

# QuickSort

Clase 04

IIC 2133 - Sección 4

Prof. Sebastián Buggedo

# Sumario

**Obertura**

Particiones

Quicksort

Epílogo

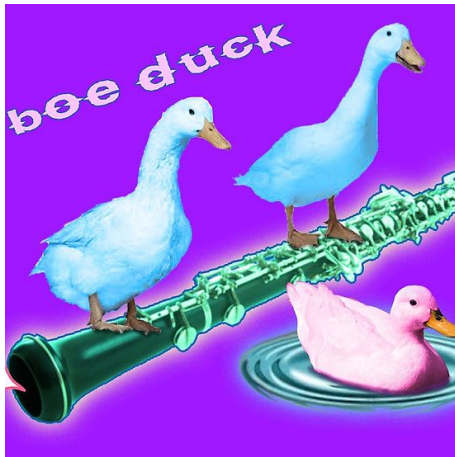


# Primer Acto: Ordenación y fundamentos

## Ordenación y análisis de algoritmos



# Playlist 1



Playlist del curso: "DatiWawos Primer Acto"

# Repaso: Dividir para conquistar

El plan para usar Merge en un algoritmo de ordenación sigue la estrategia **dividir para conquistar**

La estrategia sigue los siguientes pasos

1. Dividir el problema original en dos (o más) **sub-problemas** del mismo tipo
2. Resolver **recursivamente** cada sub-problema
3. Encontrar solución al problema original **combinando** las soluciones a los sub-problemas

Los sub-problemas son instancias más pequeñas del problema a resolver

# Repaso: El algoritmo MergeSort

**input** : Secuencia  $A$

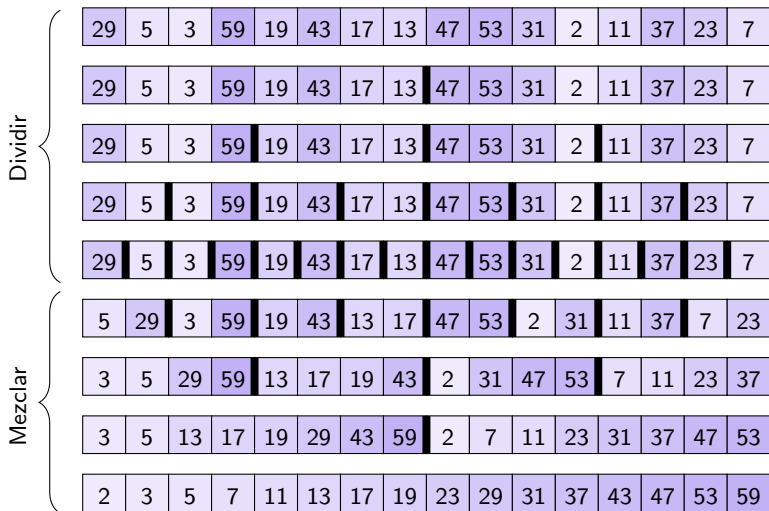
**output**: Secuencia ordenada  $B$

MergeSort ( $A$ ):

- 1    **if**  $|A| = 1$  : **return**  $A$
- 2    Dividir  $A$  en mitades  $A_1$  y  $A_2$
- 3     $B_1 \leftarrow \text{MergeSort}(A_1)$
- 4     $B_2 \leftarrow \text{MergeSort}(A_2)$
- 5     $B \leftarrow \text{Merge}(B_1, B_2)$
- 6    **return**  $B$



## MergeSort: Ejemplo de ejecución





# Repaso: Complejidad de MergeSort

Para el análisis de complejidad de tiempo, definimos

$$T(n) := \# \text{ pasos para ordenar } n \text{ elementos}$$

Con esto, consideramos los dos casos posibles al llamar a MergeSort

MergeSort ( $A$ ):

```
1  if  $|A| = 1$  : return  $A$ 
2  Dividir  $A$  en  $A_1$  y  $A_2$ 
3   $B_1 \leftarrow \text{MergeSort}(A_1)$ 
4   $B_2 \leftarrow \text{MergeSort}(A_2)$ 
5   $B \leftarrow \text{Merge}(B_1, B_2)$ 
6  return  $B$ 
```

- Si  $n = 1$ , aplica el caso base y solo involucra un paso

$$T(1) = 1$$

- Si  $n > 1$ , aplican los llamados
  - Dos llamados de tamaño  $n/2$
  - Llamado a Merge

$$T(n) = 2T(n/2) + n$$

Este análisis aplica **para toda** secuencia de input:  
Nos entregará el resultado de peor, mejor y caso promedio

# Repaso: Complejidad de MergeSort

La siguiente relación es una **relación de recurrencia**

$$T(1) = 1, \quad T(n) = 2T\left(\frac{n}{2}\right) + n$$

Podemos resolverla notando que la parte recursiva puede ser reescrita como

$$T(n) = 2T(n/2) + n \quad \Leftrightarrow \quad \frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

La gracia de esta expresión es que numeradores y denominadores incluyen la misma fracción de  $n$

Sin pérdida de generalidad, suponemos que  $n$  es potencia de 2

## Repaso: Complejidad de MergeSort

Construimos un sistema de ecuaciones reemplazando el argumento del lado izquierdo por  $n, n/2, n/4, \dots, 2$  de forma que el último término contiene  $T(1)$  (nuestro caso base)

$$\text{ecuación 1} \quad \frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

$$\text{ecuación 2} \quad \frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$$

...

$$\text{ecuación } k \quad \frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Como el lado derecho de la  $i$ -ésima ecuación considera la potencia  $2^i$ , de la  $k$ -ésima ecuación deducimos

$$1 = \frac{n}{2^k} \Rightarrow 2^k = n \Rightarrow k = \log(n)$$

# Repaso: Complejidad de MergeSort

Sumamos las  $\log(n)$  ecuaciones y simplificamos los términos que aparecen a ambos lados

$$\text{ecuación 1} \quad \frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1$$

$$\text{ecuación 2} \quad \frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1$$

...

$$\text{ecuación } k \quad \frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

$$\text{suma} \quad \frac{T(n)}{n} = \frac{T(1)}{1} + \log(n)$$

Despejando, obtenemos  $T(n) = n \log(n) + n$

La complejidad de tiempo de MergeSort es  $\mathcal{O}(n \log(n))$

# Complejidad de MergeSort

En términos de memoria adicional

MergeSort ( $A$ ):

```
1  if  $|A| = 1$  : return  $A$ 
2  Dividir  $A$  en  $A_1$  y  $A_2$ 
3   $B_1 \leftarrow \text{MergeSort}(A_1)$ 
4   $B_2 \leftarrow \text{MergeSort}(A_2)$ 
5   $B \leftarrow \text{Merge}(B_1, B_2)$ 
6  return  $B$ 
```

- El paso recursivo no ocupa memoria adicional
- Para  $|A| = n$ , la línea 5 ocupa  $\mathcal{O}(n)$
- Ojo! Los llamados recursivos no van acumulando memoria reservada, por lo que no sumamos  $\mathcal{O}(n)$  por llamado

La complejidad de memoria de MergeSort es  $\mathcal{O}(n)$

# Complejidad de algoritmos de ordenación

Resumimos los resultados de complejidad por caso hasta el momento

Algoritmo	Mejor caso	Caso promedio	Peor caso	Memoria adicional
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Insertion Sort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$
Merge Sort	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n \log(n))$	$\mathcal{O}(n)$
Quick Sort	?	?	?	?
Heap Sort	?	?	?	?

Notemos la mejora en tiempo con MergeSort  
a cambio de memoria adicional

# Dividir para conquistar

La estrategia sigue los siguientes pasos

1. Dividir el problema original en dos (o más) **sub-problemas** del mismo tipo
2. Resolver **recursivamente** cada sub-problema
3. Encontrar solución al problema original **combinando** las soluciones a los sub-problemas

# Búsqueda binaria

El algoritmo de **búsqueda binaria** también está basado en la estrategia dividir para conquistar

**input** : Secuencia  $A[0, \dots, n-1]$ , elemento  $x$ , índices  $i, f$

**output**: Índice  $m \in \{0, \dots, n-1\}$  o  $-1$

BinarySearch ( $A, x, i, f$ ):

```
1  if  $f < i$  : return  $-1$ 
2   $m \leftarrow \left\lfloor \frac{i+f}{2} \right\rfloor$ 
3  if  $A[m] = x$  : return  $m$ 
4  if  $A[m] > x$  :
5      return BinarySearch ( $A, x, i, m-1$ )
6  return BinarySearch ( $A, x, m+1, f$ )
```



# Búsqueda binaria

El algoritmo de **búsqueda binaria** también está basado en la estrategia dividir para conquistar

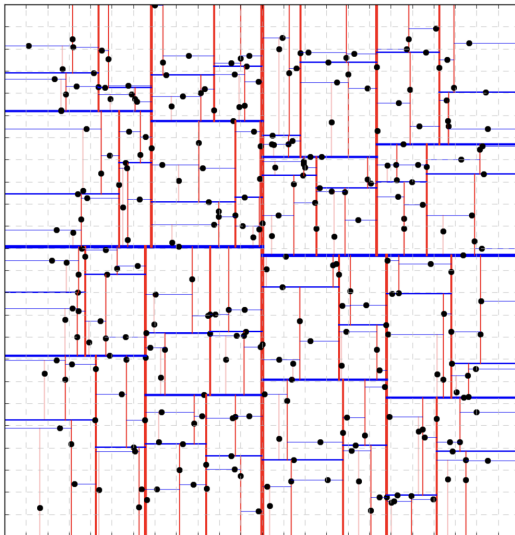
```
BSearch (A, x, i, f):  
1  if f < i : return -1  
2  m ← ⌊  $\frac{i+f}{2}$  ⌋  
3  if A[m] = x : return m  
4  if A[m] > x :  
5      return BSearch (A, x, i, m - 1)  
6  return BSearch (A, x, m + 1, f)
```

- La división en subproblemas se hace escogiendo un **pivote** central (línea 2)
- La naturaleza de la búsqueda hace que sea necesario resolver **solo uno de los subproblemas**
- Solución al problema es exactamente la solución al llamado recursivo

# Un problema para inspirarnos

- Consideremos el problema de procesar un gran conjunto de coordenadas en 2D
- Para repartir la carga, se reparten los datos en zonas rectangulares
- La idea es que los rectángulos **particionen** el espacio 2D
- Además queremos que cada zona tenga la misma cantidad de datos

# Un problema para inspirarnos



# Objetivos de la clase

- ☐ Comprender la estrategia de elegir pivote para particionar una secuencia
- ☐ Identificar cuándo tal estrategia encuentra la mediana
- ☐ Comprender y analizar el algoritmo `Partition`
- ☐ Comprender el uso de `Partition` en `Median` para encontrar el elemento central
- ☐ Comprender el uso de `Partition` para ordenar secuencias

# Sumario

Obertura

**Particiones**

Quicksort

Epílogo

# El desafío de encontrar la mediana

- En el contexto de una secuencia de valores
- ¿Cómo encontramos la mediana?
- Si ordenamos la secuencia: es la mitad...
- ¿Podemos hacerlo sin recurrir a ordenar los datos?

# El desafío de encontrar la mediana

## Definición

Dada una secuencia ordenada de  $n$  valores  $a_0, a_1, \dots, a_{n-1}$ , llamaremos **mediana** al valor  $M_e$  dado por

- Si  $n$  es impar,

$$M_e = a_{\lfloor n/2 \rfloor}$$

- Si  $n$  es par,

$$M_e = \frac{a_{n/2} + a_{n/2-1}}{2}$$

La mediana es tal que en la secuencia hay la misma cantidad de elementos mayores y menores a ella

# Primera estrategia

Dado un elemento de la secuencia al cual llamaremos **pivote**

1. ¿El pivote es la mediana?
2. Si no, ¿la mediana es mayor o menor al pivote?

Si escogemos el pivote de manera aleatoria, ¿cómo respondemos a estas preguntas?

Necesitamos el número de elementos menores y mayores que el pivote



# Primera estrategia...

## Ejemplo

Dada la siguiente secuencia de datos (de largo impar)

7	3	5	2	1	9	7	8	6	4	3
0	1	2	3	4	5	6	7	8	9	10

y el **pivote** aleatorio 

6
---

■ Elementos menores: 

3
---

5
---

2
---

1
---

4
---

3
---

■ Elementos mayores: 

7
---

9
---

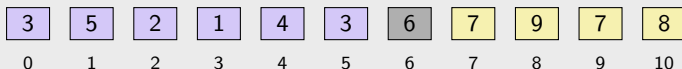
7
---

8
---

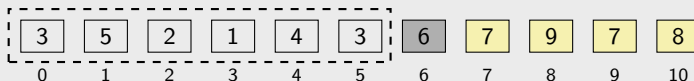
Con esta información, ¿es 6 la mediana? ¿dónde se ubica esta?

# Primera estrategia... recursiva

Moviendo los elementos según si son mayores o menores que el pivote,



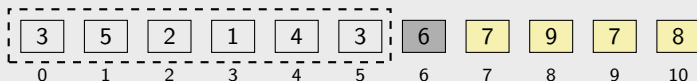
Como la posición de la mediana debiera ser  $\lfloor 11/2 \rfloor = 5$  y el pivote quedó en la posición 6, revisamos el tramo a la izquierda del pivote



# Primera estrategia... recursiva

Tomando aleatoriamente el pivote

2



■ Elementos menores:

1

■ Elementos mayores:

3

5

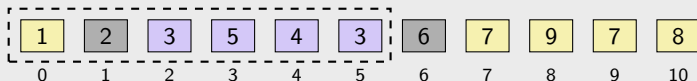
4

3

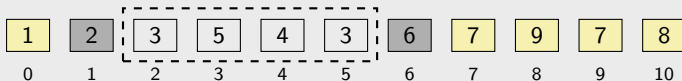
Reordenamos el tramo analizado para ver si el pivote es la mediana

# Primera estrategia... recursiva

Ubicamos los elementos según el pivote 2, obteniendo

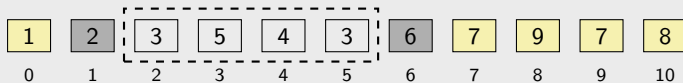


Nuevamente, como la posición del pivote es 1, nos concentramos recursivamente en el sub-tramo derecho del último pivote



# Primera estrategia... recursiva

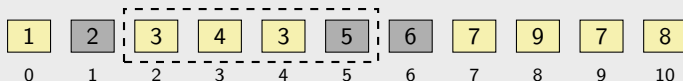
Con el pivote **5**



■ Elementos menores: **3** **4** **3**

■ Elementos mayores:  $\emptyset$

Reubicamos los elementos para ver la posición del pivote actual



Como el pivote está en la posición  $\lfloor n/2 \rfloor = \lfloor 11/2 \rfloor = 5$ , **es la mediana**

# Particiones

Si suponemos que no hay elementos iguales al pivote, elegir un pivote  $p$  **particiona** la secuencia en

- Elementos mayores al pivote,  $M = \{a_i \mid a_i > p\}$
- Elementos menores al pivote,  $m = \{a_i \mid a_i < p\}$

Luego, si reubicamos los elementos de la secuencia de forma que

- Primero se colocan los elementos de  $m$
- Luego se coloca  $p$
- Finalmente se colocan los elementos de  $M$

es claro que  $p$  está en su posición ordenada

# Partition

Supongamos que no hay elementos repetidos en la secuencia

**input** : Secuencia  $A[0, \dots, n-1]$ , índices  $i, f$

**output**: Índice del pivote aleatorio en la secuencia ordenada

**Partition** ( $A, i, f$ ):

```
1   $p \leftarrow$  pivote aleatorio en  $A[i, f]$ 
2   $m, M \leftarrow$  secuencias vacías
3  Insertar  $p$  en  $M$ 
4  for  $x \in A[i, f]$  :
5      if  $x < p$  : Insertar  $x$  en  $m$ 
6      elif  $x > p$  : Insertar  $x$  en  $M$ 
7   $A[i, f] \leftarrow \text{Concat}(m, M)$ 
8  return  $i + |m|$ 
```

Notemos que **Partition** retorna y además reordena los elementos de  $A$

# Partition

Supongamos que no hay elementos repetidos en la secuencia

Partition ( $A, i, f$ ):

```
1   $p \leftarrow$  pivote aleatorio en  $A[i, f]$ 
2   $m, M \leftarrow$  secuencias vacías
3  Insertar  $p$  en  $M$ 
4  for  $x \in A[i, f]$  :
5      if  $x < p$  : Insertar  $x$  en  $m$ 
6      elif  $x > p$  : Insertar  $x$  en  $M$ 
7   $A[i, f] \leftarrow$  Concat( $m, M$ )
8  return  $i + |m|$ 
```

- Se ubican los menores, el pivote y luego los mayores
- $i$ : # elems. a la izq. de  $A[i]$
- $|m|$ : # menores a  $p$  en  $A[i, f]$
- Por lo tanto, se retorna la posición correcta de  $p$

Su retorno es la cantidad de elementos a la izquierda de  $p$  en la secuencia ordenada



# Complejidad de Partition

Partition ( $A, i, f$ ):

```
1   $p \leftarrow$  pivote aleatorio en  $A[i, f]$ 
2   $m, M \leftarrow$  secuencias vacías
3  Insertar  $p$  en  $M$ 
4  for  $x \in A[i, f]$  :
5      if  $x < p$  : Insertar  $x$  en  $m$ 
6      elif  $x > p$  : Insertar  $x$  en  $M$ 
7   $A[i, f] \leftarrow \text{Concat}(m, M)$ 
8  return  $i + |m|$ 
```

Las inserciones al final de la EDD

- $\mathcal{O}(1)$  en arreglos y listas
- Se hace una por elemento  $x$
- Total:  $\mathcal{O}(n)$

La concatenación

- $\mathcal{O}(1)$  en listas
- $\mathcal{O}(n)$  en arreglos

Además, usa memoria adicional  $\mathcal{O}(n)$  para mantener las secuencias  $m, M$

Más adelante veremos una versión *in place* de Partition

# Median

Supongamos que no hay elementos repetidos en la secuencia

**input** : Secuencia  $A[0, \dots, n-1]$

**output**: Elemento en posición central de  $A$

Median ( $A$ ):

```
1    $i \leftarrow 0$ 
2    $f \leftarrow n - 1$ 
3    $x \leftarrow \text{Partition}(A, i, f)$ 
4   while  $x$  no es el centro :
5       if  $x < \text{centro}$  :  $i \leftarrow x + 1$ 
6       else:  $f \leftarrow x - 1$ 
7        $x \leftarrow \text{Partition}(A, i, f)$ 
8   return  $A[x]$ 
```

# Median

Supongamos que no hay elementos repetidos en la secuencia

Median ( $A$ ):

```
1   $i \leftarrow 0$ 
2   $f \leftarrow n - 1$ 
3   $x \leftarrow \text{Partition}(A, i, f)$ 
4  while  $x$  no es el centro :
5      if  $x < \text{centro}$  :  $i \leftarrow x + 1$ 
6      else:  $f \leftarrow x - 1$ 
7       $x \leftarrow \text{Partition}(A, i, f)$ 
8  return  $A[x]$ 
```

- $x$ : # datos menores al pivote
- Median entrega el elemento en la posición central ordenada
- Para  $n$  impar, Median( $A$ ) corresponde a la mediana
- Para  $n$  par, Median( $A$ ) es uno de los dos elementos centrales

Podemos parametrizar Median para encontrar el  $k$ -ésimo estadístico de orden

# Median (versión parametrizada)

Supongamos que no hay elementos repetidos en la secuencia

**input** : Secuencia  $A[0, \dots, n-1]$ , índice  $k \in \{0, \dots, n-1\}$

**output**:  $k$ -ésimo estadístico de orden de  $A$

Median ( $A, k$ ):

```
1   $i \leftarrow 0$ 
2   $f \leftarrow n - 1$ 
3   $x \leftarrow \text{Partition}(A, i, f)$ 
4  while  $x \neq k$  :
5      if  $x < k$  :  $i \leftarrow x + 1$ 
6      else:  $f \leftarrow x - 1$ 
7       $x \leftarrow \text{Partition}(A, i, f)$ 
8  return  $A[x]$ 
```

- Median( $A, 0$ ) entrega el menor elemento de  $A$
- Median( $A, n - 1$ ) entrega el mayor elemento de  $A$

# Median y dividir para conquistar

El algoritmo Median también aplica la estrategia dividir para conquistar

1. División del problema en dos sub-problemas (Partition)
2. Naturaleza del problema hace que solo sea necesario resolver uno de los dos sub-problemas
  - Similar al caso de búsqueda binaria
  - Resolvemos el sub-problema correspondiente a la zona donde debiera estar la mediana
  - Se puede cambiar el enfoque recursivo por uno iterativo
3. Solución al problema original se obtiene de la solución al sub-problema

# Complejidad de Median

## Discusión

¿Cuál es la complejidad de Median?

- ¿Importa el orden de  $A$ ?
- ¿Depende de algo distinto?
- ¿Cuáles serían su mejor y peor caso?

# Complejidad de Median

La complejidad de Median depende la elección del pivote: es **probabilística**

El **mejor caso**

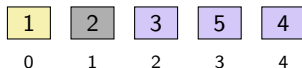
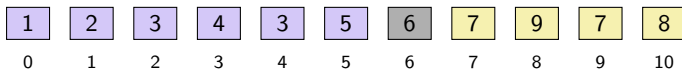
- Escoger la mediana como primer pivote
- Esto significa separar los demás elementos en  $m$  y  $M$
- Eso toma  $\mathcal{O}(n)$  por pivote  $\mathcal{O}(n)$
- Como solo se debe tomar un pivote: Total:  $\mathcal{O}(n)$

El **peor caso**

- Que todos los datos menos la mediana sean escogidos como pivote
- Para cada pivote hay que separar menores y mayores  $\mathcal{O}(n)$
- Como solo se deben tomar  $\mathcal{O}(n)$  pivotes: Total:  $\mathcal{O}(n^2)$

# Ordenar con Partition

Luego de cada ejecución de Partition, el pivote queda en su posición ordenada



Además, las dos sub-secuencias están “*del lado correcto*” del pivote

¿Cómo usar esto para ordenar?



# Sumario

Obertura

Particiones

**Quicksort**

Epílogo

# Ordenar con Partition

Nuevamente podemos usar la estrategia dividir para conquistar

1. División del problema en dos sub-problemas con Partition
2. Aplicar recursivamente Partition para cada sub-secuencia
3. No necesitamos combinar nada: la secuencia queda ordenada

Llamaremos **Quicksort** a este algoritmo

# Quicksort

**input** : Secuencia  $A[0, \dots, n-1]$ , índices  $i, f$

**output**:  $\emptyset$

QuickSort ( $A, i, f$ ):

```
1  if  $i \leq f$  :  
2       $p \leftarrow \text{Partition}(A, i, f)$   
3      Quicksort( $A, i, p-1$ )  
4      Quicksort( $A, p+1, f$ )
```

El llamado inicial es Quicksort( $A, 0, n-1$ )

# Quicksort: Ejemplo de ejecución

A S O R T I N G E X A M P L E  
A A E E T I N G O X S M P L R  
A A E  
A A  
A

L I N G O P M R X T S  
L I G M O P N  
G I L  
— I L  
— I

N P O  
— O P  
— — P

S T X  
— T X  
— T

A A E E G I L M N O P R S T X

En este ejemplo, el pivote es siempre el elemento que está en el extremo derecho del subarreglo, es decir,  $A[f]$ ; al terminar *Partition*, el pivote queda en la posición que se muestra en rojo

# Complejidad de memoria de Quicksort

En términos de memoria

- La complejidad de Quicksort depende solo de Partition
- Vimos una versión  $\mathcal{O}(n)$  de Partition
- Es posible contar con una versión *in place*

Para la versión *in place* usaremos arreglos

- Haremos intercambios dentro del arreglo
- El truco será partir poniendo el pivote al final

## Versión *in place* de Partition

**input** : Arreglo  $A[0, \dots, n-1]$ , índices  $i, f$

**output**: Índice del pivote aleatorio en la secuencia ordenada

**Partition** ( $A, i, f$ ):

```
1   $x \leftarrow$  índice aleatorio en  $\{i, \dots, f\}$ 
2   $p \leftarrow A[x]$ 
3   $A[x] \rightleftharpoons A[f]$ 
4   $j \leftarrow i$ 
5  for  $k = i \dots f - 1$  :
6      if  $A[k] < p$  :
7           $A[j] \rightleftharpoons A[k]$ 
8           $j \leftarrow j + 1$ 
9   $A[j] \rightleftharpoons A[f]$ 
10 return  $j$ 
```

Con este cambio, Quicksort usa memoria adicional  $\mathcal{O}(1)$

# Versión *in place* de Partition

Partition ( $A, i, f$ ):

```
1   $x \leftarrow$  índice aleatorio en  $\{i, \dots, f\}$ 
2   $p \leftarrow A[x]$ 
3   $A[x] \rightleftharpoons A[f]$ 
4   $j \leftarrow i$ 
5  for  $k = i \dots f - 1$  :
6      if  $A[k] < p$  :
7           $A[j] \rightleftharpoons A[k]$ 
8           $j \leftarrow j + 1$ 
9   $A[j] \rightleftharpoons A[f]$ 
10 return  $j$ 
```



# Análisis de Quicksort

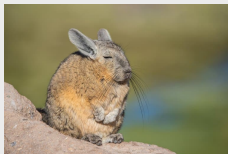
Próxima clase demostraremos

- correctitud
- complejidad de tiempo
- En particular, que en el caso promedio es  $\mathcal{O}(n \log(n))$

Partiremos de la base que `Partition` es correcto

## Ejercicio (propuesto)

Demuestre que `Partition` es correcto





# Sumario

Obertura

Particiones

Quicksort

**Epílogo**

# Objetivos de la clase

- ☐ Comprender la estrategia de elegir pivote para particionar una secuencia
- ☐ Identificar cuándo tal estrategia encuentra la mediana
- ☐ Comprender y analizar el algoritmo Partition
- ☐ Comprender el uso de Partition en Median para encontrar el elemento central
- ☐ Comprender el uso de Partition para ordenar secuencias