

2015年9月14日 (月)

AUX # 1

"Cotas inferiores"

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

|P1|

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 ($\text{comparar}(a, b)$) 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:

$(u_1, v_1), (v_1, u_2), (u_2, v_2), (v_2, u_3), \dots, (u_n, v_n)$ } 2^{n-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 2^{n-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 $\boxed{u_{k+1} < v_k}$

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

↳ las respuestas del adversario no

permiten discernir este caso \Rightarrow El algoritmo NO es correcto.

Nota: como existe un par no compartido, el algoritmo debe situar dichos elementos en la lista. Basta que el adversario diga que el orden no era el correcto (El inverso) para que el algoritmo falle
↳ 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

Σ : alfabeto

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

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

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.

$\hookrightarrow \underline{\bar{n}} \underline{i} \underline{z} \underline{a} \underline{\bar{n}} \underline{\bar{n}} \underline{a}$

El adversario tira errores a las 3 consultas del algoritmo.

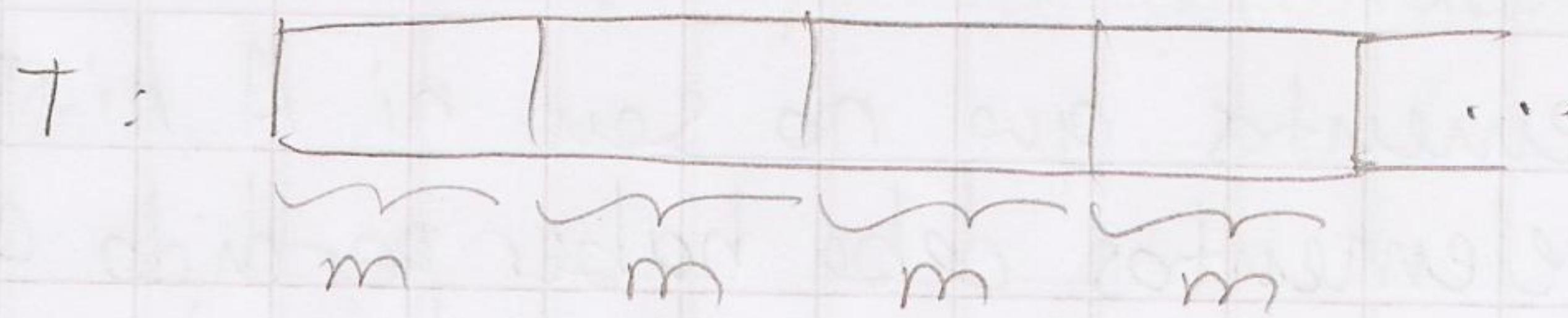
Entonces:

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

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

Reducción

- b) Terceros P inicial : P_i = "encontrar patrones en cualquier lugar" \leftarrow difícil
 Tercero : P_o = "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 $\Omega\left(\frac{kn}{m}\right)$, $k > 0$. para su correctitud.

a)

en realidad es $k+1$, pero la notación Ω 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.

$$\Rightarrow \text{En particular demora al menos el máximo de los 2 tiempos. Además: } (f+g) \in \Omega(\max(f, g)) \Rightarrow \Omega\left(\frac{n \log_2(m)}{m}\right) + \Omega\left(\frac{kn}{m}\right) = \Omega\left(\frac{n(k + \log_2(m))}{m}\right)$$

↳ 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 ese elemento.

$\Rightarrow L[x] \leftarrow \{x\}$, 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 $|L[x]| = h$, 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).

$$\Rightarrow A \text{ estuvo en } k \text{ comparaciones, con } 2^k \geq |L[A]| = n$$
$$\Rightarrow k \geq \lceil \log_2(n) \rceil$$

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

$$0 < 3 \wedge (3^{-5}n) \leq$$

2015年9月15日 (X)

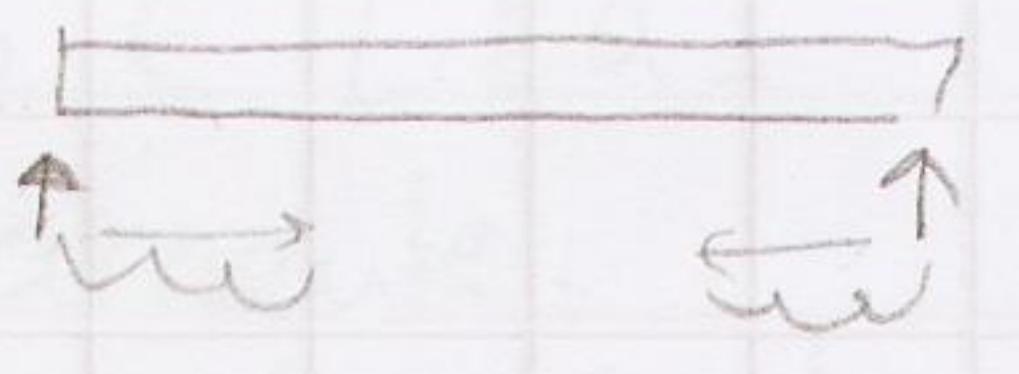
Cotas inferiores - Reducción

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	?	?
<u>no existe</u>				

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 $\text{tg } a + b + c = 0$?
 → n^3 fuerza bruta
 → $n^2 \log n$ buscar todos los pares y ordenar y buscar.
 → n^2 buscar $a + b = 0$? → 

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 $\Omega(n^{2-\varepsilon}) \nexists \varepsilon > 0$

Gronlund & Pettie

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

Y 3SUM se ocupaba para reducir problemas 3SUM-hard ☺

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

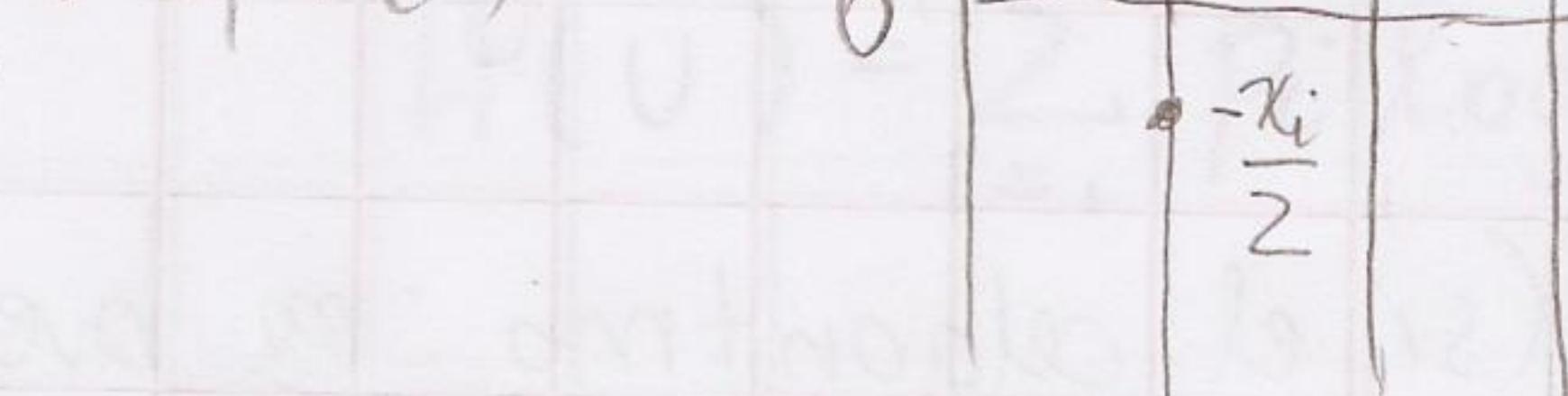
Sea $x_1 \dots x_n$ un problema de 3SUM

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

$(0, x_i)$

$(1, -x_i/2)$

$(2, x_i)$



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

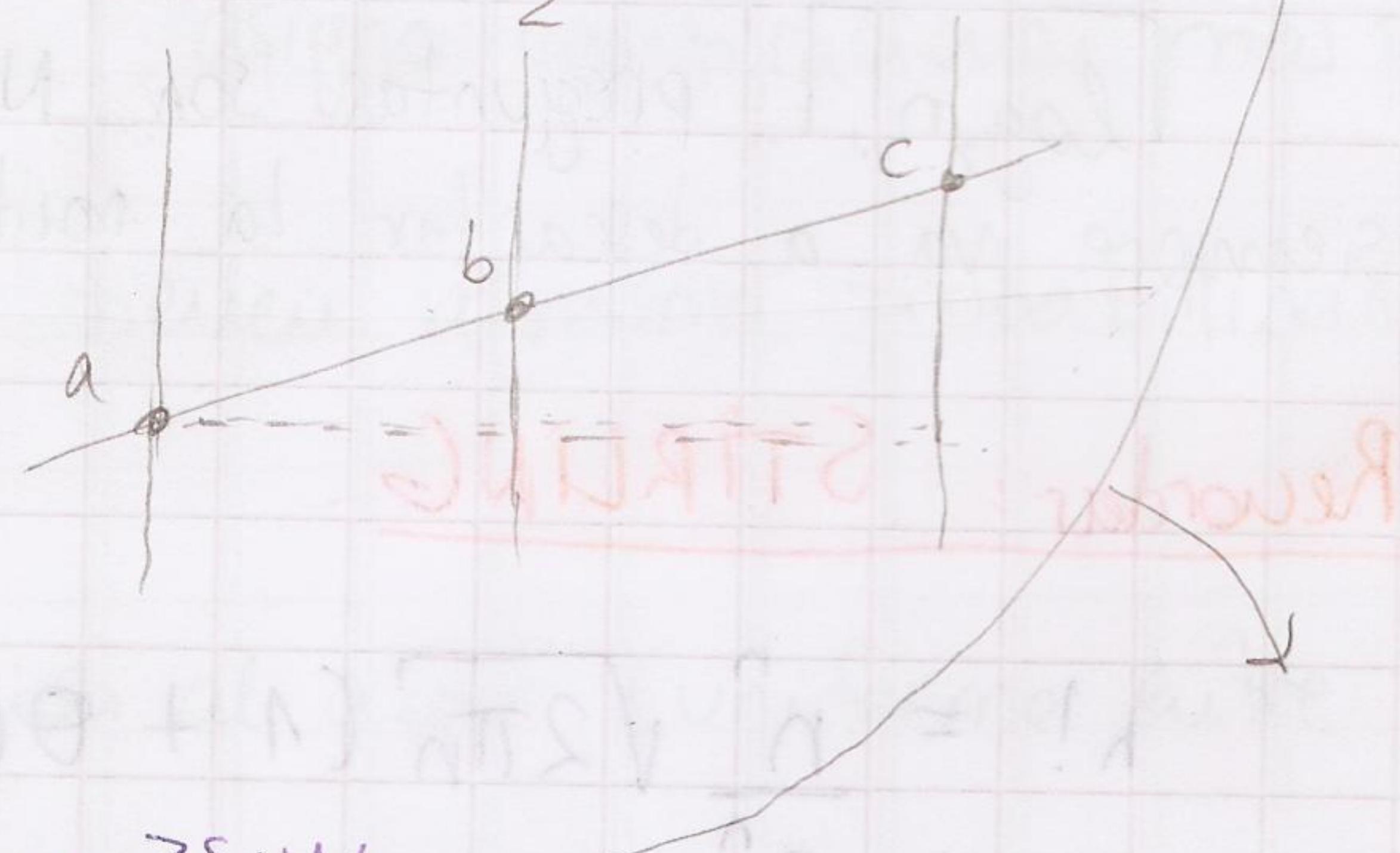
$$-\frac{x_j}{2} = \frac{x_i + x_k}{2} \Rightarrow x_i + x_j + x_k = 0.$$

& al resolver 3-colinear ~~soluciono~~ 3SUM.

Sean 3 puntos colineales

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

$a \quad \frac{-x_j}{2} \quad x_k$



→ no es lineal, por adversario.

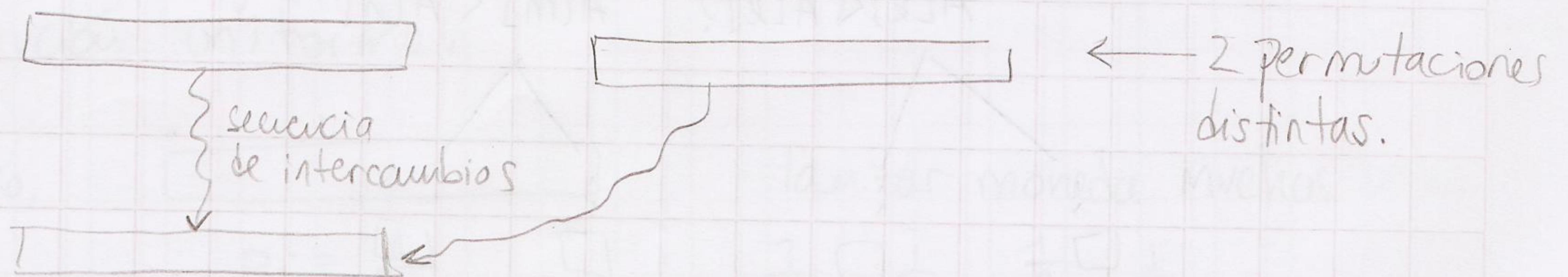
Teoría de la información

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



← 2 permutaciones distintas.

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, $3/4$.

$\Rightarrow \lceil \log_2 n! \rceil$ preguntas binarias para elegir los cambios a hacer en el arreglo.

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

Recordar: STIRLING.

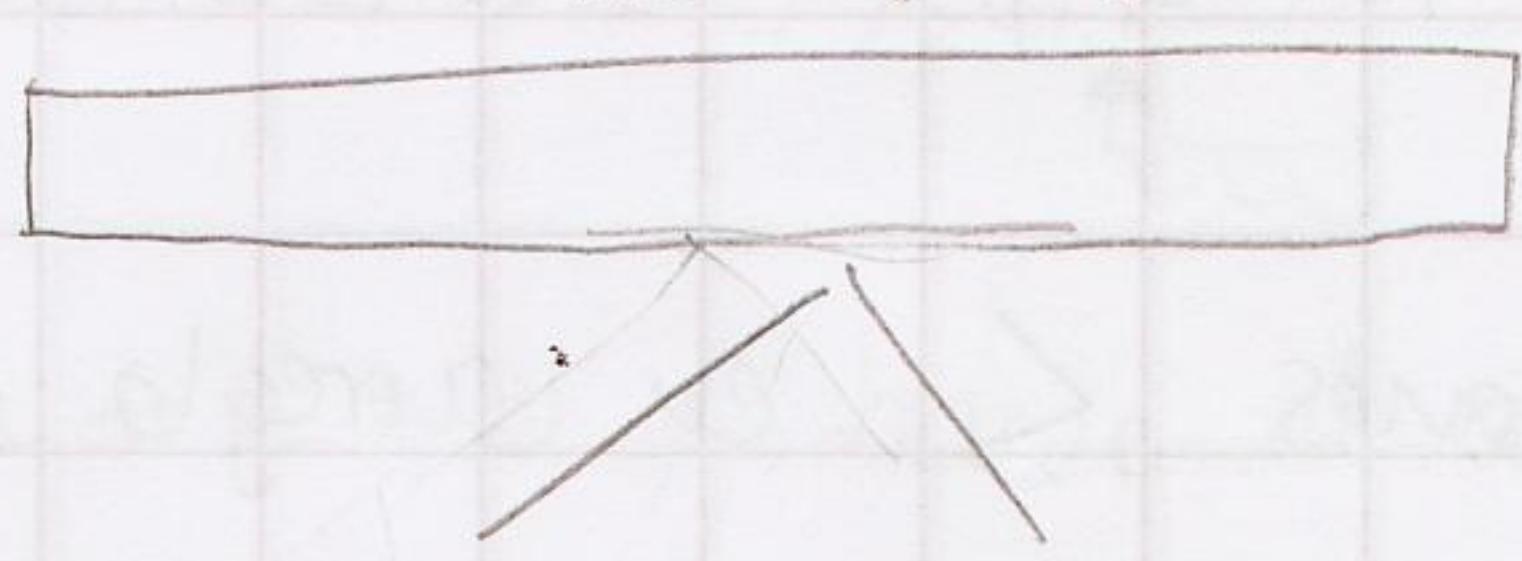
↳ analogía de cortar y elegir la torta.
 Minimizar el peor caso

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

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

Caso promedio:

$A[i] < A[j] ?$



$A[k] < A[l] ? \quad A[m] < A[n] ?$



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).

Aux # 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 objetos de U usa en promedio, al menos

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

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

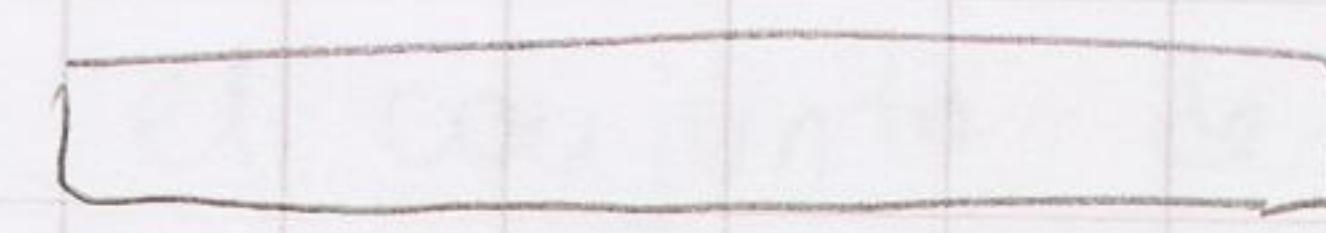
Supongamos transmito una permutación al azar (uniforme entre los $n!$)

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

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,



lanzar moneda muchas veces

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

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

1 01 001 0001

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

cota inferior,

independiente de la forma de codificar

Si al tener un arreglo con ese sesgo y quiero buscarlo, 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? ☺ ↗ ↘ !