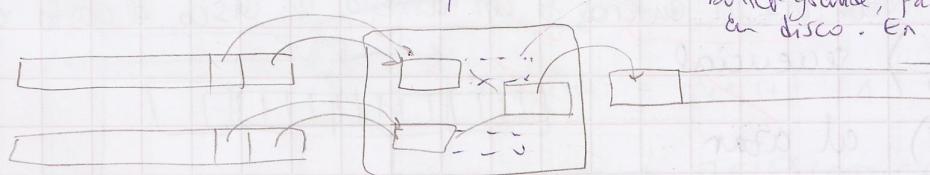
$\Theta(n \log n)$  en disco? horrible.

El algoritmo deja de dividir hasta que los pedazos son de tamaño  $M$ . Luego, en ese piso se hacen  $\Theta(\frac{n}{B})$  accesos.

Costo total:  $\Theta\left(\frac{n}{B} \cdot \log \frac{n}{M}\right)$

pisos.

en la vida real es mejor tener un buffer grande, para leer secuencialmente en disco. En el modelo I/O da lo mismo



RAM  
el merge es  
asíncrono, luego  
se lleva el bloque  
de at/put en  
cualquier momento.

¿se puede hacer mejor?

¿Por qué merge divide de a 2? ¿Si dividimos de a más?



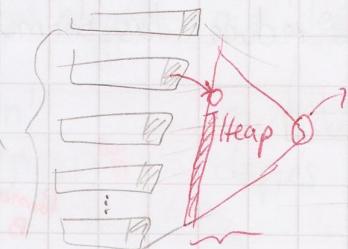
$$\Theta(n(\log_k n)(k))$$

esencialmente para elegir min k

$$= \Theta(n(\log_k n) \log_2 k)$$

$$= \Theta(n \log_2 n)$$

:  
no es mejor...



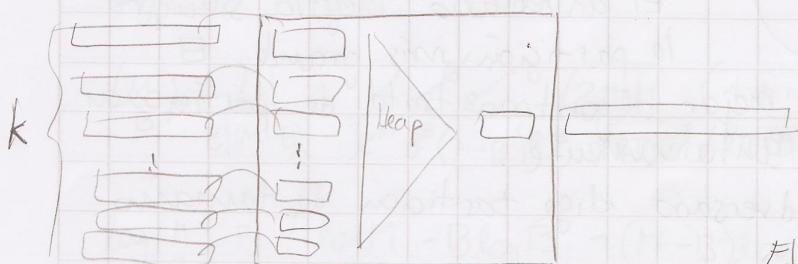
de tamaño k

Se extrae el min y se inserta elto al heap del mismo arreglo del cual se extrajo min  
 $\Rightarrow \Theta(\log_2 k)$

¿Cómo extraigo el mínimo de k? eficientemente?

$\rightarrow$  Usar heap

Sirve este merge para el modelo I/O de ordenamiento en disco?



$$\Rightarrow \Theta\left(\frac{n}{B} \log\left(\frac{n}{M}\right)\right)$$

Pues las op. de heap se hacen en memoria

$\Rightarrow$  costo  $\Theta$ .

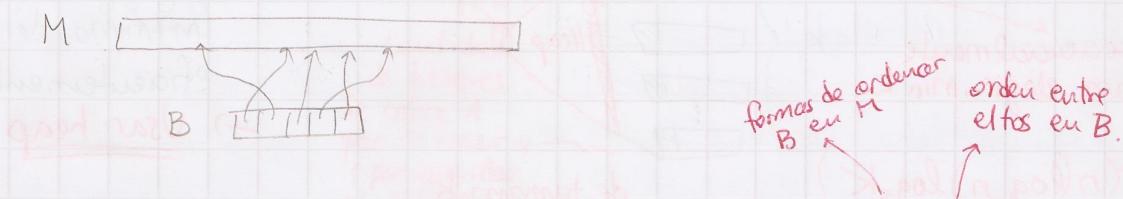
El buen k es  $\leq \frac{M}{B}$  (lo máximo de bloques que caben en memoria)

$$\Rightarrow \boxed{\Theta\left(\frac{n}{B} \log_{\frac{M}{B}}\left(\frac{n}{M}\right)\right)}$$

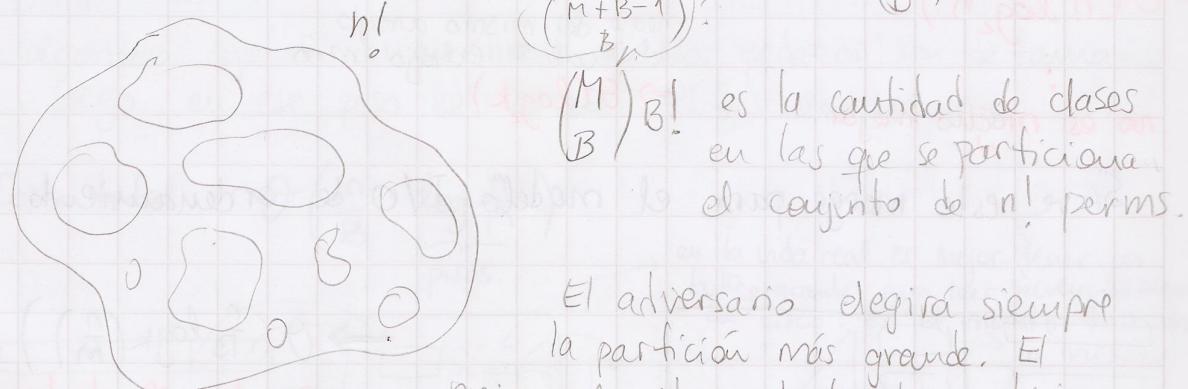
Pero ese k óptimo en la vida real no lo es tanto, pues conviene tener buffers más grandes.

- Cota inferior para ordenar en disco
- Recordar que ordenar lo podemos interpretar como buscar la única permutación.

- Inicialmente, todas las  $n!$  permutaciones son posibles.
- El algoritmo lee el input y va descartando permutaciones.
- Seo puede terminar cuando reduce las permutaciones a 1.



En este procedimiento, el proceso puede resultar en  $\binom{M}{B} B!$  casos.



El adversario elegirá siempre la partición más grande. El mejor algoritmo trata de particionar equitativamente.

En cualquier caso, el adversario elige partición de tamaño al menos  $\frac{n!}{\binom{M}{B} B!}$ .

$$\frac{n!}{\binom{M}{B} B!}$$

$$\frac{(n!)^B}{(\binom{M}{B})^B B!}$$

## Aux #3

2023-9-1 288

Luego de  $t$  lecturas de bloques, el tamaño del conjunto es

$$\frac{n!}{(M)^t (B!)^t} \quad \text{y de ahí es fácil sacar } t.$$

Pero somos generosos.  $t \geq \frac{n!}{B}$  para leer todo.

En tal caso, hay bloques que leemos más de una vez.

Ahora  $\frac{n!}{(M)^t (B!)^t \frac{n!}{B}}$  cantidad de permutaciones admisibles después de leer  $t$  veces.

$$\frac{n!}{(M)^t (B!)^t \frac{n!}{B}} \leq 1 \Rightarrow \frac{n!}{(B!)^{\frac{n!}{B}}} \leq \left(\frac{M}{B}\right)^t$$

$$\Rightarrow \log n! - \left(\frac{n!}{B}\right) \log B! \leq t \log \left(\frac{M}{B}\right)$$

$$n! = \frac{n^n}{e^n} \sqrt{2\pi n} \left(1 + O\left(\frac{1}{n}\right)\right)$$

$$\log n! = n \log n - O(n)$$

$$\binom{M}{B} = \frac{M!}{B!(M-B)!} = \frac{M^M e^{B(M-B)}}{e^M B^B (M-B)^{M-B}} \frac{\sqrt{2\pi M}}{\sqrt{2\pi B} \sqrt{2\pi (M-B)}} (1 + \dots)$$

$$\log \binom{M}{B} = M \log M - B \log B - (M-B) \log (M-B) + O(\log M)$$

$$B \log M + (M-B) \log M$$

$$= B \log \frac{M}{B} + (M-B) \log \frac{M}{M-B} + O(\log M) = B \log \frac{M}{B} + O(B + \log M)$$

$$\ln M + x \leq x$$

$$\log \frac{M-B+B}{M-B} = \log 1 + \frac{B}{M-B} = O\left(\frac{B}{M-B}\right)$$

$$O\left(B \log \frac{M}{B}\right)$$

Luego  $n \log n - n \log B \leq t$

$$B \log \frac{M}{B}$$

$$\Rightarrow \frac{n}{B} \log \frac{M}{B} \left( \frac{n}{B} \right) \leq t.$$

Cota Superior  $\frac{n}{B} \log \frac{M}{B} \left( \frac{n}{B} \right) = \Omega \left( \frac{n}{B} \log \frac{M}{B} n \right)$

↳ no es tan temible.

$$M > n^\alpha \quad \forall \alpha < 1$$

M tendría que ser demasiado grande para cambiar el orden de magnitud.

$$\frac{n}{B} \log \frac{n}{B} \left( \frac{n}{B} \right)$$

Tan grande que no tendría sentido práctico.

## Auxiliar 3 - Memoria Externa

CC4102 - Diseño y Análisis de Algoritmos

Profesor: Gonzalo Navarro      Auxiliar: Jorge Bahamonde

28 de Septiembre del 2015

1. Dados dos arreglos  $A[1, N]$  y  $B[1, N]$  de enteros que no caben en memoria principal, se desea construir (también en disco) el arreglo  $C[i] = A[B[i]]$ . Diseñe un algoritmo eficiente de memoria externa para construir C y analícelo.
2. Diseñe un algoritmo eficiente para memoria secundaria que, dada una relación binaria  $R$  sobre el dominio  $\{1\dots n\}$ , determine si  $R$  es simétrica. Suponga que la relación  $R$  se presenta en disco como un archivo secuencial de todos sus pares  $(i, j) \in \{1\dots n\} \times \{1\dots n\}$ .
3. Considere un arreglo de  $N$  elementos que no caben en memoria principal. Se desea encontrar un elemento que sea mayoría (es decir, que aparezca más de  $\frac{N}{2}$  veces) y reportar su frecuencia. Si no existe tal elemento, se debe reportar **no**. Se dispone de una memoria de tamaño  $M = \Theta(1)$ . Diseñe un algoritmo eficiente que resuelva este problema.

**Propuesto:** Extender este problema a encontrar  $k$  elementos que aparezcan más de  $\frac{N}{k+1}$  veces

## Propuestirijillos

1. Dado un grafo dirigido  $G = (V, E)$ , con  $|V| = n$  y  $|E| = e$ , ambos mucho mayores que  $M$ . El grafo es un archivo secuencial de pares  $(u, v)$  en disco, donde se listan las aristas. Se desconoce  $n$  y  $e$ . Considere  $V = \{1...n\}$  para simplificar. Se pide construir y analizar algoritmos eficientes en el modelo de memoria secundaria para:

- Calcular el grado interior y exterior (número de arcos entrantes y salientes) de todos los nodos.
- Calcular el cuadrado del grafo, definido como  $G^2 = (V, E')$ , donde:

$$E' = \{(u, v) \mid u, v \in E \wedge \exists w \in V, (u, w) \wedge (w, v) \in E\}$$

Puede suponer, para este caso, que  $M \geq n$ , pero no que  $M \geq e$ .

- **Bonus track:** calcule cualquier potencia del grafo. Proponga un algoritmo eficiente en memoria y analícelo. No es aceptable una solución que cueste  $k - 1$  veces o más que lo que obtuvo en el punto anterior.

2. Considere dos matrices esparsas de  $N \times N$ , almacenadas en disco en forma de una secuencia  $(i, j, v)$  para cada valor  $A(i, j) = v \neq 0$ . Se requiere obtener el producto de las dos matrices, pero se dispone solamente de una RAM de tamaño  $M = \Theta(N)$ . Diseñe un algoritmo eficiente en términos de  $p$  y  $p'$ , las cantidades de celdas no cero en las dos matrices. (Hint: se puede conseguir básicamente  $O(pp')/(MB))$ ).

## AVX #3

2015年9月28日 (月)

Memoria externa ~~entra 2 → N en bloques de~~ ~~log<sub>m</sub>n~~

$$\text{Ordenar: } \Theta(N \log N) \rightarrow \Theta\left(\frac{N}{B} \log \frac{\frac{N}{B}}{B}\right) = \Theta(n \log_m n)$$

N: n° de elementos

B: unidad (de bloque) de I/O al disco

M: tamaño de la memoria

$$\frac{N}{B} = n, \frac{M}{B} = m$$

[P1]

$$A[1..N] \Rightarrow C[i] = A[B[i]] \quad i=1..N$$

contienen  
todos los números  
 $\in \{1..N\}$

$\Theta(N)$  no queremos!

queremos que gire en  
función de n y m

involucra bloques.

\* Sea un arreglo  $X[1..N]$

• Al ordenar, obtengo  $[1..N]$

• Si considero a  $X$  como una permutación  $\phi_X$ , ordenar  
aplica  $\phi_X^{-1}$

• Algoritmo:

• Copiar A a  $A'$  tq  $A'[i] = (\bar{i}, A[i])$   $\// \Theta\left(\frac{N}{B}\right) = \Theta(n)$

• Ordeno  $A'$  por el segundo argumento  $\Rightarrow (\phi_A^{-1}, \bar{i})$   $\// \Theta(n \log_m n)$

• Copio B en el 2º argumento  $\Rightarrow (\phi_A^{-1}, \phi_B)$   $\// \Theta(n)$

• Ordeno  $A'$  por el primer argumento  $\Rightarrow (\bar{i}, \phi_A^{-1}, \phi_B)$   $\// \Theta(n \log_m n)$

• Copio el 2º argumento en C

$$\Rightarrow \underbrace{\phi_A^{-1} \circ \phi_B}_{A[B[i]]} \quad // \Theta(n)$$

Tiempo total:  $\Theta(n \log_m n) \ll \Theta(n)$

(B) BÁSICAS PPTOS

## E # X A

Hacer dibujo!! pptos, revisar si  $A \leftrightarrow B$  estuvieron bien.

$$A = [2 \ 4 \ 1 \ 5 \ 3] = (-1) \quad A' = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 1 & 5 & 3 \end{bmatrix}$$

$$B = [3 \ 5 \ 2 \ 1 \ 4]$$

$$\Rightarrow C = [1 \ 3 \ 4 \ 2 \ 5] \quad 2) \quad A' = \begin{bmatrix} 3 & 1 & 5 & 2 & 4 \\ 1 & 2 & 3 & 4 & 5 \end{bmatrix}$$

$$3) \quad A' = \begin{bmatrix} 3 & 1 & 5 & 2 & 4 \\ 3 & 5 & 2 & 1 & 4 \end{bmatrix} \quad 4) \quad A' = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 1 & 3 & 4 & 2 \end{bmatrix}$$

5) Esto es  $B[A[C]]$

P

$$(n) \circ = \begin{pmatrix} n \\ 0 \end{pmatrix} \quad (1) \circ A = E_1 A \text{ de } 1 \times n \text{ o } A \text{ nula}$$

$$(n \circ) \circ B = (1 \circ \circ B) \leftarrow \text{otro modo de escribir } B \text{ de } 1 \times n \text{ o } A \text{ nula}$$

$$(n) \circ \rightarrow \text{otro modo de escribir } B \text{ de } 1 \times n \text{ o } A \text{ nula}$$

$$(n \circ) \circ \rightarrow (1 \circ \circ B) \leftarrow \text{otro modo de escribir } B \text{ de } 1 \times n \text{ o } A \text{ nula}$$

$$(n) \circ \rightarrow (1 \circ \circ B) \leftarrow \text{otro modo de escribir } B \text{ de } 1 \times n \text{ o } A \text{ nula}$$

$$(n) \circ \rightarrow (1 \circ \circ B) \leftarrow \text{otro modo de escribir } B \text{ de } 1 \times n \text{ o } A \text{ nula}$$

P2

## Relación Binaria R

$$(R \subseteq \{1..n\} \times \{1..n\})$$

$$\left\{ \left[ \underbrace{(a_1, b_1)}, \underbrace{(a_2, b_2)} \dots \right] \right.$$

$$|R| \gg M$$

Simetria: " $(\underline{a}, \underline{b}) \in R \Leftrightarrow (\underline{b}, \underline{a})' \in R'$ "

- $\Rightarrow$  • Hacer copia de R:  $R'$

  - Ordenar R por el primer argumento  $\rightarrow$  y luego por el 2º
  - Ordenar  $R'$  por el segundo  $\rightarrow$  luego por el 1º
  - Comparar secuencialmente (invirtiendo los pares de  $R'$ )
  - Si una comp. falla  $\Rightarrow$  no es simétrica.

P3

$A$ ,  $|A|=N$ ,  $A = a_1, a_2, \dots, a_n$

Mayoria ( $a_1, \dots, a_n$ )

$X \leftarrow a_1$

C ← 1

for  $y$  in  $a_2 \dots a_n$ :

$$- \text{ if } x =$$

$c \leftarrow c + 1$

else  $c \leftarrow c - 1$

if  $c = \emptyset$

~~C ← 1~~

$x \leftarrow y$

100% 100%

Elm X.

return x;

Dem: por contradicción.

Supongamos que  $\exists k$  con  $k > \underline{N}$  aparciales

Si  $a_k \neq x$ , todas las instancias de  $a_k$  deben haber sido "quemadas" por otros elementos.

no necesariamente el mismo en todos los casos.

$$\Rightarrow \text{hay } > \frac{N}{2} \text{ } a_i's \neq a_k \Rightarrow \leftarrow$$

Llego Mayónia retorna el  $a_k$  correcto si  $\exists$ ; si no

$\Rightarrow$  Verificar  $(x, a_1, \dots, a_N)$

Compara cada  $a_i$  con  $x$  y cuenta el # de aciertos

$\Rightarrow$  si son más de  $\frac{N}{2}$   $\rightarrow$  retorna el conte.

$\Rightarrow$  else retorna "No"

\* ojo con el falso positivo del contador, en casos extremos

PPTOS :

para 1 lado y otro

[P1] a) ordenar y contar

b) pares de nodos en los que existe un camino de largo ?.

c)

2015年9月29日 (K)

## Colas de prioridad en memoria secundaria

### OPERACIONES:

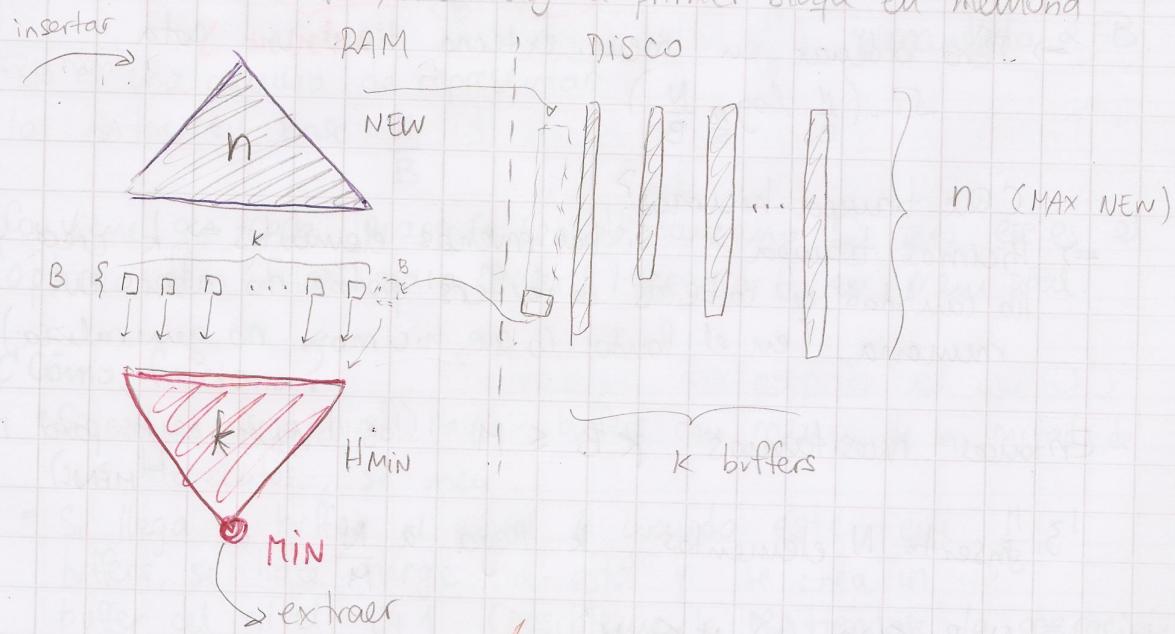
- inicializar
- insertar elementos
- extraer mínimo.

En mem principal, tenemos el HEAP BINARIO

Idea: tener un heap en memoria principal

• Si se llena, ordeno sus elementos y los escribo a disco  
en un arreglo: "buffer"

• De cada buffer, monto el primer bloque en memoria



• En otro heap en memoria se almacenan estos bloques

(Ojo al insertar, comparar con el mínimo del H<sub>MIN</sub>)

Análisis:

Cuánto Paga un elemento al insertarlo y eventualmente extraerlo?

(en I/O)

- $\Theta(1/B)$
- entra a NEW  $\rightarrow \emptyset$
  - viaja a un buffer en Disco  $\rightarrow 1/B$  (promediado cuando entran B elementos a la vez)
  - Pasa a un minibloque  $\rightarrow 1/B$
  - llega a  $H_{\min}$   $\rightarrow \emptyset$
  - Sale  $\rightarrow \emptyset$
- no se pondera por  $M$  (tamaño buffer)  
ni pres en el modelo no se considera secuencialidad.

Cuál pasa si inserto  $N$  elementos y dp los saco todos?

$$\Rightarrow \Theta(1/B) \cdot N = \Theta(N/B), \text{ pero salen } \underline{\text{en orden}}$$

$\Rightarrow$  Pero ordenar en mem. externa tiene una cota

$$\Omega\left(\frac{N}{B} \log_M \frac{N}{B}\right)$$

Cuál trampa hicimos?

$\Rightarrow$  Hicimos trampa : Si hay muchos elementos  $\Rightarrow k$  crece y la cantidad de cabezas de buffers podría no cabrer en memoria. (en el fondo, lo que hicimos no generaliza)

Entonces necesitamos:  $kB \leq M$  (los bloques usados para rellenar  $H_{\min}$ )

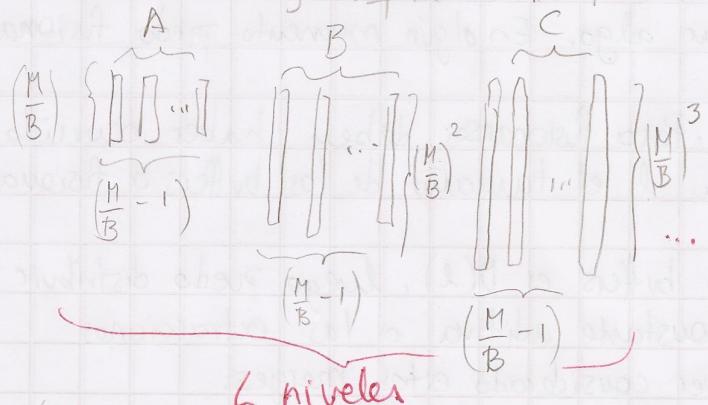
Si inserto  $N$  elementos,  $k$  llega a  $\frac{N}{M}$

$$\Rightarrow \frac{N \cdot B}{M} \leq M \Rightarrow N \leq \frac{M^2}{B}$$

\* En este caso  $\log_M \frac{N}{B} \approx 1$ , luego  $\Theta\left(\frac{N}{B} \log_M \frac{N}{B}\right) \approx \Theta\left(\frac{N}{B}\right)$

¿Qué hacemos si  $N > \frac{M^2}{B}$ ? es decir, los bloques para  $HMIN$  no caben en mem. ppal.

La idea es limitar el número de cabezas de buffers en mem. ppal.  
Haremos una jerarquía de buffers.



Esto es una manera de representar los datos en base  $\frac{M}{B}$

Por ejemplo, si llega un  $\frac{M}{B}$ -ésimo buffer a A,  
 $\frac{M}{B}$  se hace merge de todos los buffers para crear un nuevo buffer de tamaño  $(\frac{M}{B})^2$  y mandarlo a B.

Conviene que cada jerarquía sea de tamaño  $\frac{M}{B}$  pues ese es el óptimo de "filas" para hacer Mergesort en mem. ppal.

¿Cómo funciona?

- Si en un mismo nivel hay 2 buffers con menos de la mitad de sus elementos, se une.
- Si llega un buffer al nivel  $i$  cuando éste tiene  $\frac{M}{B} - 1$  buffers, se hace merge de éstos y se crea un buffer del nivel  $i+1$  (posiblemente) repitiéndose la operación.

¿Cuántos niveles hay?

El max número de niveles es tal que

$$\frac{N}{B} = \Theta\left(\left(\frac{M}{B}\right)^L\right) \Rightarrow L = \Theta\left(\log_{\frac{M}{B}} \frac{N}{B}\right)$$

el IPC del elemento



Evaluemos el costo de la vida de un elemento.

- Supongamos que pasa por todos los niveles

- Entrar al 1<sup>er</sup> nivel:  $1/B$

- Pasar del  $i \rightarrow i+1$ :  $2/B \Rightarrow \Theta\left(\frac{L}{B}\right)$  (leer, merge en mem ppal, escribir)

- Salir a mem ppal:  $1/B$

Wait! Nos faltó contar algo. En algún momento puedo fusionar buffers semivacíos.

★ Los buffers hacen llenos. Para fusionarse, deben haber ocurrido  $\frac{l}{2} + \frac{l}{2}$  extracciones (con  $l$  el tamaño de los buffers a fusionar)

Hacer el merge de los buffers es  $\Theta(l)$ , luego puedo distribuir este costo como una constante aditiva a las extracciones  
 $\Rightarrow$  "está bien" no haber considerado estos merges.

- Insertemos y luego extraigamos  $N$  elementos

$$\Theta\left(\frac{NL}{B}\right) = \Theta\left(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}\right)$$

es  $L$ !

¿Caben los bloques en memoria?

## Colas de prioridad (Pepaso)

04/Octubre/2015

- insertar
- Extraer mínimo



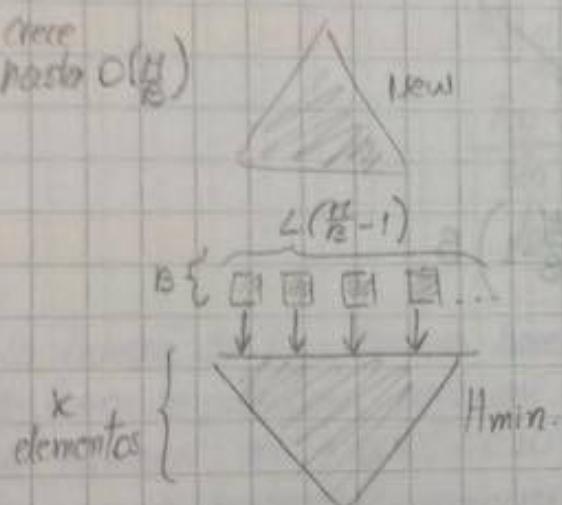
heaps binarios

■ = arreglos  
(ordenados)

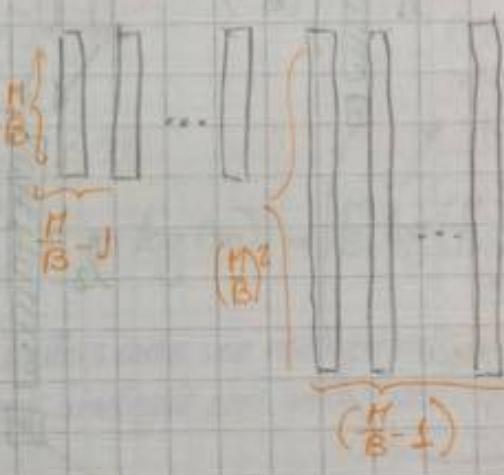
□ = bloques

Memoria

Algoritmo  $O\left(\frac{H}{B}\right)$



Disco

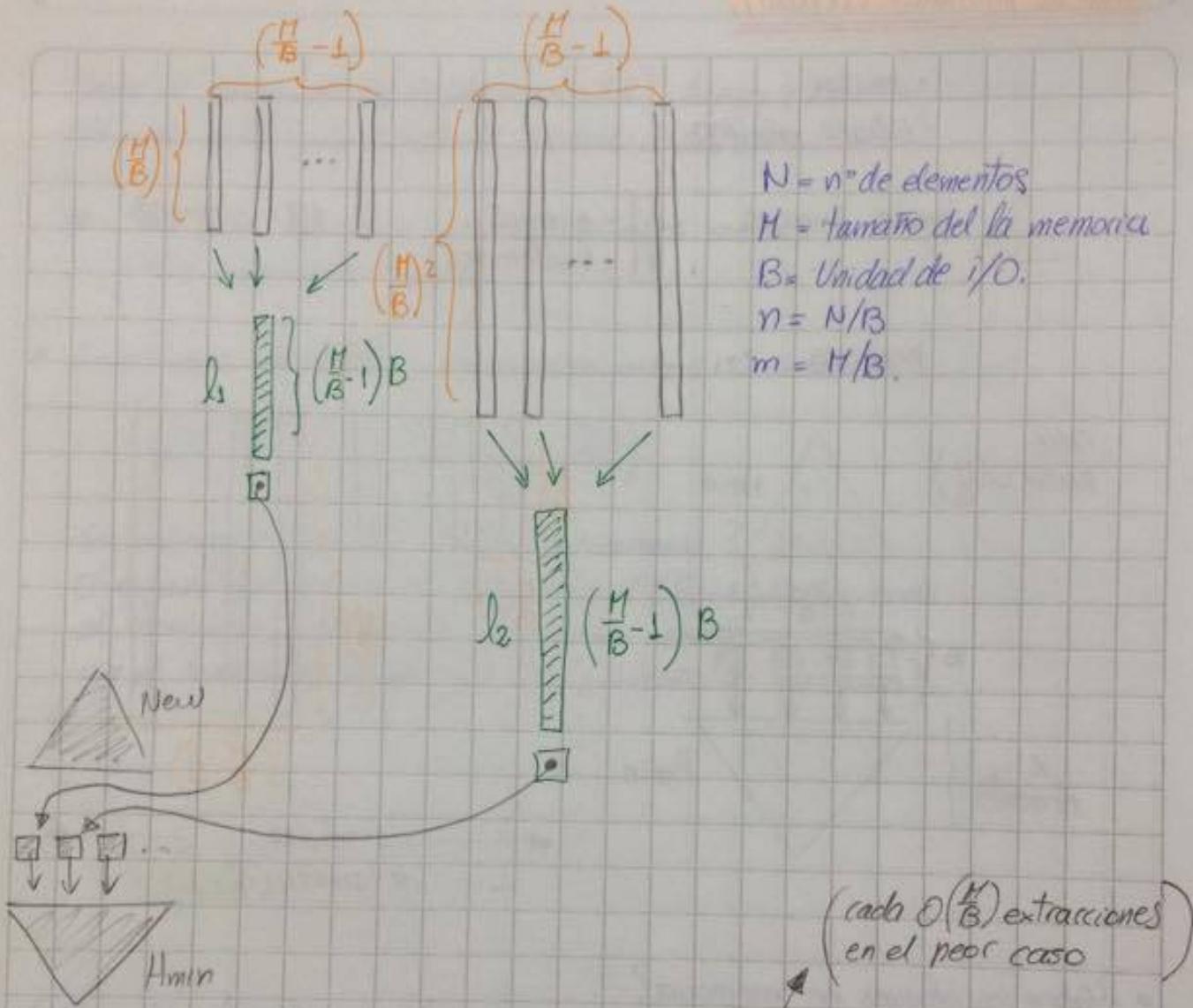


$L \cdot \left(\frac{H}{B} - 1\right)$   
niveles  
de tamaño  $\left(\frac{H}{B} - 1\right)$

- ¿Labren los bloques en memoria?  
Espacio ocupado por los bloques

$$\begin{aligned} L \cdot \left(\frac{H}{B} - 1\right) B &= \left(\frac{H}{B} - 1\right) B \log_{\frac{H}{B}} \left(\frac{H}{B}\right) \\ &= O(H \log_{\frac{H}{B}} \left(\frac{H}{B}\right)) \end{aligned}$$

la idea es aprovechar los niveles que creamos.  
Idea: para cada nivel, guardaremos sólo 1 bloque en memoria.



En disco, agregamos 1 arreglo (buffer) ordenado  $l_i$ , para cada nivel  $i$

→ Cada  $l_i$  contiene el merge de tamaño  $(\frac{H}{B} - 1) \cdot B$  de los buffers de nivel  $i$ .

- Al heap Min lo alimentan las cabezas de  $l_i + i$
- Hay que reciclar  $l_i$  si se vacía (tiempo lineal  $O\left(\frac{H}{B}\right)$ )
- Al crear un nuevo buffer en el nivel  $i$ , hay que hacer un merge parcial con  $l_i$ .

Entonces, para el espacio ocupado por los bloques:

$$LB = B \log_B \left( \frac{N}{B} \right) \leq M \quad \text{imponemos/ necesitamos...}$$

$$B \left( \log_B \left( \frac{N}{B} \right) + \log_B (B) \right) \leq B \cdot m \quad n \cdot \frac{M}{B} \Rightarrow N = n \cdot B$$

$$\log_B (n) + \log_B (B) - \log_B (B) \leq m$$

$$\log_B (n) \leq m$$

El primer intento estaba

$$\log(n) \leq m$$

$$N \leq \frac{M^2}{B}$$

$$\log(n) \leq m \log(m) - \frac{M}{B} \log\left(\frac{M}{B}\right)$$

\* La "trampa" del logaritmo es hacer buffers cada vez más grandes, de forma de potencias, para así hacer analogía con una "distribución de números"  $a10 + b10^2 + c10^3 + \dots$

Así uno va haciendo formaciones de tiempo logarítmico, asumiendo que los buffers son ordenados. (Y luego tener una representante por cada familia de buffers y hacer merge de disco.)

# Auxiliar/Clase 4 - B-Trees y algo de R-Trees

CC4102 - Diseño y Análisis de Algoritmos  
Profesor: Gonzalo Navarro      Auxiliar: Jorge Bahamonde

5 de Octubre del 2015

## B-Trees

Un *B – Tree* de orden  $m$  es un árbol que satisface las siguientes propiedades:

- Cada nodo tiene  $\leq m$  hijos.
  - Cada nodo, excepto el nodo raíz y las hojas, tienen  $\geq m/2$  hijos.
  - Todas las hojas se encuentran en el mismo nivel.
  - Un nodo que no es hoja y tiene  $k$  hijos contiene  $k - 1$  llaves.
1. Especifique el algoritmo de inserción en un B-Tree.
  2. Especifique el algoritmo de borrado en un B-Tree.
  3. Especifique el algoritmo de búsqueda en un B-Tree.

## R-Trees

Un *R – Tree* es una estructura de datos jerárquica, basada en una variante de B-Trees que sólo contiene datos en las hojas (a esta variante se le llama B+ -Trees); se les usa para organizar dinámicamente un conjunto de objetos geométricos. Cada nodo de un R-Tree representa el MBR (*minimum bounding rectangle*) que engloba a sus hijos. Los MBRs de diferentes nodos pueden solaparse. Adicionalmente, un R-Tree tiene las siguientes propiedades:

- Cada nodo hoja tiene hasta  $M$  elementos, donde el mínimo número de entradas es  $m \geq M/2$ .
  - El número de entradas que cada nodo interno no-raíz almacena está entre  $m \geq M/2$  y  $M$ .
  - Todas las hojas del R-Tree están en el mismo nivel.
  - El número mínimo de entradas en el nodo raíz es 2, a menos que sea una hoja.
1. Especifique el algoritmo de inserción en un R-Tree.
  2. Especifique el algoritmo de borrado en un R-Tree.
  3. Especifique el algoritmo de búsqueda en un R-Tree.