# Ordenamiento (*sorting*)

Ariel Bendersky[1,2], Nicolás D'Ippolito[1,2]

[1]ICC, UBA-CONICET
[2]Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Buenos Aires, Argentina

Algoritmos y Estructuras de Datos II
Segundo cuatrimestre de 2019

# (2) Ordenar

- ¿En qué situaciones es necesario ordenar información?
- ¿Con qué objetivos?
- Problema clásico, muy estudiado.

# (3) Suposiciones

- El problema de ordenamiento trabaja sobre un array de $A[0 \ldots n-1]$ de elementos,
- que tienen una clave $\alpha_i$,
- tal que existe una relación de orden $<_\alpha$.
- Se busca que $A$ contenga una permutación de su contenido inicial tal que $\forall i < n-1, A[i] <_\alpha A[i+1]$.
- Ojo: sorting *in place*.

# (4) El método más intuitivo

- Recorro el arreglo y en cada posición $i$:
  - Seleccionar el mínimo elemento que se encuentra entre la posición actual y el final.
  - Ubicarlo en la posición $i$, intercambiándolo con el ocupante original de esa posición.
- Se llama Selection Sort.
- ¿Cuál es el invariante?
  - Los elementos entre la posición 0 y la posición $i$ son los $i + 1$ elementos más pequeños del arreglo original, y
  - los elementos entre la posición 0 y la posición $i$ se encuentran ordenados,

# (5) Complejidad

- ¿Cuántas operaciones realiza?
  - Hace $n$ swaps.
  - ¿Y cuántas comparaciones?
  - Hay $n-1$ ejecuciones del ciclo principal.
- En la $i$-ésima iteración hay que encontrar el mínimo de entre $n-i$ elementos y por lo tanto necesitamos $n-i-1$ comparaciones.
- $\sum_{i=0}^{n-2}(n-i-1) = \sum_{i=1}^{n-1} = \frac{n(n-1)}{2}$
- Observar que el costo no depende del eventual ordenamiento parcial o total del arreglo.

# (6) Insertion Sort

- En Selection Sort teníamos una parte ya ordenada (digamos $0 \ldots k$) y la extendíamos buscando el elemento que seguía para ubicar en la posición $k + 1$.
- Este otro algoritmo propone tomar $A[k + 1]$ y ver qué lugar le corresponde dentro del rango $A[0 \ldots k + 1]$.
- Algoritmo:

  Recorro el arreglo con $i$ entre 0 y $n - 1$:

  // Inserto $A[i]$ entre las posiciones 0 e $i - 1$, es decir:
  *elem_a_insertar* := $A[i]$
  $j := i - 1$
  mientras ($j >= 0$ y $A[j] >$ *elem_a_insertar*)
  $A[j + 1] := A[j]$ // Voy desplazando.
  $j - -$
  // Al salir del ciclo $j$ es negativo o $A[j]$ es $\leq$ *elem_a_insertar* y
  // $A[j + 1]$ (si existe) es $\geq$ *elem_a_insertar*.
  $A[j + 1] :=$ *elem_a_insertar*

# (7) Insertion Sort (cont.)

- ¿Cuál es el invariante?
  - Los elementos entre la posición 0 y la posición $i$ son los elementos que ocupaban las posiciones 0 a $i$ del arreglo original, y
  - los elementos entre la posición 0 y la posición $i$ se encuentran ordenados,
  - aunque no en sus posiciones definitivas.
  - El arreglo es una permutación del arreglo original (o sea que los elementos de las posiciones $i + 1$ hasta $n - 1$ son los que ocupaban esas posiciones en el arreglo original).

# (8) Insertion Sort – complejidad

- $n-1$ ejecuciones del ciclo principal.
- En la $i$-ésima iteración hay que ubicar al elemento junto a otros $i-1$ elementos y por lo tanto necesitamos $i-1$ comparaciones.
- Costo: $\sum_{i=1}^{n-1}(i-1) = \frac{(n-1)(n-2)}{2}$
- Complejidad: $O(n^2)$
- ¿Y si está parcialmente ordenado?
- ¿Y si está ordenado al revés?

# (9) Insertion Sort – otras características

- ¿De qué me sirve la parte del invariante que dice "pero no en sus posiciones definitivas"?
- Es estable.
- Un algoritmo es estable si mantiene el orden anterior de elementos con igual clave.
- ¿Para qué sirve la estabilidad?
- ¿Son estables los algoritmos que vimos hasta ahora?

# (10) MergeSort

- ¿Podemos ordenar mediante D&C?
- Algoritmo:

  Si $n < 2$, ya está ordenado.

  Si no,

  Dividir el arreglo en dos mitades.
  Ordenarlas recursivamente.
  Combinarlas mediante un *merge*.
- Complejidad?
- $T(n) = 2T(n/2) + n$
- Es decir, $T(n) = \Theta(n \log n)$.
- Lo veremos más en detalle la última clase.

# (11) QuickSort

- Debido a C.A.R. Hoare
- Muy estudiado, analizado y utilizado en la práctica.
- Idea:
  - Supongamos que conocemos el elemento mediano del arreglo.
  - Separar el arreglo en dos mitades: los elementos menores que el mediano por un lado y los mayores por el otro.
  - Ordenar las dos mitades
  - ¡Y listo!
- ¿Y si no conocemos el mediano? ¡Lo buscamos!

# (12) QuickSort

- QuickSort(A, inicio, fin)

  if (fin-inicio<2) return

  pivot:= elegir_pivot(A, inicio, fin)

  nueva_posición_pivot:= separar(A, pivot, inicio, fin)

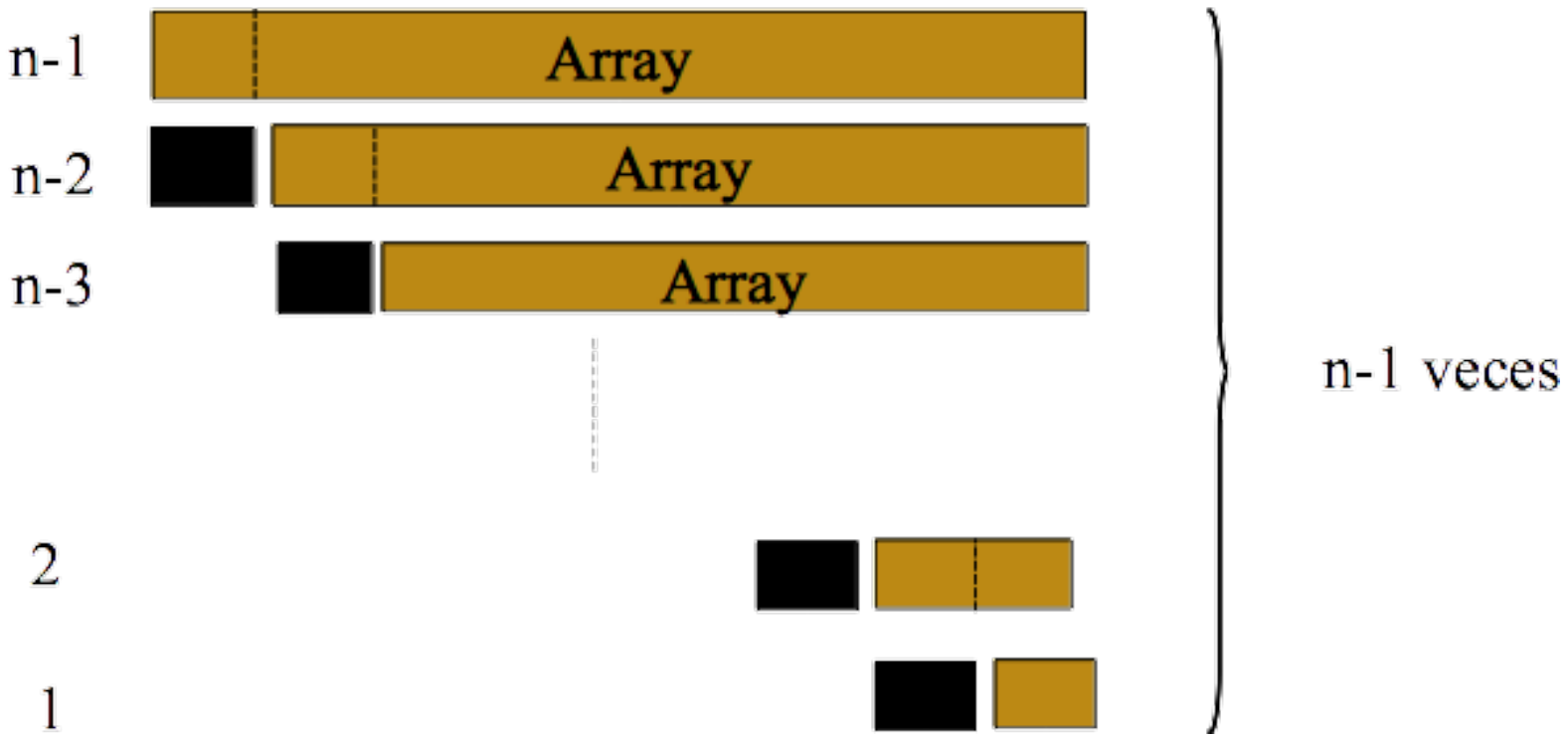
  QuickSort(A, inicio, nueva_posición_pivot)

  QuickSort(A, nueva_posición_pivot+1, fin)
- La elección del pivote es clave.
- Si elijo el del medio, la separación puede llegar a $O(n^2)$.
- Si elijo mejor, me toma más tiempo pero la separación puede ser más eficiente.

# (13) QuickSort (cont.)

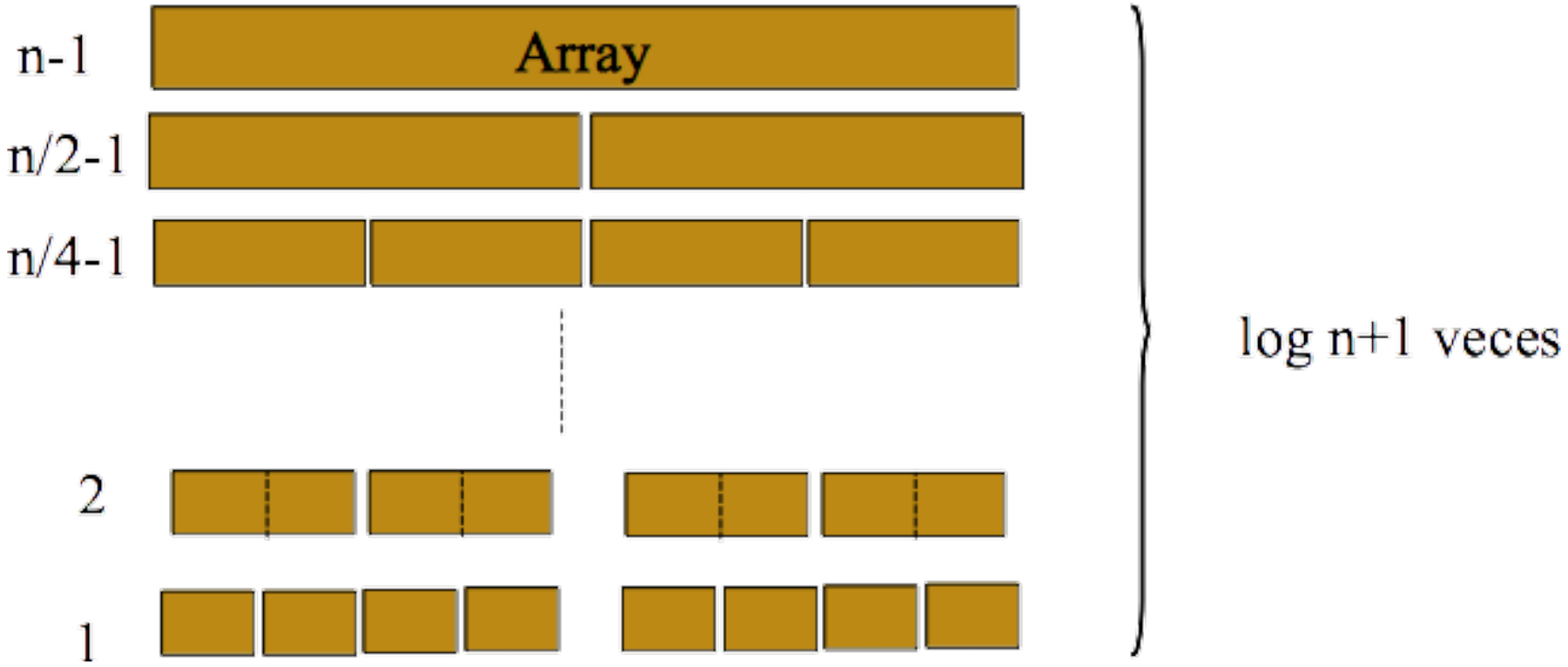
- ¿Cuál es su complejidad?
- Peor caso: $O(n^2)$
- Caso promedio: $O(n \log n)$ (demo igual a MergeSort, por TM).
- Mejor caso: $O(n \log n)$.
- En la práctica es muy eficiente.

# (14) QuickSort, peor caso




- El elemento pivot es siempre el mínimo.
- Costo $= O(n-1+n-2+\ldots+2+1) = O(n^2)$

# (15) QuickSort, mejor caso




- $n$ potencia de 2 por simplicidad.
- Costo $= n + 2\frac{n}{2} + 4\frac{n}{4} + \ldots + n\frac{n}{n}$
- $= \sum_{i=0}^{\log n} 2^i \frac{n}{2^i} = n \log(n+1)$

# (16) Cota inferior

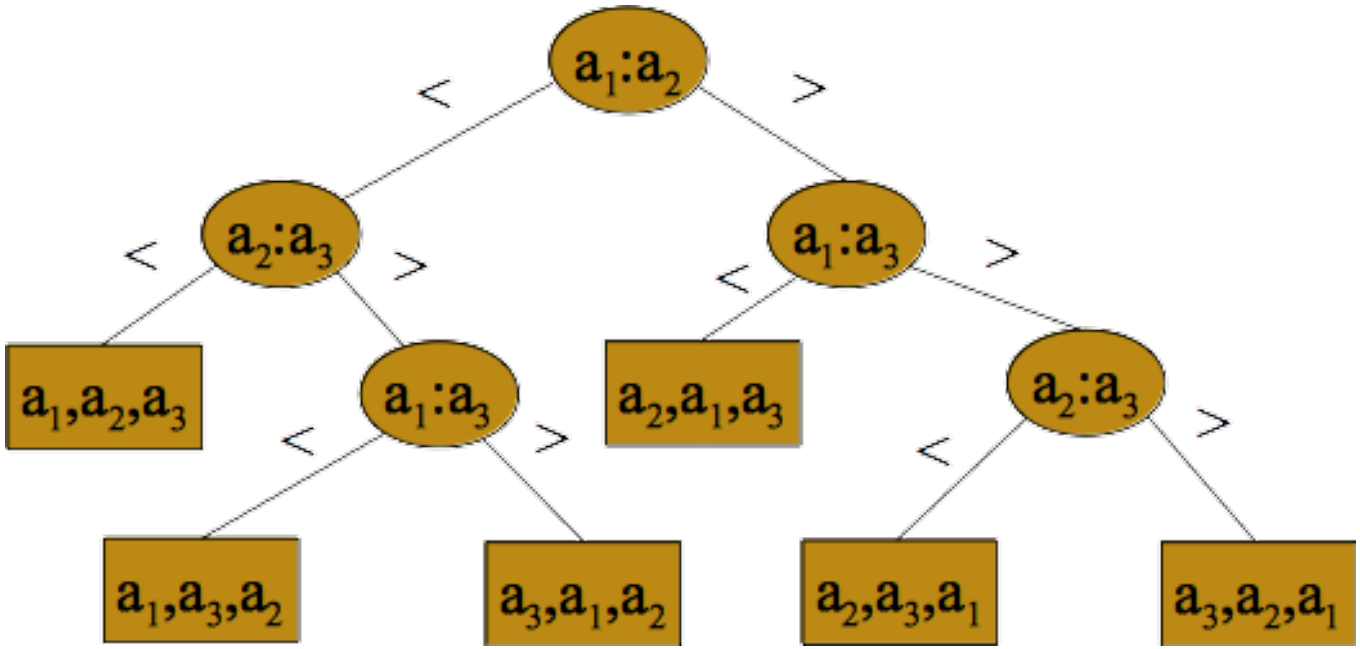
- ¿Cuánto nos toma ordenar?
- ¿Podremos hacerlo en menos de $O(n \log n)$ en el peor caso?
- ¿Cómo podemos analizar eso?

# (17) Formalicemos

- Observación fundamental: todos los algoritmos deben comparar elementos (es decir, ese es nuestro modelo de cómputo).
- Dados $a_i$ y $a_k$, tres casos posibles: $a_i < a_k$, $a_i > a_k$, o $a_i = a_k$.
- Supongamos por simplicidad que todos los $a_i$ son distintos.
- Por ende, supongamos también que todas las comparaciones tienen la forma $a_i < a_k$.
- Nota: si los elementos pueden tener valores iguales entonces se consideran solamente comparaciones del tipo $a_i \leq a_k$.

# (18) Árboles de decisión

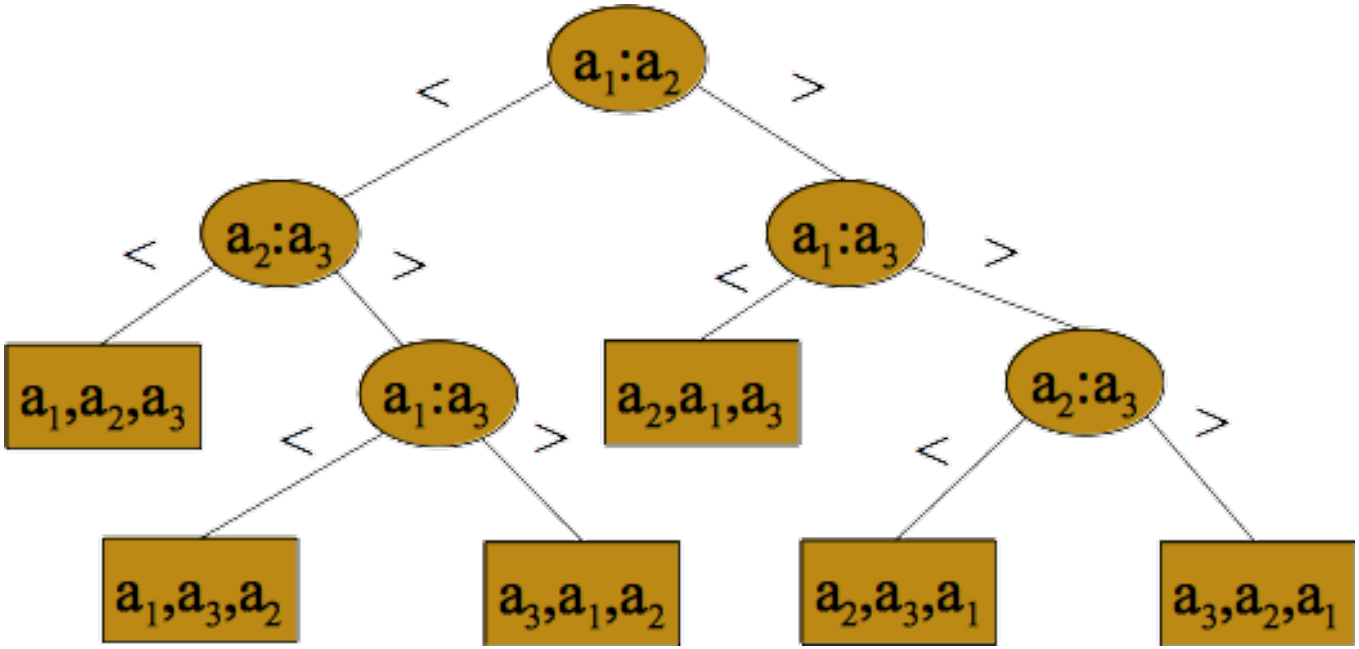
Árbol de decisión para el conjunto $\{a_1, a_2, a_3\}$




- Un árbol de decisión representa las comparaciones ejecutadas por un algoritmo sobre un input dado.
- Cada hoja corresponde a una de las posibles permutaciones.

# (19) Árboles de decisión (cont.)

Árbol de decisión para el conjunto $\{a_1, a_2, a_3\}$




- Hay $n!$ posibles permutaciones.
- Por ende, el árbol debe contener $n!$ hojas.
- La ejecución de un algoritmo corresponde a un camino en el árbol de decisión correspondiente al input considerado.

# (20) Árboles de decisión (cont.)

- El camino más largo de la raíz a una hoja (altura) representa el número de comparaciones que el algoritmo tiene que realizar en el caso peor.
- Teorema: cualquier árbol de decisión que ordena $n$ elementos tiene altura $\Theta(n \log n)$.
- Demostración (esquema):
  1) Un árbol de decisión es un árbol binario.
  2) Con $n!$ hojas.
  3) Altura mínima de un árbol binario de $x$ hojas: $\Theta(\log x)$.
  4) De 1, 2 y 3, sale que la altura mínima de un árbol de decisión es $\Theta(\log n!) = \Theta(n \log n)$.

# (21) Árboles de decisión (cont.)

- Corolario: *ningún* algoritmo de ordenamiento tiene complejidad mejor que $\Theta(n \log n)$.
- Corolario: el algoritmo MergeSort tiene complejidad asintótica óptima.
- ¿Ninguno?
- Notar que existen algoritmos de ordenamiento con complejidad más baja, pero requieren ciertas hipótesis extra sobre el input.
- Tarea: leer de los libros: Bubble Sort ($O(n^2)$), Counting Sort, Bucket Sort y Radix Sort ($O(n)$, no basados en comparaciones pero con requerimientos y suposiciones particulares).

# (22) Hoy vimos

- Distintos algoritmos para ordenar.
- Los más simples (insertion y selection) tardan $O(n^2)$.
- Mergesort y Quicksort tardan $O(n \log n)$.
- Vimos que esa complejidad es óptima.
- Lo que viene: Colas de prioridad.