

# Modalidad del curso

- Curso dividido en 3 grandes módulos temáticos
- Cada clase estará dividida en dos partes:  
15 min intro/exos + 1h15 + 15 min de pausa + 1h15
- Examen escrito al final de la última clase. Duración 1h

# ¿De qué trata este curso?

**Analysis of Algorithms (AofA)** is a field at the boundary of computer science and mathematics. The goal is to obtain a precise understanding of the *asymptotic, average-case characteristics of algorithms and data structures*. [...]

The area of Analysis of Algorithms is frequently traced to 27 July 1963, when *Donald E. Knuth* wrote “Notes on Open Addressing”.

Del sitio de la comunidad **AofA**

<https://www.math.aau.at/AofA/>



---

Wikipedia, CC BY-SA 3.0.

# Contenido

1. Introducción al análisis probabilístico de algoritmos:
  - Motivación, ejemplos clásicos (sorting, hashing, ...)
  - Modelos modernos (branch prediction).
2. Introducción a la Combinatoria analítica:
  - Funciones generatrices ordinarias y exponenciales.
  - Singularidades, extracción de coeficientes y Teorema de Transferencia.
  - Aplicaciones algorítmicas.
3. Aplicaciones a la generación aleatoria de estructuras discretas<sup>1</sup>:
  - Método recursivo.
  - Boltzmann samplers.

---

<sup>1</sup>Si tiempo.

## 1. Introducción al análisis de algoritmos

- Algoritmos de sorting
- Tablas de Hash

# Introducción: análisis de algoritmos

Estudiar teóricamente la performance de un algoritmo:

- independientemente del lenguaje de programación,
- independientemente del hardware.

# Introducción: análisis de algoritmos

Estudiar teóricamente la performance de un algoritmo:

- independientemente del lenguaje de programación,
- independientemente del hardware.

⇒ contar operaciones concretas efectuadas.

# Introducción: análisis de algoritmos

Estudiar teóricamente la performance de un algoritmo:

- independientemente del lenguaje de programación,
- independientemente del hardware.

⇒ contar operaciones concretas efectuadas.

En los estudios más clásicos:

- Se considera solo el peor caso.
- Solo en orden de magnitud cuando el tamaño del input  $n \rightarrow \infty$ .  
Por ejemplo  $O(n^2)$ ,  $O(n \log n)$ , etc.

# Introducción: análisis de algoritmos

Estudiar teóricamente la performance de un algoritmo:

- independientemente del lenguaje de programación,
- independientemente del hardware.

⇒ contar operaciones concretas efectuadas.

En los estudios más clásicos:

- Se considera solo el peor caso.
- Solo en orden de magnitud cuando el tamaño del input  $n \rightarrow \infty$ .  
Por ejemplo  $O(n^2)$ ,  $O(n \log n)$ , etc.

## Ejemplo

Consideremos el problema de ordenar un array de  $n$  elementos distintos.

Si contamos comparaciones:

1. Mergesort  $\Theta(n \log n)$  en peor caso, Bubble sort y Quicksort  $\Theta(n^2)$ ,
2. pero Quicksort se comporta en  $O(n \log n)$  en media (valor esperado !)



# Nociones básicas de probabilidad

La **media** de una variable aleatoria discreta  $X$  es

$$\mathbb{E}[X] = \sum_{k \in \mathbb{Z}} k \cdot \Pr(X = k),$$

cuando la suma converge absolutamente, es decir  $\mathbb{E}[|X|] < \infty$ .

Recordamos las siguientes propiedades básicas:

- Desigualdad triangular:  $|\mathbb{E}[X]| \leq \mathbb{E}[|X|]$
- La media es lineal  $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$ .
- Para una función indicatriz  $\mathbf{1}_A$  tenemos  $\mathbb{E}[\mathbf{1}_A] = \Pr(A)$ .

# Nociones básicas de probabilidad

La **media** de una variable aleatoria discreta  $X$  es

$$\mathbb{E}[X] = \sum_{k \in \mathbb{Z}} k \cdot \Pr(X = k),$$

cuando la suma converge absolutamente, es decir  $\mathbb{E}[|X|] < \infty$ .

Recordamos las siguientes propiedades básicas:

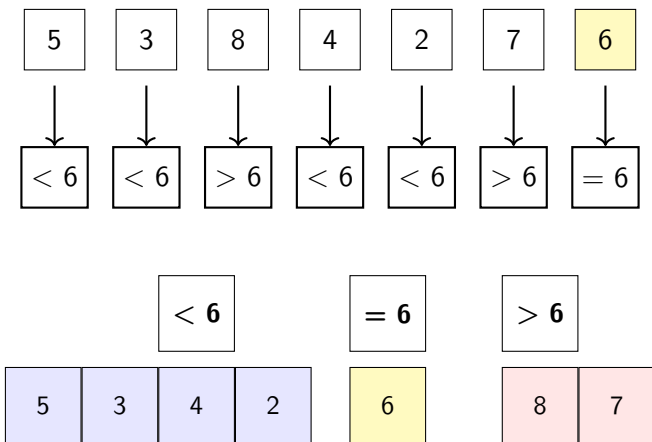
- Desigualdad triangular:  $|\mathbb{E}[X]| \leq \mathbb{E}[|X|]$
- La media es lineal  $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$ .
- Para una función indicatriz  $\mathbf{1}_A$  tenemos  $\mathbb{E}[\mathbf{1}_A] = \Pr(A)$ .

## Fórmula de la probabilidad total

Sean eventos  $S_1, S_2, \dots$  disjuntos con  $\bigcup_i S_i = \Omega$  (todo) :

$$\mathbb{E}[X] = \sum_{k=1}^{\infty} \Pr(S_k) \times \mathbb{E}[X|S_k].$$

# Quicksort



Quicksort particiona según un pivot y luego continua recursivamente.

# Quicksort

Por simplicidad<sup>2</sup> consideramos que el pivot es elegido determinísticamente:

```
def partition(arr, low, high):  
    pivot = arr[high]  
    i = low  
  
    for j in range(low, high):  
        if arr[j] < pivot:  
            arr[i], arr[j] = arr[j], arr[i]  
            i += 1  
  
    arr[i], arr[high] = arr[high], arr[i]  
    return i
```

**Peor caso:** o todos mayores, o todos menores que el pivot:

- Cantidad de comparaciones =  $1 + 2 + \dots + (n - 1) = \frac{n(n-1)}{2} = \Theta(n^2)$ .
- Puede suceder si el array está ya ordenado !

---

<sup>2</sup>Mejor sería un pivot aleatorio, o permutar la entrada para evitar ataques.

# Quicksort: modelo aleatorio

Veamos ahora qué sucede si el array es una permutación aleatoria

- cada permutación  $\pi$  de  $(1, 2, \dots, n)$  tiene probabilidad  $p(\pi) = 1/n!$
- equivalente a elegir  $n$  números aleatorios del intervalo  $[0, 1]$   
 $\implies$  argumento de simetría !

# Quicksort: modelo aleatorio

Veamos ahora qué sucede si el array es una permutación aleatoria

- cada permutación  $\pi$  de  $(1, 2, \dots, n)$  tiene probabilidad  $p(\pi) = 1/n!$
- equivalente a elegir  $n$  números aleatorios del intervalo  $[0, 1]$   
 $\implies$  argumento de simetría !

Nos interesa la cantidad de comparaciones  $C_n(\pi)$  necesarias para ordenar:

- En media  $E_n = \mathbb{E}[C_n] = \sum_{\pi \in \mathcal{S}_n} C_n(\pi) \times p(\pi) = \frac{1}{n!} \sum_{\pi \in \mathcal{S}_n} C_n(\pi)$ ,
- En distribución  $\Pr(C_n > \lambda) = \sum_{\pi \in \mathcal{S}_n: C_n(\pi) > \lambda} p(\pi)$ .

# Quicksort: comportamiento en media

Para la media  $E_n$  de la cantidad de comparaciones  $C_n$  tenemos:

## Proposición

*Quicksort satisface  $E_n = 2(n+1)H_n - 4n$ , donde  $H_n = 1 + \frac{1}{2} + \dots + \frac{1}{n}$  son las sumas armónicas.*

Entonces  $E_n \sim 2n \log n$  donde  $\log$  es el logaritmo natural (neperiano).<sup>3</sup>

---

<sup>3</sup>Las sumas armónicas satisfacen  $H_n \sim \int_1^n \frac{dx}{x} = \log n$ .

# Análisis en media de Quicksort

Con probabilidad  $1/n$  el rango de  $\pi(n)$  es  $j$ , entonces

$$\begin{aligned} E_n &= \sum_{j=1}^n \mathbb{E}[C_n | \text{pivot} = j] \cdot \Pr(\text{pivot} = j), && \text{(prob. total)} \\ &= \frac{1}{n} \times \sum_{j=1}^n \mathbb{E}[C_n | \text{pivot} = j], \\ &= \frac{1}{n} \times \sum_{j=1}^n \mathbb{E}[C_{j-1} + \tilde{C}_{n-j} + n - 1], && (C_{j-1} \text{ y } \tilde{C}_{n-j} \text{ indep.}) \\ &= \frac{1}{n} \times \sum_{j=0}^{n-1} (E_j + E_{n-1-j}) + n - 1. && \text{(linealidad esperanza)} \end{aligned}$$

En la tercera línea  $\tilde{C}_{n-j}$  es el costo de ordenar el array de la parte alta, que contiene  $j+1, \dots, n$  en el orden inicial. Su distribución es la misma que  $C_{n-j}$ .



# Estudio en media

La **media** es una buena medida cuando pensamos ejecutar **muchas veces** un algoritmo.

## Teorema (Ley de los grandes números)

*Si  $X_1, X_2, \dots$  son independientes e idénticamente distribuidas, con  $\mathbb{E}[|X_1|] < \infty$ , entonces con probabilidad 1 :*

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n X_i = \mathbb{E}[X_1].$$

# Estudio en media

La **media** es una buena medida cuando pensamos ejecutar **muchas veces** un algoritmo.

## Teorema (Ley de los grandes números)

*Si  $X_1, X_2, \dots$  son independientes e idénticamente distribuidas, con  $\mathbb{E}[|X_1|] < \infty$ , entonces con probabilidad 1 :*

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n X_i = \mathbb{E}[X_1].$$

- ¿y si lo queremos ejecutar solamente una vez?
- ¿la **media** refleja la complejidad de **una sola ejecución**?

# Estudio en media

La **media** es una buena medida cuando pensamos ejecutar **muchas veces** un algoritmo.

## Teorema (Ley de los grandes números)

*Si  $X_1, X_2, \dots$  son independientes e idénticamente distribuidas, con  $\mathbb{E}[|X_1|] < \infty$ , entonces con probabilidad 1 :*

$$\lim_{n \rightarrow \infty} \frac{1}{n} \sum_{i=1}^n X_i = \mathbb{E}[X_1].$$

- ¿y si lo queremos ejecutar solamente una vez?
- ¿la **media** refleja la complejidad de **una sola ejecución**? En general: **no**.

# Concentración en probabilidad

Decimos que una secuencia de variables aleatorias  $X_n$  satisface  $X_n \sim f(n)$  en probabilidad sii, para cada  $\varepsilon > 0$  fijo,

$$\Pr(X_n \in [(1 - \varepsilon)f(n), (1 + \varepsilon)f(n)]) \rightarrow 1.$$

---

<sup>4</sup>De hecho se sabe mucho más al respecto, ver el artículo: C. McDiarmid y R. Hayward. 1992. *Strong concentration for Quicksort*. SODA '92.

# Concentración en probabilidad

Decimos que una secuencia de variables aleatorias  $X_n$  satisface  $X_n \sim f(n)$  en probabilidad si, para cada  $\varepsilon > 0$  fijo,

$$\Pr(X_n \in [(1 - \varepsilon)f(n), (1 + \varepsilon)f(n)]) \rightarrow 1.$$

Probaremos más tarde que la cantidad de comparaciones  $C_n$  en quicksort<sup>4</sup> satisface  $C_n \sim 2n \log n$  en probabilidad.

## Proposición

*La cantidad de comparaciones satisface  $C_n \sim 2n \log n$  en probabilidad.*

Las funciones generatrices nos ahorrarán muchos cálculos.

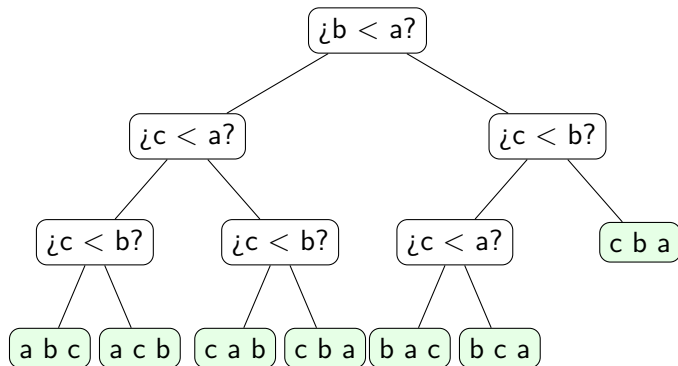
---

<sup>4</sup>De hecho se sabe mucho más al respecto, ver el artículo: C. McDiarmid y R. Hayward. 1992. *Strong concentration for Quicksort*. SODA '92.

# Optimalidad en media

Algoritmo basado en comparaciones se representa como árbol binario:

- nodos internos corresponden a comparaciones; rama izquierda False, rama derecha True.
- hojas corresponden a los posibles output del algoritmo.



# Optimalidad en media y entropía

Hojas son permutaciones  $\Rightarrow n!$  hojas.

- Altura del árbol [peor caso] es al menos  $\log_2(n!)$ ,
- $\log_2(n!) \sim n \log_2 n$

# Optimalidad en media y entropía

Hojas son permutaciones  $\Rightarrow n!$  hojas.

- Altura del árbol [peor caso] es al menos  $\log_2(n!)$ ,
- $\log_2(n!) \sim n \log_2 n$

Pero esto es también cierto para la media. Sea  $\ell_\pi$  la profundidad de la hoja  $\pi$ , notar que  $C_n(\pi) = \ell_\pi$  es la cantidad de comparaciones,

## Teorema (Profundidad media de un árbol binario)

*Para cualquier distribución  $\mathbf{p} = (p(\pi))_\pi$  sobre las hojas*

$$\mathbb{E}[\ell] = \sum \ell_\pi p(\pi) \geq H_2(\mathbf{p}),$$

*donde  $H_2(\mathbf{p}) = -\sum_\pi p(\pi) \log_2 p(\pi)$  es la entropía binaria.*



# Optimalidad en media y entropía

Hojas son permutaciones  $\Rightarrow n!$  hojas.

- Altura del árbol [peor caso] es al menos  $\log_2(n!)$ ,
- $\log_2(n!) \sim n \log_2 n$

Pero esto es también cierto para la media. Sea  $\ell_\pi$  la profundidad de la hoja  $\pi$ , notar que  $C_n(\pi) = \ell_\pi$  es la cantidad de comparaciones,

## Teorema (Profundidad media de un árbol binario)

Para cualquier distribución  $\mathbf{p} = (p(\pi))_\pi$  sobre las hojas

$$\mathbb{E}[\ell] = \sum \ell_\pi p(\pi) \geq H_2(\mathbf{p}),$$

donde  $H_2(\mathbf{p}) = -\sum_\pi p(\pi) \log_2 p(\pi)$  es la entropía binaria.

En nuestro caso  $p(\pi) = 1/n!$  para cada permutación, y  $\mathbb{E}[C_n] \geq \log_2 n!$ .

# Prueba: entropía es cota inferior

## Lema

*Para un árbol binario completo  $\sum_h \text{hojas} 2^{-\ell_h} = 1$*

## Lema

*Para todo  $x > 0$ ,  $\log x \leq x - 1$ . La igualdad se verifica si  $x = 1$ .*

# Prueba: entropía es cota inferior

## Lema

*Para un árbol binario completo  $\sum_h \text{hojas} 2^{-\ell_h} = 1$*

## Lema

*Para todo  $x > 0$ ,  $\log x \leq x - 1$ . La igualdad se verifica si  $x = 1$ .*

**Prueba del Teorema.** Combinamos los lemas con las propiedades del logaritmo, interpretando  $-\ell_h = \log 2^{-\ell_h}$ .

# QuickSort: modelo del input

- En nuestro modelo de quicksort el input  $\pi$  es una **permutación uniforme** :

$$\Pr(\pi = (a_1, \dots, a_n)) = (n!)^{-1}.$$

# QuickSort: modelo del input

- En nuestro modelo de quicksort el input  $\pi$  es una **permutación uniforme** :

$$\Pr(\pi = (a_1, \dots, a_n)) = (n!)^{-1}.$$

- Corresponde a considerar  $n$  números (flotantes) de  $[0, 1]$ .

# QuickSort: modelo del input

- En nuestro modelo de quicksort el input  $\pi$  es una **permutación uniforme** :

$$\Pr(\pi = (a_1, \dots, a_n)) = (n!)^{-1}.$$

- Corresponde a considerar  $n$  números (flotantes) de  $[0, 1]$ .
- Razonable sin conocimiento a priori del input.

# QuickSort: modelo del input

- En nuestro modelo de quicksort el input  $\pi$  es una **permutación uniforme** :

$$\Pr(\pi = (a_1, \dots, a_n)) = (n!)^{-1}.$$

- Corresponde a considerar  $n$  números (flotantes) de  $[0, 1]$ .
- Razonable sin conocimiento a priori del input.

Otros algoritmos (Powersort, Timsort, ...) suponen que el input puede estar **parcialmente ordenado** en pedazos:

- input dividido en *runs* crecientes/decrecientes de longitud  $a_1, \dots, a_r$ .

$$\underbrace{[1, 5, 7]}_{a_1=3}, \underbrace{[2, 4, 9]}_{a_2=3}, \underbrace{[6, 4, 4]}_{a_3=3}, \underbrace{[12, 4]}_{a_4=2};$$

# QuickSort: modelo del input

- En nuestro modelo de quicksort el input  $\pi$  es una **permutación uniforme** :

$$\Pr(\pi = (a_1, \dots, a_n)) = (n!)^{-1}.$$

- Corresponde a considerar  $n$  números (flotantes) de  $[0, 1]$ .
- Razonable sin conocimiento a priori del input.

Otros algoritmos (Powersort, Timsort, ...) suponen que el input puede estar **parcialmente ordenado** en pedazos:

- input dividido en *runs* crecientes/decrecientes de longitud  $a_1, \dots, a_r$ .

$$\underbrace{[1, 5, 7]}_{a_1=3}, \underbrace{[2, 4, 9]}_{a_2=3}, \underbrace{[6, 4, 4]}_{a_3=3}, \underbrace{[12, 4]}_{a_4=2}; \text{ o quizás } \underbrace{[1, 5, 7]}_{a_1=3}, \underbrace{[2, 4, 9]}_{a_2=3}, \underbrace{[6, 4]}_{a_3=2}, \underbrace{[4, 12]}_{a_4=2}, \underbrace{[4]}_{a_5=1}.$$



# QuickSort: modelo del input

- En nuestro modelo de quicksort el input  $\pi$  es una **permutación uniforme** :

$$\Pr(\pi = (a_1, \dots, a_n)) = (n!)^{-1}.$$

- Corresponde a considerar  $n$  números (flotantes) de  $[0, 1]$ .
- Razonable sin conocimiento a priori del input.

Otros algoritmos (Powersort, Timsort, ...) suponen que el input puede estar **parcialmente ordenado** en pedazos:

- input dividido en *runs* crecientes/decrecientes de longitud  $a_1, \dots, a_r$ .

$$\underbrace{[1, 5, 7]}_{a_1=3}, \underbrace{[2, 4, 9]}_{a_2=3}, \underbrace{[6, 4, 4]}_{a_3=3}, \underbrace{[12, 4]}_{a_4=2}; \text{ o quizás } \underbrace{[1, 5, 7]}_{a_1=3}, \underbrace{[2, 4, 9]}_{a_2=3}, \underbrace{[6, 4]}_{a_3=2}, \underbrace{[4, 12]}_{a_4=2}, \underbrace{[4]}_{a_5=1}.$$

- merge(sort) inteligente aprovecha los runs existentes!

# QuickSort: modelo del input

- En nuestro modelo de quicksort el input  $\pi$  es una **permutación uniforme** :

$$\Pr(\pi = (a_1, \dots, a_n)) = (n!)^{-1}.$$

- Corresponde a considerar  $n$  números (flotantes) de  $[0, 1]$ .
- Razonable sin conocimiento a priori del input.

Otros algoritmos (Powersort, Timsort, ...) suponen que el input puede estar **parcialmente ordenado** en pedazos:

- input dividido en *runs* crecientes/decrecientes de longitud  $a_1, \dots, a_r$ .

$$\underbrace{[1, 5, 7]}_{a_1=3}, \underbrace{[2, 4, 9]}_{a_2=3}, \underbrace{[6, 4, 4]}_{a_3=3}, \underbrace{[12, 4]}_{a_4=2}; \text{ o quizás } \underbrace{[1, 5, 7]}_{a_1=3}, \underbrace{[2, 4, 9]}_{a_2=3}, \underbrace{[6, 4]}_{a_3=2}, \underbrace{[4, 12]}_{a_4=2}, \underbrace{[4]}_{a_5=1}.$$

- merge(sort) inteligente aprovecha los runs existentes!

La elección del modelo probabilista es un paso clave.

# Fusión de dos runs

Run A: 

2
---

5
---

7
---

Run B: 

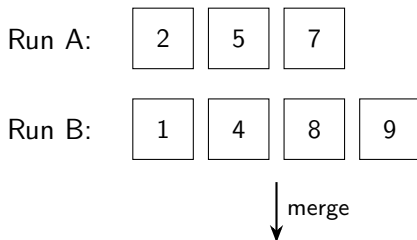
1
---

4
---

8
---

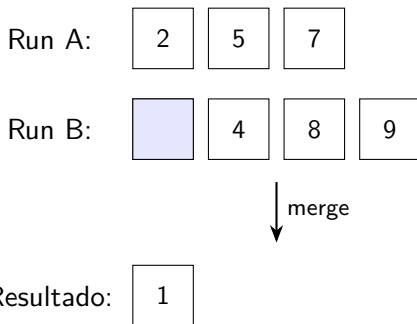
9
---

# Fusión de dos runs

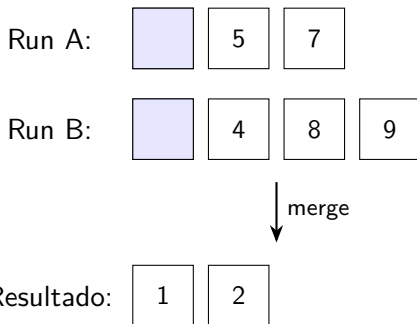


Resultado:

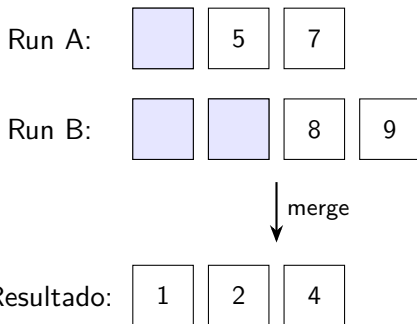
# Fusión de dos runs



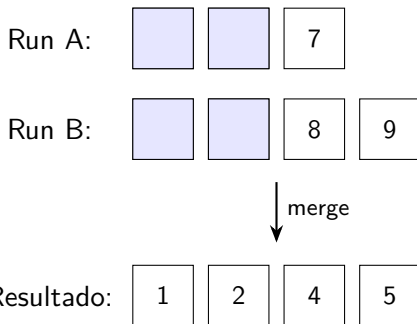
# Fusión de dos runs



# Fusión de dos runs

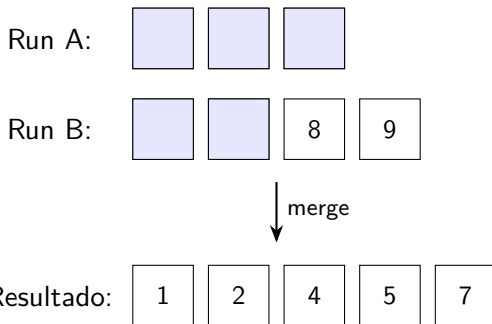


# Fusión de dos runs





# Fusión de dos runs




# Fusión de dos runs

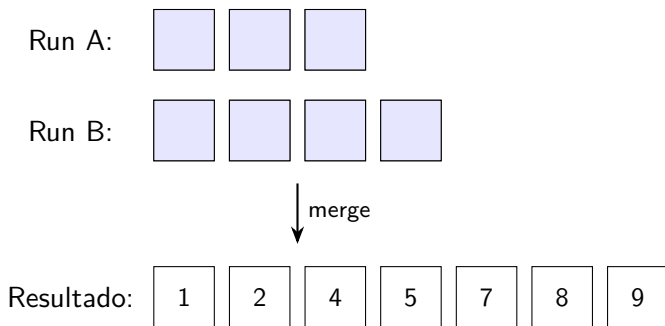
Run A: 

Run B: 

↓ merge

Resultado: 

# Fusión de dos runs



**Suposición:** la fusión (merge) de dos runs (corridas), de longitud  $a_1$  y  $a_2$ , cuesta  $a_1 + a_2$ .

# Entropía de runs

## Teorema

*El costo  $C$  de cualquier algoritmo basado en la fusión de runs<sup>a</sup> satisface*

$$C(\pi) \geq n \cdot \mathcal{H}(\pi),$$

*donde  $\mathcal{H} = H_2(a_1/n, \dots, a_r/n) = -\sum \frac{a_i}{n} \log_2 \frac{a_i}{n}$  es la entropía de run de  $\pi$ .*

---

<sup>a</sup>Sin contar la detección de runs.

# Entropía de runs

## Teorema

El costo  $C$  de cualquier algoritmo basado en la fusión de runs<sup>a</sup> satisface

$$C(\pi) \geq n \cdot \mathcal{H}(\pi),$$

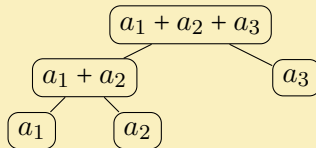
donde  $\mathcal{H} = H_2(a_1/n, \dots, a_r/n) = -\sum \frac{a_i}{n} \log_2 \frac{a_i}{n}$  es la entropía de run de  $\pi$ .

---

<sup>a</sup>Sin contar la detección de runs.

## Demostración.

- Estrategia de fusión corresponde a árbol binario  $\Rightarrow$  costo  $C = \sum a_i \ell_i$ .



$$C = 2 \cdot a_1 + 2 \cdot a_2 + 1 \cdot a_3$$

# Entropía de runs

## Teorema

*El costo  $C$  de cualquier algoritmo basado en la fusión de runs<sup>a</sup> satisface*

$$C(\pi) \geq n \cdot \mathcal{H}(\pi),$$

*donde  $\mathcal{H} = H_2(a_1/n, \dots, a_r/n) = -\sum \frac{a_i}{n} \log_2 \frac{a_i}{n}$  es la entropía de run de  $\pi$ .*

---

<sup>a</sup>Sin contar la detección de runs.

## Demostración.

- Estrategia de fusión corresponde a árbol binario  $\Rightarrow$  costo  $C = \sum a_i \ell_i$ .
- Renormalizando obtenemos el resultado. □

# Entropía de runs

## Teorema

*El costo  $C$  de cualquier algoritmo basado en la fusión de runs satisface*

$$C(\pi) \geq n \cdot \mathcal{H}(\pi),$$

*donde  $\mathcal{H} = H_2(a_1/n, \dots, a_r/n) = -\sum \frac{a_i}{n} \log_2 \frac{a_i}{n}$  es la entropía de run de  $\pi$ .*

$\mathcal{H}$  puede ser mucho menor que  $\log_2 n$ .

## Proposición

*Tenemos  $\mathcal{H} \leq \log_2 r$  donde  $r$  es la cantidad de runs.*

# Entropía de runs

## Teorema

*El costo  $C$  de cualquier algoritmo basado en la fusión de runs satisface*

$$C(\pi) \geq n \cdot \mathcal{H}(\pi),$$

*donde  $\mathcal{H} = H_2(a_1/n, \dots, a_r/n) = -\sum \frac{a_i}{n} \log_2 \frac{a_i}{n}$  es la entropía de run de  $\pi$ .*

$\mathcal{H}$  puede ser mucho menor que  $\log_2 n$ .

## Proposición

*Tenemos  $\mathcal{H} \leq \log_2 r$  donde  $r$  es la cantidad de runs.*

$\Rightarrow$  Existen varios algoritmos en tiempo  $\Theta(n\mathcal{H} + n)$ .



# Entropía de runs

No se pierde mucho trabajando solo con fusiones.

## Teorema (Barbay, Navarro, '13)

*Sea  $\mathcal{C} = \mathcal{C}(a_1, \dots, a_r)$  la clase de las permutaciones con runs de largo  $a_1, a_2, \dots, a_r$ , con  $a_i \geq 2$  para  $i = 1, \dots, r - 1$ .*

*Para todo algoritmo  $\mathcal{A}$  basado en la comparación de pares de elementos, existe un elemento  $\pi \in \mathcal{C}$  que requiere al menos  $n\mathcal{H} - 3n$  comparaciones.*

# Entropía de runs

No se pierde mucho trabajando solo con fusiones.

## Teorema (Barbay, Navarro, '13)

Sea  $\mathcal{C} = \mathcal{C}(a_1, \dots, a_r)$  la clase de las permutaciones con runs de largo  $a_1, a_2, \dots, a_r$ , con  $a_i \geq 2$  para  $i = 1, \dots, r - 1$ .

Para todo algoritmo  $\mathcal{A}$  basado en la comparación de pares de elementos, existe un elemento  $\pi \in \mathcal{C}$  que requiere al menos  $n\mathcal{H} - 3n$  comparaciones.

## Borrador de prueba.

Siempre existe  $\pi$  que requiere al menos  $\log_2 |\mathcal{C}|$  operaciones.

Se necesita una cota [no trivial<sup>a</sup>], en este caso  $2^{r-1} |\mathcal{C}| \geq \binom{n}{a_1, \dots, a_r}$ . □

---

<sup>a</sup>Ver referencias, en particular <https://arxiv.org/pdf/1805.08612>

Tim Peters<sup>5</sup> diseñó en 2002 un nuevo algoritmo para Python:

*This describes an adaptive, stable, natural mergesort, modestly called timsort (hey, I earned it <wink>). It has supernatural performance on many kinds of partially ordered arrays (less than  $\lg(N!)$  comparisons needed, and as few as  $N-1$ ), yet as fast as Python's previous highly tuned samplesort hybrid on random arrays.*

*In a nutshell, the main routine marches over the array once, left to right, alternately identifying the next run, then merging it into the previous runs "intelligently". Everything else is complication for speed, and some hard-won measure of memory efficiency.*

---

<sup>5</sup><https://svn.python.org/projects/python/trunk/Objects/listsort.txt>

# TimSort principio e historia

- Leer **runs** de izquierda a derecha, agregándolas a una **pila (stack)**.
- La pila  $\rightarrow R_1, R_2, \dots$  debe satisfacer un *invariante*:  
si el invariante no se cumple, desencadena secuencia de fusiones.
- Merges se realizan entre runs adyacentes (localidad/cache).

# TimSort principio e historia

- Leer **runs** de izquierda a derecha, agregándolas a una **pila (stack)**.
- La pila  $\rightarrow R_1, R_2, \dots$  debe satisfacer un *invariante*:  
si el invariante no se cumple, desencadena secuencia de fusiones.
- Merges se realizan entre runs adyacentes (localidad/cache).

Invariante inspirado por **Fibonacci**:

$$r_{i+2} > r_i + r_{i+1}, \quad r_{i+1} > r_i,$$

donde  $r_i = |R_i|$  son las longitudes.

- Varias condiciones de merge  $\Rightarrow$  originalmente con bugs ! 🐛
- Algoritmo era usado en Python [ahora PowerSort], usado en Java.
- Ha inspirado muchos algoritmos nuevos, basados en runs.

# TimSort reglas e historia

Sea la pila  $\rightarrow R_1, R_2, \dots$  y las longitudes  $\rightarrow r_1, r_2, \dots$

**while** *runs fuera de pila*  $\neq \emptyset$  **do**

tomar primer run (izq. a der.) y agregar a la pila ;

**while** *True* **do**

**if**  $r_1 > r_3$  **then**

        | merge  $R_2$  and  $R_3$

**else if**  $r_1 \geq r_2$  **then**

        | merge  $R_1$  and  $R_2$ ;

**else if**  $r_1 + r_2 \geq r_3$  **then**

        | merge  $R_1$  and  $R_2$ ;

**else if**  $r_2 + r_3 \geq r_4$  **then**

        | merge  $R_1$  and  $R_2$ ;

**else**

        | **break**

# TimSort reglas e historia

Sea la pila  $\rightarrow R_1, R_2, \dots$  y las longitudes  $\rightarrow r_1, r_2, \dots$

**while** *runs fuera de pila*  $\neq \emptyset$  **do**

tomar primer run (izq. a der.) y agregar a la pila ;

**while** *True* **do**

**if**  $r_1 > r_3$  **then**

| merge  $R_2$  and  $R_3$

**else if**  $r_1 \geq r_2$  **then**

| merge  $R_1$  and  $R_2$ ;

**else if**  $r_1 + r_2 \geq r_3$  **then**

| merge  $R_1$  and  $R_2$ ;

**else if**  $r_2 + r_3 \geq r_4$  **then**

| merge  $R_1$  and  $R_2$ ;

**else**

| **break**

- La condición en rojo no estaba en la versión original. Bug descubierto por [de Gouw et al \(2015\)](#) cuando trataban de probar formalmente la corrección. 🦋
- Existía un segundo bug: el tamaño máximo de la pila en Java. Descubierto por [Auger et al \(2018\)](#) al realizar el análisis preciso del algoritmo. 🦋

# Optimalidad en media y entropía

Teorema (Auger, Jugé, Nicaud, Pivoteau '18)

*En el peor caso TimSort es  $1,5n\mathcal{H} + O(n)$ .*



# Optimalidad en media y entropía

Teorema (Auger, Jugé, Nicaud, Pivoteau '18)

*En el peor caso TimSort es  $1,5n\mathcal{H} + O(n)$ .*

TimSort no es óptimo

Teorema (Wild, Munro '18)

*En el peor caso PowerSort es  $n\mathcal{H} + O(n)$ .*

# Optimalidad en media y entropía

Teorema (Auger, Jugé, Nicaud, Pivoteau '18)

*En el peor caso TimSort es  $1,5n\mathcal{H} + O(n)$ .*

TimSort no es óptimo

Teorema (Wild, Munro '18)

*En el peor caso PowerSort es  $n\mathcal{H} + O(n)$ .*

Una permutación aleatoria (típica) tiene muchos runs cortos!

$n = 20$  :  $[\textcolor{red}{11}, \textcolor{red}{18}, \textcolor{blue}{1}, \textcolor{blue}{5}, \textcolor{red}{2}, \textcolor{red}{14}, \textcolor{red}{20}, \textcolor{blue}{3}, \textcolor{blue}{8}, \textcolor{blue}{15}, \textcolor{red}{6}, \textcolor{red}{4}, \textcolor{blue}{16}, \textcolor{blue}{17}, \textcolor{red}{13}, \textcolor{red}{10}, \textcolor{blue}{19}, \textcolor{blue}{9}, \textcolor{red}{7}, \textcolor{red}{12}]$ .

$$\mathcal{H} = 3,1\dots, \quad \log_2 20 = 4,3\dots$$

# Probabilidad de run de largo $\geq k$

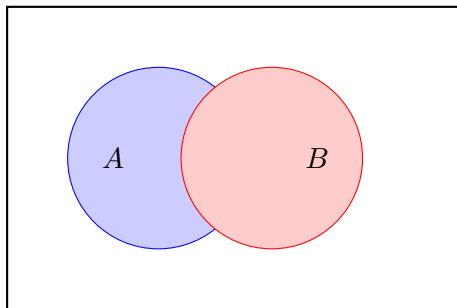
Sea  $S_i$ : run de longitud  $\geq k$  comienza en  $i$ . Notar que  $\Pr(S_i) \leq 2/k!$

# Probabilidad de run de largo $\geq k$

Sea  $S_i$ : run de longitud  $\geq k$  comienza en  $i$ . Notar que  $\Pr(S_i) \leq 2/k!$

Técnica: Union bound

$$\Pr(A \cup B) \leq \Pr(A) + \Pr(B)$$



# Probabilidad de run de largo $\geq k$

Sea  $S_i$ : run de longitud  $\geq k$  comienza en  $i$ . Notar que  $\Pr(S_i) \leq 2/k!$

## Técnica: Union bound

$$\Pr(A \cup B) \leq \Pr(A) + \Pr(B)$$

Por el union bound tenemos:

$$P(n, k) := \Pr(\exists \text{run de longitud } \geq k) = \Pr\left(\bigcup_i S_i\right) \leq \sum_i \Pr(S_i) \leq 2n/k!.$$

## Proposición

$$P = P(n, k) \leq 2 \exp(\log n - k \log k + k).$$

# Entropía de corridas de permutación aleatoria

Utilizando

$$P = P(n, k) \leq 2 \exp(\log n - k \log k + k),$$

obtenemos que para  $k \geq 2 \frac{\log n}{\log \log n}$ ,  $P(n, k) \rightarrow 0$ . Las runs son cortas !

## Proposición

Con alta probabilidad (es decir  $p \rightarrow 1$ ) todas las runs  $A_1, \dots, A_r$  de una permutación aleatoria uniforme satisfacen  $A_i \leq 2 \frac{\log n}{\log \log n}$ .

# Entropía de corridas de permutación aleatoria

Utilizando

$$P = P(n, k) \leq 2 \exp(\log n - k \log k + k),$$

obtenemos que para  $k \geq 2 \frac{\log n}{\log \log n}$ ,  $P(n, k) \rightarrow 0$ . Las runs son cortas !

## Proposición

Con alta probabilidad (es decir  $p \rightarrow 1$ ) todas las runs  $A_1, \dots, A_r$  de una permutación aleatoria uniforme satisfacen  $A_i \leq 2 \frac{\log n}{\log \log n}$ .

## Corolario

Con alta probabilidad, para una permutación aleatoria uniforme<sup>a</sup>,  
 $\mathcal{H} \geq \sum \frac{A_i}{n} \log_2 \left( \frac{n}{2(\log n)/\log \log n} \right) = \log_2 n + O(\log \log n)$ ,  $\mathcal{H} \leq \log_2 n$ .

---

<sup>a</sup>La constante del término  $O$  en realidad se puede calcular explícitamente y no depende de la secuencia de conjuntos elegidos, cuya probabilidad tiende a 1.

# Entropía de corridas de permutación aleatoria

Utilizando

$$P = P(n, k) \leq 2 \exp(\log n - k \log k + k),$$

obtenemos que para  $k \geq 2 \frac{\log n}{\log \log n}$ ,  $P(n, k) \rightarrow 0$ . Las runs son cortas !

## Proposición

Con alta probabilidad (es decir  $p \rightarrow 1$ ) todas las runs  $A_1, \dots, A_r$  de una permutación aleatoria uniforme satisfacen  $A_i \leq 2 \frac{\log n}{\log \log n}$ .

## Corolario

Con alta probabilidad, para una permutación aleatoria uniforme<sup>a</sup>,  
 $\mathcal{H} \geq \sum \frac{A_i}{n} \log_2 \left( \frac{n}{2(\log n)/\log \log n} \right) = \log_2 n + O(\log \log n)$ ,  $\mathcal{H} \leq \log_2 n$ .

---

<sup>a</sup>La constante del término  $O$  en realidad se puede calcular explícitamente y no depende de la secuencia de conjuntos elegidos, cuya probabilidad tiende a 1.

$\implies$  Modelo de permutaciones uniformes  $\neq$  modelo de runs largas



# Con alta probabilidad y en media

Probamos que, con probabilidad  $p \rightarrow 1$ ,

$$\mathcal{H} = \log_2 n + O(\log \log n),$$

es decir, que esto se cumple para  $\pi \in A_n \subseteq S_n$  con  $\Pr(A_n) \rightarrow 1$ .

## Pregunta

¿Qué podemos decir sobre la esperanza  $\mathbb{E}[\mathcal{H}]$  ?

# Con alta probabilidad y en media

Probamos que, con probabilidad  $p \rightarrow 1$ ,

$$\mathcal{H} = \log_2 n + O(\log \log n),$$

es decir, que esto se cumple para  $\pi \in A_n \subseteq S_n$  con  $\Pr(A_n) \rightarrow 1$ .

## Pregunta

¿Qué podemos decir sobre la esperanza  $\mathbb{E}[\mathcal{H}]$  ?

$$\begin{aligned}\mathbb{E}[\mathcal{H}] &= \Pr(A_n) \times \mathbb{E}[\mathcal{H} \mid A_n] + \Pr(A_n^c) \times \mathbb{E}[\mathcal{H} \mid A_n^c], \\ &\geq \Pr(A_n) \times \mathbb{E}[\mathcal{H} \mid A_n], \\ &= \Pr(A_n) \times (\log_2 n + O(\log \log n)) .\end{aligned}$$

# Con alta probabilidad y en media

Probamos que, con probabilidad  $p \rightarrow 1$ ,

$$\mathcal{H} = \log_2 n + O(\log \log n),$$

es decir, que esto se cumple para  $\pi \in A_n \subseteq S_n$  con  $\Pr(A_n) \rightarrow 1$ .

## Pregunta

¿Qué podemos decir sobre la esperanza  $\mathbb{E}[\mathcal{H}]$  ?

$$\begin{aligned}\mathbb{E}[\mathcal{H}] &= \Pr(A_n) \times \mathbb{E}[\mathcal{H} \mid A_n] + \Pr(A_n^c) \times \mathbb{E}[\mathcal{H} \mid A_n^c], \\ &\geq \Pr(A_n) \times \mathbb{E}[\mathcal{H} \mid A_n], \\ &= \Pr(A_n) \times (\log_2 n + O(\log \log n)) .\end{aligned}$$

Para la cota superior tenemos suerte:  $\mathcal{H} \leq \log_2(n)$  siempre.

**Conclusión:**  $\mathbb{E}[\mathcal{H}] \sim \log_2 n$  también.

# Problema: número de runs

```
def runs(arr): # arr = permutacion
    res = []
    i, n = 0, len(arr)
    while i < n:
        j = i + 1
        if j < n and arr[i] <= arr[j]:
            # creciente
            while j < n and arr[j - 1] <= arr[j]:
                j += 1
        elif j < n and arr[i] > arr[j]:
            # decreciente
            while j < n and arr[j - 1] > arr[j] :
                j += 1
        else:
            # elemento aislado
            j = i + 1
        res.append(j - i)
        i = j
    return res
```

# Problema: número de runs

## Problema

La cantidad esperada de runs es  $\mathbb{E}[r] \sim cn$  para una cierta  $c > 0$ .

Veamos la permutación como una secuencia  $X_1, X_2, \dots$  de números iid de  $[0, 1]$ .

- (a) Probar  $runs(X_1, \dots, X_{i+j}) \leq runs(X_1, \dots, X_i) + runs(X_{i+1}, \dots, X_{i+j})$ .
- (b) Probar que  $e_k := \mathbb{E}[runs(X_1, \dots, X_k)]$  satisface  $e_{i+j} \leq e_i + e_j$  para todo  $i, j \geq 0$ . Concluir que  $e_k/k \rightarrow c$  para cierta  $c \geq 0$ .
- (c) Mostrar que la constante es positiva  $c > 0$ .

---

<sup>5</sup>Pista (b). Lema de Fekete...

<sup>5</sup>Pista (c). ¿Qué podemos decir si  $X_i < X_{i+1}$  y  $X_{i+1} > X_{i+2}$  ?

# Entropía de corrida: modelo aleatorio

## Distribución de Zipf

Dado  $\alpha > 1$ , consideramos

$$\Pr(\ell = k) \propto k^{-\alpha}.$$

Cuando  $\alpha \leq 2$ , la longitud esperada de  $\ell$  es infinita.

# Entropía de corrida: modelo aleatorio

## Distribución de Zipf

Dado  $\alpha > 1$ , consideramos

$$\Pr(\ell = k) \propto k^{-\alpha}.$$

Cuando  $\alpha \leq 2$ , la longitud esperada de  $\ell$  es infinita.

- Valores más irregulares. Ejemplo con  $\alpha = 3/2$

1	1	2	1	6	8	9	14	3	5
953	1	6	32	2	24	1	1	3	1
21	1	26	2	1	1	9	2	49	4
1	1	1	1	2	48	68	4	189	2

- Usada para modelar frecuencias de palabras en lenguaje natural.

# Entropía de corrida: modelo aleatorio

## Distribución de Zipf

Dado  $\alpha > 1$ , consideramos

$$\Pr(\ell = k) \propto k^{-\alpha}.$$

Cuando  $\alpha \leq 2$ , la longitud esperada de  $\ell$  es infinita.

- Valores más irregulares. Ejemplo con  $\alpha = 3/2$

1	1	2	1	6	8	9	14	3	5
953	1	6	32	2	24	1	1	3	1
21	1	26	2	1	1	9	2	49	4
1	1	1	1	2	48	68	4	189	2

- Usada para modelar frecuencias de palabras en lenguaje natural.

¿Modelo más razonable? ¿Producir permutación con longitudes dadas?



# Para aprender más



*Nicolas Auger, Vincent Jugé, Cyril Nicaud, y Carine Pivoteau,*  
On the Worst-Case Complexity of TimSort  
<https://arxiv.org/pdf/1805.08612>



*Jérémy Barbay y Gonzalo Navarro,*  
On compressing permutations and adaptive sorting.  
<http://dx.doi.org/10.1016/j.tcs.2013.10.019>



*Nearly-Optimal Mergesorts: Fast, Practical Sorting Methods That Optimally Adapt to Existing Runs,*  
<https://doi.org/10.4230/LIPIcs.ESA.2018.63>

# Tablas de Hash

## Motivación

Implementar un array asociativo  $m$ :

- universo  $\mathcal{U}$  de claves  $k \in \mathcal{U}$  grande,
- asociar a cada clave  $k$  un valor  $m[k]$ ,
- insertar, buscar, borrar...

# Tablas de Hash

## Motivación

Implementar un array asociativo  $m$ :

- universo  $\mathcal{U}$  de claves  $k \in \mathcal{U}$  grande,
- asociar a cada clave  $k$  un valor  $m[k]$ ,
- insertar, buscar, borrar...

Las *tablas de Hash*:

- **Idea** : utilizar un array  $A$  pequeño, de tamaño  $K \ll |\mathcal{U}|$ 
  - considerar una función  $h : \mathcal{U} \rightarrow \mathbb{Z}$  *pseudo-aleatoria*,
  - insertar  $k$  en  $A[i]$  donde  $i = h(k) \bmod K$ . [modelo :  $i$  es uniforme]
- **Problema** : colisiones, dos keys  $k_1$  y  $k_2$  con  $h(k_1) = h(k_2)$ .

# La paradoja del cumpleaños

Las colisiones están relacionadas con la famosa *paradoja del cumpleaños*:

## Paradoja del cumpleaños

¿Cuál es el **número mínimo de personas** requerido para que la probabilidad de que dos o más personas tengan el mismo cumpleaños<sup>a</sup> sea mayor que  $1/2$ ?

---

<sup>a</sup>solo el día del año, no el año

# La paradoja del cumpleaños

Las colisiones están relacionadas con la famosa *paradoja del cumpleaños*:

## Paradoja del cumpleaños

¿Cuál es el **número mínimo de personas** requerido para que la probabilidad de que dos o más personas tengan el mismo cumpleaños<sup>a</sup> sea mayor que  $1/2$ ?

---

<sup>a</sup>solo el día del año, no el año

Más en general, ¿cuántas personas para la primera “colisión”?

# La paradoja del cumpleaños

Las colisiones están relacionadas con la famosa *paradoja del cumpleaños*:

## Paradoja del cumpleaños

¿Cuál es el **número mínimo de personas** requerido para que la probabilidad de que dos o más personas tengan el mismo cumpleaños<sup>a</sup> sea mayor que  $1/2$ ?

---

<sup>a</sup>solo el día del año, no el año

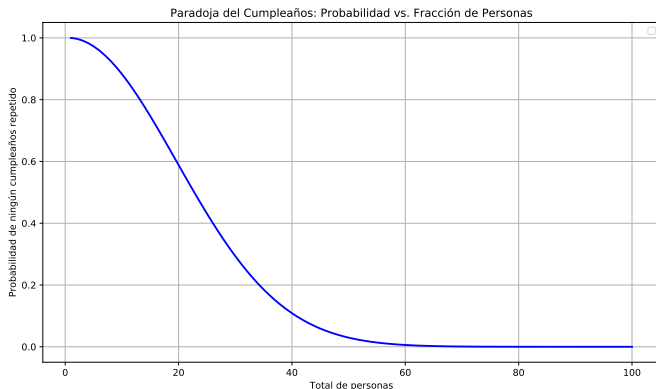
Más en general, ¿cuántas personas para la primera “colisión”?

- Supongamos que tenemos  $K$  posibles valores ( $K = 365$ ),
- y consideramos  $n$  elementos (las personas),
- ¿cuál es la probabilidad de que hayan dos elementos iguales?

**Modelo:** cada valor tiene probabilidad  $1/K$ , elementos independientes.

# La paradoja del cumpleaños

$$p_n = \Pr(n \text{ valores distintos}) = \prod_{i=1}^{n-1} \left(1 - \frac{i}{K}\right).$$



La probabilidad de al menos un cumpleaños repetido es  $q_n = 1 - p_n$ .

# La paradoja del cumpleaños

Estimemos la probabilidad de  $n$  valores distintos:  $p_n = \prod_{i=1}^{n-1} \left(1 - \frac{i}{K}\right)$ ,



# La paradoja del cumpleaños

Estimemos la probabilidad de  $n$  valores distintos:  $p_n = \prod_{i=1}^{n-1} \left(1 - \frac{i}{K}\right)$ ,

## Proposición

*Considerando  $n \sim \sqrt{2\theta K}$  con  $K \rightarrow \infty$ ,  $p_n \sim e^{-\theta}$ .*

# La paradoja del cumpleaños

Estimemos la probabilidad de  $n$  valores distintos:  $p_n = \prod_{i=1}^{n-1} \left(1 - \frac{i}{K}\right)$ ,

## Proposición

*Considerando  $n \sim \sqrt{2\theta K}$  con  $K \rightarrow \infty$ ,  $p_n \sim e^{-\theta}$ .*

## Demostración.

Para la prueba usamos las desigualdades  $1 + x \leq e^x$ , válida para  $x \in \mathbb{R}$ , y  $1 + x \geq e^{x-x^2/2}$ , válida para  $x \in [0, 1]$ . □

# La paradoja del cumpleaños

Estimemos la probabilidad de  $n$  valores distintos:  $p_n = \prod_{i=1}^{n-1} \left(1 - \frac{i}{K}\right)$ ,

## Proposición

*Considerando  $n \sim \sqrt{2\theta K}$  con  $K \rightarrow \infty$ ,  $p_n \sim e^{-\theta}$ .*

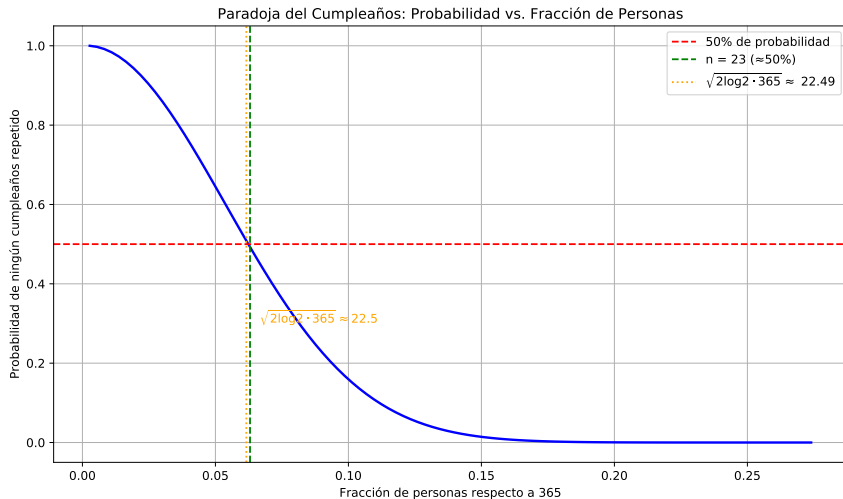
## Demostración.

Para la prueba usamos las desigualdades  $1 + x \leq e^x$ , válida para  $x \in \mathbb{R}$ , y  $1 + x \geq e^{x-x^2/2}$ , válida para  $x \in [0, 1]$ . □

Primera colisión ocurre (con gran proba.) cuando  $n$  es de orden  $\sqrt{K}$ .

# La paradoja del cumpleaños

Ilustración de la aproximación:  $n \sim \sqrt{2\theta K}$ ,  $p_n \sim e^{-\theta}$  con  $\theta = \log 2$



# Tablas de Hash

Tablas de Hash requieren un mecanismo de resolución de colisiones:

- **Política de resolución de colisiones :**

1. [External Hashing] Cada célula  $A[i]$  contiene una lista encadenada.
2. [Internal Hashing / Open addressing ] Si la célula está ya ocupada, buscar otra en el mismo array.

- **Política de rehashing :**

- si tasa de ocupación<sup>6</sup> del array  $A$  es alta, nuevo array de tamaño mayor,
- necesario re-insertar todo. [paso lento!]

---

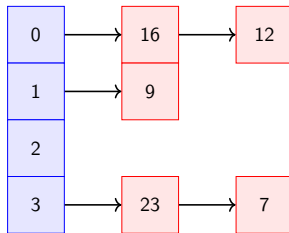
<sup>6</sup>“Load factor” en inglés.

# Tablas de Hash

Tablas de Hash requieren un mecanismo de resolución de colisiones:

- **Política de resolución de colisiones :**

1. [External Hashing] Cada célula  $A[i]$  contiene una lista encadenada.



2. [Internal Hashing / Open addressing ] Si la célula está ya ocupada, buscar otra en el mismo array.

- **Política de rehashing :**

- si tasa de ocupación<sup>6</sup> del array  $A$  es alta, nuevo array de tamaño mayor,
- necesario re-insertar todo. [paso lento!]

---

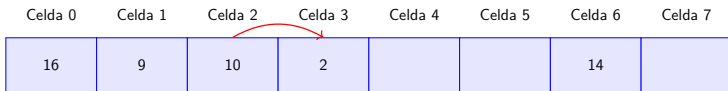
<sup>6</sup>“Load factor” en inglés.

# Tablas de Hash

Tablas de Hash requieren un mecanismo de resolución de colisiones:

- **Política de resolución de colisiones :**

1. [External Hashing] Cada célula  $A[i]$  contiene una lista encadenada.
2. [Internal Hashing / Open addressing ] Si la célula está ya ocupada, buscar otra en el mismo array.



[la flecha roja es indicativa]

- **Política de rehashing :**

- si tasa de ocupación<sup>6</sup> del array  $A$  es alta, nuevo array de tamaño mayor,
- necesario re-insertar todo. [paso lento!]

---

<sup>6</sup>“Load factor” en inglés.

# Rehashing

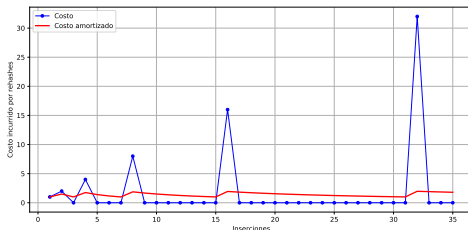
Cuando el load factor  $\alpha = n/K$  excede un valor dado  $\gamma$  (p.e.,  $\gamma = 0,85$ ), considerar un array nuevo con capacidad<sup>7</sup>  $K' = 2K$ .

## Proposición

*El costo amortizado por inserción es constante.*

## Concepto: costo amortizado

En lugar de considerar el costo de una sola operación  $c_t$ , nos interesa el costo medio de la secuencia total de operaciones  $\frac{1}{T} \sum_{t=1}^T c_t$ .

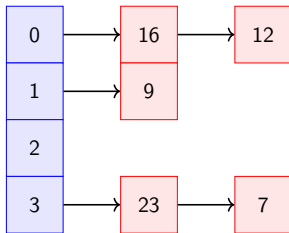


<sup>7</sup>Aquí no consideramos que el tamaño puede reducirse.



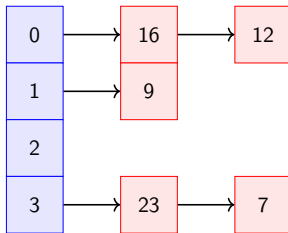
# External hashing

Cada celda  $A[i]$  contiene una lista encadenada



# External hashing

Cada celda  $A[i]$  contiene una lista encadenada

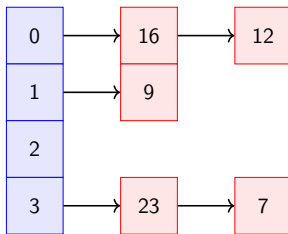


## Observación

Cada celda contiene en media  $\alpha = n/K$  elementos.

# External hashing

Cada celda  $A[i]$  contiene una lista encadenada



## Observación

Cada celda contiene en media  $\alpha = n/K$  elementos.

## Proposición (Lookup)

Con alta probabilidad ( $K \rightarrow \infty$ ), ninguna lista tiene longitud mayor que  $2 \frac{\log K}{\log \log K}$ .

# External hashing

Consideremos solo inserciones. Sea  $\gamma > 0$  la tasa de ocupación máxima,  $n/K \leq \gamma$ .

## Proposición (Lookup)

*Con alta probabilidad ( $K \rightarrow \infty$ ), ninguna lista tiene longitud mayor que  $2 \frac{\log K}{\log \log K}$ .*

# External hashing

Consideremos solo inserciones. Sea  $\gamma > 0$  la tasa de ocupación máxima,  $n/K \leq \gamma$ .

## Proposición (Lookup)

*Con alta probabilidad ( $K \rightarrow \infty$ ), ninguna lista tiene longitud mayor que  $2 \frac{\log K}{\log \log K}$ .*

## Sketch de demostración.

Sea  $X_1, \dots, X_n$  la secuencia de celda elegidas para las  $n$  inserciones.

1. Consideramos celda  $C_0$ , y sea  $C_0(n)$  la lista luego de  $n$  inserciones.
2. Probamos  $\Pr(|C_0(n)| \geq m) \leq \binom{n}{m} K^{-m}$  por el union-bound.
3. Si consideramos cualquier celda, por el union-bound

$$P_m := \Pr(\exists j : |C_j(n)| \geq m) \leq K \times \binom{n}{m} K^{-m}.$$

Ahora, tomando  $m = 2 \frac{\log K}{\log \log K}$  probamos que la cota tiende a 0. □

# Internal hashing / Open addressing

**Internal hashing:** Si la celda está ya ocupada, buscar otra en el mismo array.

- Internal hashing / Open addressing es más común en la actualidad.
- Muchas estrategias para decidir la secuencia (probe sequence).

# Internal hashing / Open addressing

**Internal hashing:** Si la celda está ya ocupada, buscar otra en el mismo array.

- Internal hashing / Open addressing es más común en la actualidad.
- Muchas estrategias para decidir la secuencia (probe sequence).

## Probing sequence / secuencia de búsqueda

Para buscar/insertar un elemento  $x$ :

- Comenzar por  $i_0 = h(x) \bmod K$ .
- Si posición ocupada por otra clave, seguir para  $i_1, i_2, \dots$  etc.

# Internal hashing / Open addressing

**Internal hashing:** Si la celda está ya ocupada, buscar otra en el mismo array.

- Internal hashing / Open addressing es más común en la actualidad.
- Muchas estrategias para decidir la secuencia (probe sequence).

## Probing sequence / secuencia de búsqueda

Para buscar/insertar un elemento  $x$ :

- Comenzar por  $i_0 = h(x) \bmod K$ .
- Si posición ocupada por otra clave, seguir para  $i_1, i_2, \dots$  etc.

Módulo  $K$ ,

- **Linear probing:**  $i_1 = i_0 + 1, i_2 = i_1 + 1, \dots$
- **Quadratic probing:**  $i_1 = i_0 + 1, i_2 = i_1 + 2, \dots, i_j = i_{j-1} + j, \dots$
- **Double hashing:**  $\Delta(x) = h_2(x), i_1 = i_0 + \Delta, i_2 = i_1 + \Delta, \dots$



# Secuencia de búsqueda: modelo

El comportamiento de *linear* y *quadratic probing* es complejo:

- **Linear probing** presenta el llamado *primary clustering*, pero aprovecha localidad (memoria cache).



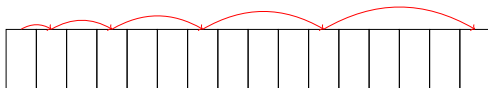
# Secuencia de búsqueda: modelo

El comportamiento de *linear* y *quadratic probing* es complejo:

- **Linear probing** presenta el llamado *primary clustering*, pero aprovecha localidad (memoria cache).



- **Quadratic probing** se comporta inicialmente en modo similar a linear probing, pero luego los saltos aumentan en tamaño.



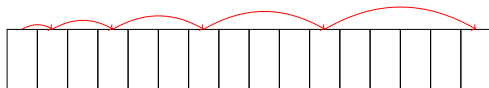
# Secuencia de búsqueda: modelo

El comportamiento de *linear* y *quadratic probing* es complejo:

- **Linear probing** presenta el llamado *primary clustering*, pero aprovecha localidad (memoria cache).



- **Quadratic probing** se comporta inicialmente en modo similar a linear probing, pero luego los saltos aumentan en tamaño.



Modelos simplificados para el análisis:

- **Random probing**: secuencia de búsqueda de números aleatorios uniformes (incluso repetidos).
- **Uniform probing**: secuencia de búsqueda es una permutación  $\pi \in \mathcal{S}_K$  aleatoria.

# Secuencia de búsqueda: modelo

## Parámetros de interés [análisis sin supresiones]

1. Búsqueda exitosa: buscar un elemento presente.
2. Búsqueda no exitosa: buscar un elemento no presente.

# Secuencia de búsqueda: modelo

## Parámetros de interés [análisis sin supresiones]

1. Búsqueda exitosa: buscar un elemento presente.
2. Búsqueda no exitosa: buscar un elemento no presente.

## First Come First Serve (FCFS)

Los elementos se insertan donde termina su búsqueda no exitosa.

Es decir, los elementos ya insertados no se desplazan

En tiempo: Inserción  $n$ -ésima = búsqueda no exitosa con  $n - 1$  elementos

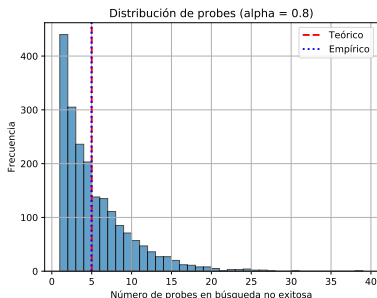
# Random probing: búsqueda no exitosa

Buscar un elemento no presente corresponde a una inserción.

## Teorema

*El costo medio de una búsqueda no exitosa, cuando hay  $n$  elementos, es*

$$U_n = \frac{1}{1 - \alpha}, \quad \alpha = \frac{n}{K}.$$



No hay concentración: ley  $\sim$  geométrica.

# Random probing: búsqueda exitosa

## Teorema

*El costo medio de una búsqueda exitosa es*

$$S_n = \frac{1}{\alpha} \log \left( \frac{1}{1-\alpha} \right) + O(n^{-1}).$$

# Random probing: búsqueda exitosa

## Teorema

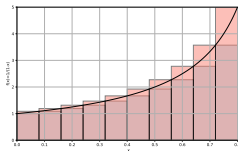
*El costo medio de una búsqueda exitosa es*

$$S_n = \frac{1}{\alpha} \log \left( \frac{1}{1-\alpha} \right) + O(n^{-1}).$$

## Técnica: aproximar sumas con integrales

Si  $f$  es positiva, monótona y acotada en  $[a, b]$ :

$$\sum_{j=a \cdot N}^{b \cdot N-1} f\left(\frac{j}{N}\right) \cdot \frac{1}{N} = \int_a^b f(x) dx + O(N^{-1}).$$





# Uniform hashing

En Uniform Hashing las secuencias de búsqueda son permutaciones de  $\mathcal{S}_K$

- Eliminar la posibilidad de elementos repetidos **no cambia sustancialmente el resultado**.
- Esto es **esperado**: si la secuencia de búsqueda es  $\ll \sqrt{K}$  no esperamos repetidos (**paradoja del cumpleaños**).

# Uniform hashing

En Uniform Hashing las secuencias de búsqueda son permutaciones de  $\mathcal{S}_K$

- Eliminar la posibilidad de elementos repetidos **no cambia sustancialmente el resultado**.
- Esto es **esperado**: si la secuencia de búsqueda es  $\ll \sqrt{K}$  no esperamos repetidos (**paradoja del cumpleaños**).

Teorema (Búsqueda en Uniform hashing, Peterson '57)

*El costo medio de una búsqueda con uniform hashing es*

$$U_n = \frac{K+1}{K-n+1} \sim \frac{1}{1-\alpha}, \quad S_n \sim \frac{1}{\alpha} \log \left( \frac{1}{1-\alpha} \right).$$

# Linear probing

Linear probing es más complejo

Teorema (Búsqueda en Linear probing, Knuth '63)

$$\text{No exitosa} \sim \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right), \quad \text{Exitosa} \sim \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right).$$

# Linear probing

Linear probing es más complejo

Teorema (Búsqueda en Linear probing, Knuth '63)

$$\text{No exitosa} \sim \frac{1}{2} \left( 1 + \frac{1}{(1 - \alpha)^2} \right), \quad \text{Exitosa} \sim \frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right).$$

Se trata del artículo histórico de Donald. E. Knuth que da origen al área:

## NOTES ON “OPEN” ADDRESSING

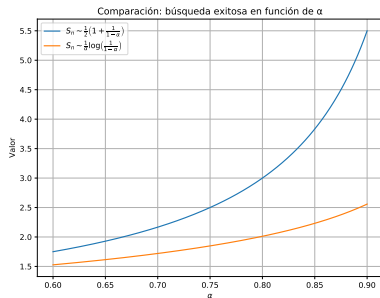
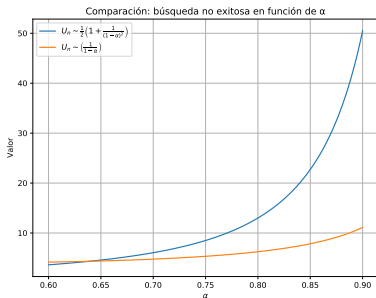
[MY FIRST ANALYSIS OF AN ALGORITHM, ORIGINALLY

DONE DURING SUMMER 1962 IN MADISON.]

DON KNUTH 7/22/63

**1. Introduction and Definitions.** Open addressing is a widely-used technique for keeping “symbol tables.” The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solutions after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.

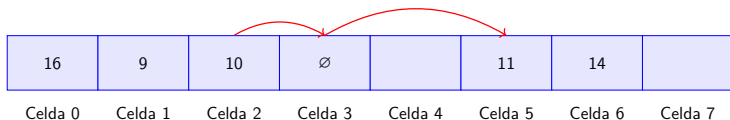
# Comparación de tiempos de búsqueda según $\alpha$



# Y si hay supresiones

Para borrar:

- Introducir *tombstones* (marcas especiales) para indicar que la celda alguna vez fue ocupada,



[las flechas rojas son solo indicativas]

- Las tombstones ocupan una celda, y se cuenta para los rehashings.
- Se puede insertar un elemento en un tombstone.

# Para aprender más



*Donald E. Knuth,*

The Art of Computer Programming, Vol. 3: Sorting and Searching.



*Donald E. Knuth*

Notes on Open Addressing.

<https://jeffe.cs.illinois.edu/teaching/datastructures/2011/notes/knuth-OALP.pdf>



*Conrado Martínez, Cyril Nicaud y Pablo Rotondo*

Mathematical models to analyze Lua hybrid tables.

Preprint <https://arxiv.org/abs/2208.13602>