Análisis Probabilístico de Algoritmos

Pablo Rotondo LIGM, Université Gustave Eiffel

ECI, **Buenos Aires**, 28 de Julio a 1 de Agosto, 2025.

Modalidad del curso

- Curso divido en 3 grandes módulos temáticos

- Cada clase estará divida 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/`

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[1]:
    - Método recursivo.
    - Boltzmann samplers.

1. Introducción al análisis de algoritmos

Introducción: análisis de algoritmos
Estudiar teóricamente la performance de un algoritmo:

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

$\Rightarrow$ 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 $n \to \infty$.
  Por ejemplo $O(n^2), O(n \log n)$, etc.

Ejemplo 1. 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 $\mathbb{E}[\mathbf{1}_A] = \Pr(A)$.

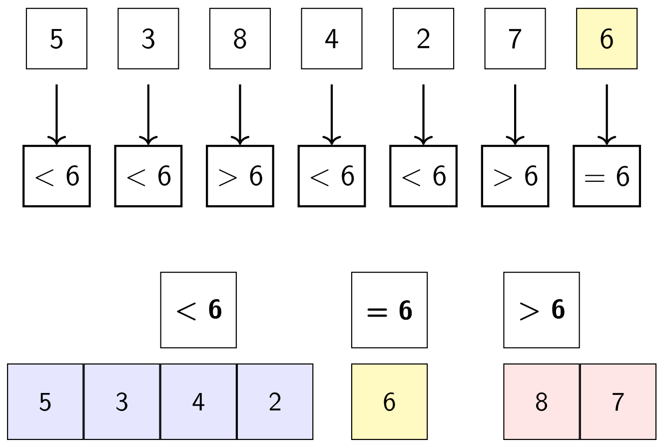
Fórmula de la probabilidad total
Sean eventos $S_1, S_2, \ldots$ disjuntos con $\bigcup_i S_i = \Omega$ (todo) :

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

[1]Si tiempo.

1.1. Algoritmos de sorting

Quicksort



Quicksort particiona según un pivot y luego continua recursivamente.

Quicksort

Por simplicidad[2] 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 $= 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = \Theta(n^2)$.
- Puede suceder si el array está ya ordenado !

Quicksort: modelo aleatorio

Veamos ahora qué sucede si el array es una permutación aleatoria

- cada permutación π de $(1, 2, \ldots, n)$ tiene probabilidad $p(\pi) = 1/n!$
- equivalente a elegir n números aleatorios del intervalo $[0, 1]$
  $\Longrightarrow$ 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 $\Pr(C_n > \lambda) = \sum_{\pi \in \mathcal{S}_n : C_n(\pi) > \lambda} p(\pi)$.

[2] Mejor sería un pivot aleatorio, o permutar la entrada para evitar ataques.

Quicksort: comportamiento en media
Para la media E_n de la cantidad de comparaciones C_n tenemos:

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

Entonces $E_n \sim 2n \log n$ donde $\log$ es el logaritmo natural (neperiano). [3]

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 $j+1, \ldots, n$ en el orden inicial. Su distribución es la misma que C_{n-j}.
Tenemos entonces

$$nE_n = 2\sum_{j=0}^{n-1} E_j + n(n-1)\,, \quad (n-1)E_{n-1} = 2\sum_{j=0}^{n-2} E_j + (n-1)(n-2)\,.$$

Restando las ecuaciones $nE_n - (n-1)E_{n-1} = 2E_{n-1} + 2(n-1)$ i.e., $nE_n = (n+1)E_{n-1} + 2(n-1)$.
Así $\frac{1}{n+1}E_n = \frac{1}{n}E_{n-1} + \frac{2(n-1)}{n(n+1)} = \frac{1}{n}E_{n-1} + \frac{2}{n+1} - \frac{2}{n(n+1)} = \frac{1}{n}E_{n-1} + \frac{4}{n+1} - \frac{2}{n}$. Sumando de 1 a n, $\frac{1}{n+1}E_n = 4H_{n+1} - 4 - 2H_n = 2H_n - 4\frac{n}{n+1}$, lo cual prueba la proposición. □

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

Teorema 1 (Ley de los grandes números)**.** *Si $X_1, X_2, \ldots$ son independientes e identicamente distribuidas, con $\mathbb{E}[|X_1|] < \infty$, entonces con probabilidad 1 :*

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

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

[3]Las sumas harmónicas satisfacen $H_n \sim \int_1^n \frac{dx}{x} = \log n$.

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,

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

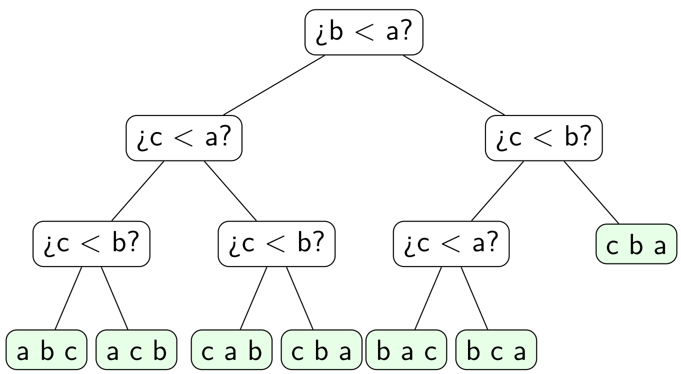
Probaremos más tarde que la cantidad de comparaciones C_n en quicksort[4] satisface $C_n \sim 2n\log n$ en probabilidad.

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

Las funciones generatrices nos ahorrarán muchos cálculos.

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 $\log_2(n!)$,
- $\log_2(n!) \sim n\log_2 n$

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

Teorema 2 (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 $\mathbb{E}[C_n] \geq \log_2 n!$.

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

Prueba: entropía es cota inferior

Lema 1. *Para un árbol binario completo* $\sum_{h \text{ hojas}} 2^{-\ell_h} = 1$

Lema 2. *Para todo* $x > 0$, $\log x \leq x - 1$ *. La igualdad se verifica sii* $x = 1$*.*

Prueba del Teorema. Usando las propiedades del logaritmo:

$$-\sum_{\pi \in \mathcal{S}_n} \ell_\pi p(\pi) = \sum_{\pi \in \mathcal{S}_n} \log_2 p(\pi) \cdot p(\pi) + \sum_{\pi \in \mathcal{S}_n} \log_2 (2^{-\ell_\pi}/p(\pi)) \cdot p(\pi)\,.$$

Gracias a nuestros lemas,

$$\sum_{\pi \in \mathcal{S}_n} \log_2 (2^{-\ell_\pi}/p(\pi)) \cdot p(\pi) \leq \frac{1}{\log 2} \sum_{\pi \in \mathcal{S}_n} (2^{-\ell_\pi}/p(\pi) - 1) \cdot p(\pi) = 0\,,$$

y esto demuestra la proposición. □

QuickSort: modelo del input

- En nuestro modelo de quicksort el input π 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

Entropía de runs

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

Teorema 3. *El costo* C *de cualquier algoritmo basado en la fusión de runs*[5] *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* π*.*

Demostración.

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

[5] Sin contar la detección de runs.

Entropía de runs

Teorema 4. *El costo C de cualquier algoritmo basado en la fusión de runs satisface*

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

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

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

Proposición 3. *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 5 (Barbay, Navarro,'13)**.** *Sea $\mathcal{C} = \mathcal{C}(a_1, \ldots, a_r)$ la clase de las permutaciones con runs de largo $a_1, a_2, \ldots, a_r$, con $a_i \geq 2$ para $i = 1, \ldots, r-1$.*

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 π que requiere al menos $\log_2 |\mathcal{C}|$ operaciones.
Se necesita una cota [no trivial[6]] , en este caso $2^{r-1}|\mathcal{C}| \geq \binom{n}{a_1,\ldots,a_r}$. □

TimSort
Tim Peters[7] diseña 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.

TimSort principio e historia

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

[6] Ver referencias, en particular `https://arxiv.org/pdf/1805.08612`
[7] `https://svn.python.org/projects/python/trunk/Objects/listsort.txt`

Invariante inspirado por Fibonacci:

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

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

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

Optimalidad en media y entropía

Teorema 6 (Auger, Jugé, Nicaud, Pivoteau '18)**.** *En el peor caso TimSort es* $1{,}5\,n\mathcal{H} + O(n)$.

TimSort no es óptimo

Teorema 7 (Wild,Munro'18)**.** *En el peor caso PowerSort es* $n\mathcal{H} + O(n)$.

Una permutación aleatoria (típica) tiene muchos runs cortos!

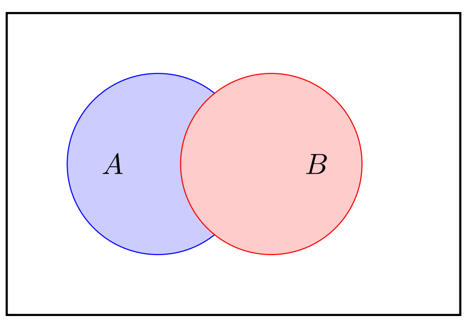
$$n = 20: \quad [11, 18, 1, 5, 2, 14, 20, 3, 8, 15, 6, 4, 16, 17, 13, 10, 19, 9, 7, 12]\,.$$

$$\mathcal{H} = 3{,}1\ldots\,, \qquad \log_2 20 = 4{,}3\ldots\,.$$

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:

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

Proposición 4.

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

Demostración. Notar que $e^k = \sum_{i=0}^{\infty} k^i/i! \geq k^k/k!$. □

Entropía de corridas de permutación aleatoria
Utilizando

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

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

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

Corolario
Con alta probabilidad, para una permutación aleatoria uniforme[8],
$\mathcal{H} \ge \sum \frac{A_i}{n} \log_2\left(\frac{n}{2(\log n)/\log\log n}\right) = \log_2 n + O(\log\log n)\,, \quad \mathcal{H} \le \log_2 n\,.$

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

Con alta probabilidad y en media
Probamos que, con probabilidad $p \to 1$,

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

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

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

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

Para la cota superior tenemos suerte: $\mathcal{H} \le \log_2(n)$ 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
```

[8]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.

```
        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 $X_1, X_2, \ldots$ de números iid de $[0,1]$.

(a) Probar $runs(X_1, \ldots, X_{i+j}) \leq runs(X_1, \ldots, X_i) + runs(X_{i+1}, \ldots, X_{i+j})$.

(b) Probar que $e_k := \mathbb{E}[runs(X_1, \ldots, X_k)]$ satisface $e_{i+j} \leq e_i + e_j$ para todo $i, j \geq 0$. Concluir que $e_k/k \to c$ para cierta $c \geq 0$.

(c) Mostrar que la constante es positiva $c > 0$.

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`

1.2. Tablas de Hash

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*:

[8] *Pista (b).* Lema de Fekete...
[8] *Pista (c).* ¿Qué podemos decir si $X_i < X_{i+1}$ y $X_{i+1} > X_{i+2}$?

- **Idea** : utilizar un array A pequeño, de tamaño $K \ll |\mathcal{U}|$
  - considerar una función $h : \mathcal{U} \to \mathbb{Z}$ *pseudo-aleatoria*,
  - insertar k en $A[i]$ donde $i = h(k) \operatorname{mód} K$. [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[9] sea mayor que 1/2?.

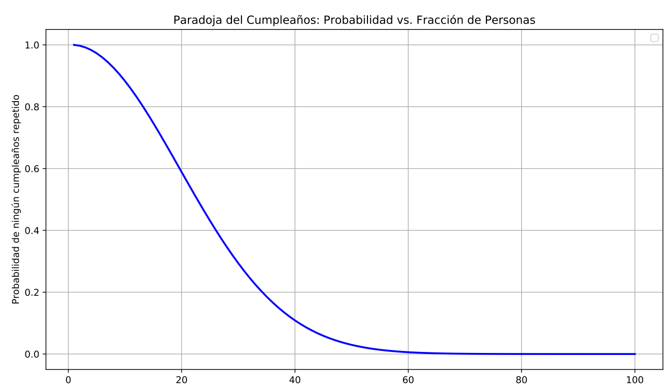
Más en general, ¿cuántas personas para la primera "colisión"?

- 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 $q_n = 1 - p_n$.

[9]solo el día del año, no el año

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

- Usando la desigualdad $1 + x \leq e^x$, valida para $x \in \mathbb{R}$,

$$p_n \leq \exp\left(-\sum_{i=1}^{n-1} \frac{i}{K}\right) \leq \exp\left(-\frac{(n-1)^2}{2K}\right).$$

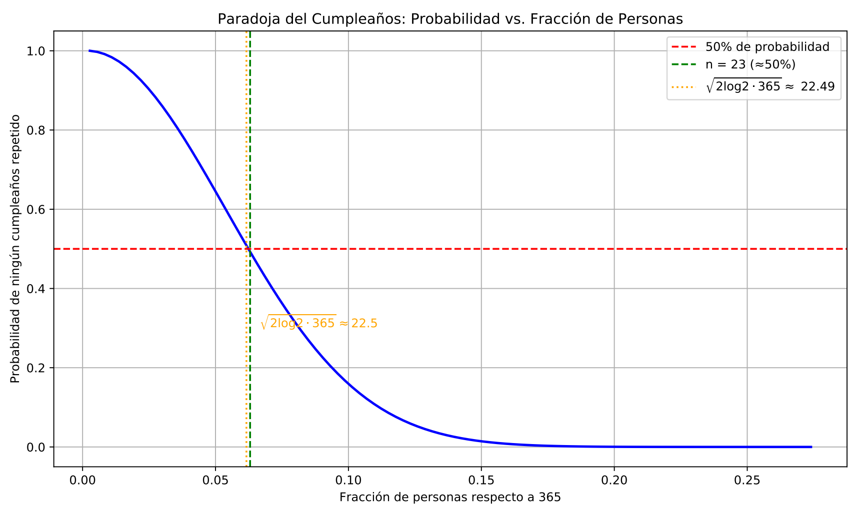
- Usando la desigualdad $1 + x \geq e^{x - x^2/2}$, valida para $x \in [0,1]$,

$$p_n \geq \exp\left(-\sum_{i=1}^{n-1} \frac{i}{K} - \frac{1}{2}\sum_{i=1}^{n-1} \frac{i^2}{K^2}\right) \geq \exp\left(-\frac{n^2}{2K} - \frac{n^3}{2K^2}\right).$$

Proposición 5. *Considerando $n \sim \sqrt{2\theta K}$ con $K \to \infty$, $p_n \sim e^{-\theta}$.*

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

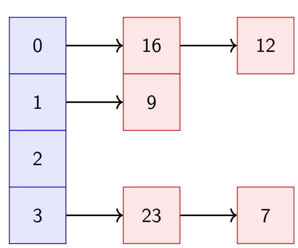
La paradoja del cumpleaños
Ilustración de la aproximación: $n \sim \sqrt{2\theta K}$, $p_n \sim e^{-\theta}$ con $\theta = \log 2$



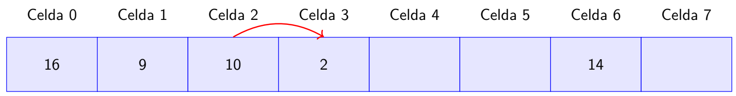
Tablas de Hash
Tablas de Hash requieren un mecanismo de resolución de colisiones:

- **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.



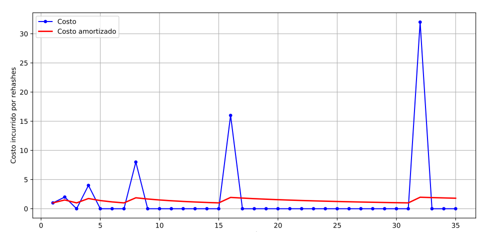
[la flecha roja es indicativa]

- **Política de rehashing :**
  - si tasa de ocupación[10] del array A es alta, nuevo array de tamaño mayor,
  - necesario re-insertar todo. [paso lento!]

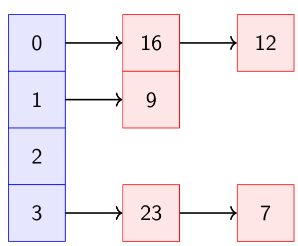
Rehashing
Cuando el load factor $\alpha = n/K$ excede un valor dado γ (p.e., $\gamma = 0{,}85$), considerar un array nuevo con capacidad[11] $K' = 2K$.

Proposición 6. *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 $\frac{1}{T}\sum_{t=1}^{T} c_t$.



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



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

Proposición 7 (Lookup)**.** *Con alta probabilidad* $(K \to \infty)$, *ninguna lista tiene longitud mayor que* $2\frac{\log K}{\log\log K}$.

[10] "Load factor" en inglés.
[11] Aquí no consideramos que el tamaño puede reducirse.

External hashing

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

Proposición 8 (Lookup)**.** *Con alta probabilidad* ($K \to \infty$)*, ninguna lista tiene longitud mayor que* $2\frac{\log K}{\log\log K}$*.*

Demostración. Sea $X_1, .., X_n$ la secuencia de células elegidas para las n inserciones.

Consideremos la célula C_0, y sea $C_0(n)$ la lista luego de n inserciones. Por el union-bound su longitud satisface:

$$\Pr(|C_0(n)| \ge m) \le \sum_{i_1<..<i_m} \Pr(X_{i_1} = \ldots = X_{i_m} = 0) = \binom{n}{m} K^{-m} .$$

Y, nuevamente por el union-bound,

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

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

Observamos que

$$\binom{n}{m} = \frac{n \cdot \ldots \cdot (n-m+1)}{m!} \le \frac{n^m}{m!} , \qquad \tfrac{m^m}{m!} \le \sum_{k=0}^{\infty} \tfrac{m^k}{k!} = e^m .$$

Deducimos, recordando que $n/K \le \gamma$,

$$\begin{aligned} P_m \le K \times \frac{n^m}{(m/e)^m} K^{-m} &\le K \times \frac{\gamma^m}{(m/e)^m} \\ &= \exp(\log K + m\log\gamma + m - m\log m) . \end{aligned}$$

Tomando[12] $m = 2\frac{\log K}{\log\log K}$,

$$\log K + m\log\gamma + m - m\log m = -\log K + o(\log K) \to -\infty . \qquad \square$$

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/inserir un elemento x:

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

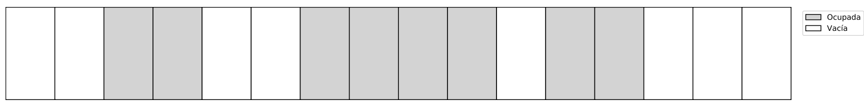
Módulo K,

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

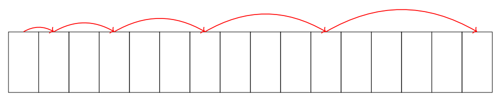
[12] $f(m) := m\log\gamma + m - m\log m$ es decreciente si $m \ge \gamma$.

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.

First Come First Serve (FCFS)
Los elementos se insertan donde termina su búsqueda no exitosa.

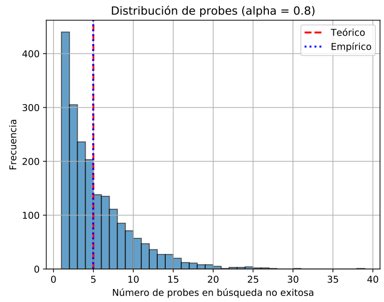
Es decir, los elementos ya insertados no se desplazan

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

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



No hay concentración: ley ~ geométrica.

Random probing: búsqueda exitosa

Teorema 9. *El costo medio de una búsqueda exitosa es*

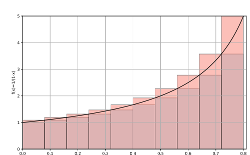
$$S_n = \tfrac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right) + O(n^{-1}) .$$

Técnica: aproximar sumas con integrales
Si f es positiva, monótona y acotada en $[a,b]$:

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

La prueba de la fórmula se sigue de

$$\sum_{j=A}^{B-1} f(\tfrac{j}{N}) \cdot \tfrac{1}{N} \le \int_A^B f(x)dx \le \sum_{j=A+1}^{B} f(\tfrac{j}{N}) \cdot \tfrac{1}{N} .$$



Demostración. Notamos que $S_n = \frac{1}{n}\sum_{k=0}^{n-1} U_k$. En efecto, U_k es el costo de buscar el $(k+1)$-ésimo elemento insertado, y $1/n$ es la probabilidad de buscar éste último.
Entonces

$$S_n = \tfrac{1}{n}\sum_{k=0}^{n-1} U_k = \tfrac{1}{n}\sum_{k=0}^{n-1} \frac{1}{1-k/K} .$$

Usando la técnica de sumas e integrales

$$nS_n \le K \int_0^{n/K} \frac{dx}{1-x} = K\log(\tfrac{1}{1-\alpha}) ,$$

y también

$$(1 - \tfrac{1}{1-\alpha}) + K\log(\tfrac{1}{1-\alpha}) \le nS_n .$$

Como $\frac{n}{K} \le \alpha \le \gamma < 1$ el término $(1 - \frac{1}{1-\alpha})$ está acotado y obtenemos el resultado dividiendo por n. $\square$

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 10 (Búsqueda en Uniform hashing, Peterson '57)**.** *El costo medio de una búsqueda con uniform hashing es*

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

Linear probing

Linear probing es más complejo

Teorema 11 (Búsqueda en Linear probing, Knuth '63)**.**

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

El punto clave del análisis es el siguiente lema:

Lema 3. *La probabilidad de tener las celdas $C[0]$ y $C[k+1]$ vacías y $C[1],\ldots,C[k]$ ocupadas es:*

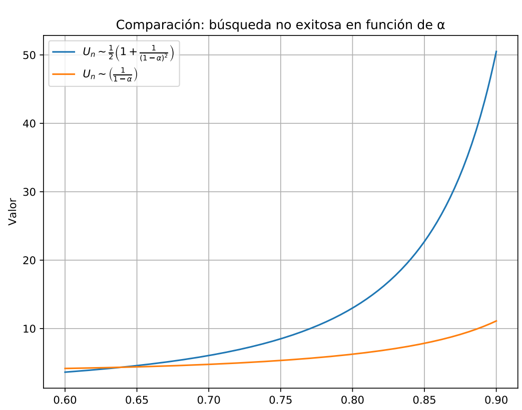
$$\frac{1}{K^n}\binom{n}{k}(k+1)^k\left(1-\frac{k}{k+1}\right)(K-k-1)^{n-k}\left(1-\frac{n-k}{K-k-1}\right)$$

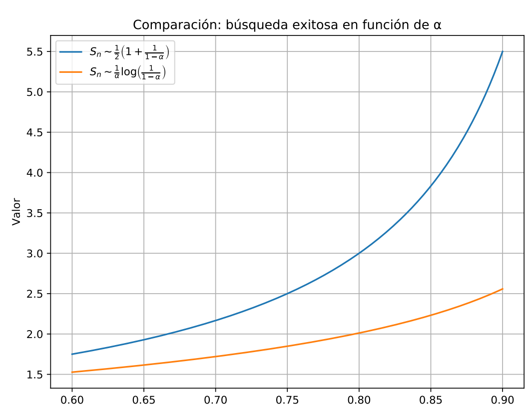
Demostración. Sea $f(M,r)$ la cantidad de secuencias de r inserciones (eligiendo sus hashes) en una tabla de M entradas $0,1,\ldots,M-1$, tales que la posición 0 resta vacía al final.
En tal situación, por simetría circular de Linear Probing (la proba. es igual para todos), la probabilidad de que al final la posición 0 esté vacía es $1-\frac{r}{M}$ y tenemos $f(M,r) = M^r \cdot \left(1-\frac{r}{M}\right)$
La probabilidad que buscamos es

$$\binom{n}{k} f(k+1,k)\,f(K-k-1,n-k)\,,$$

ya que hay que elegir cuáles de las n inserciones van al primer segmento de 0 a k (por eso la binomial). $\square$

Comparación de tiempos de búsqueda según α

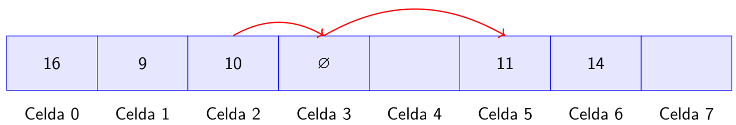




Y si hay supresiones

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`