Ayudantía N° 4

# QuickSort

# Repaso

Pseudocódigo

# QuickSort

**Input:** secuencia A y dos enteros i, f

**Output:** Nada

```
QuickSort (A, i, f):
1    if i ≤ f :
2        p ← Partition(A, i, f)
3        Quicksort(A, i, p − 1)
4        Quicksort(A, p + 1, f)
```

# El verdadero héroe:

- Elige el pivote
- Coloca al pivote en su posición correcta en el arreglo ordenado
- Pone los elementos menores que él a su izquierda y los mayores a su derecha

```
Partition (A, i, f):
1     x ← índice aleatorio en {i, ..., f}
2     p ← A[x]
3     A[x] ⇄ A[f]
4     j ← i
5     for k = i ... f − 1 :
6         if A[k] < p :
7             A[j] ⇄ A[k]
8             j ← j + 1
9     A[j] ⇄ A[f]
10    return j
```

# Resumen QuickSort

- Elegir pivote
- Mover elementos a cada lado del pivote, a un lado los mayores y al otro los menores
- La lista va a quedar separada en dos sublistas, una con los elementos a la izq. del pivote y otra con los de la derecha
- Repetir recursivamente para cada sublista mientras estas contengan más de un elemento.
- Lista ordenada! 

# Complejidad

| | |
|---|---|
| Mejor Caso | O(n * log(n)) |
| Caso Promedio | O(n * log(n)) |
| Peor Caso | O(n^2) |

¿Cuándo ocurre cada uno de ellos?

# Veamos la visualización !

https://visualgo.net/en/sorting

# Extra : Ejercitemos

a) Supongamos que al ejecutar Quicksort sobre un arreglo particular, la subrutina partition hace siempre el mayor número posible de intercambios; ¿cuánto tiempo toma Quicksort en este caso? ¿Qué fracción del mayor número posible de intercambios se harían en el mejor caso? Justifica

**Pregunta 1 a) - I1-2020-2**

# Solución 1 a) - I1-2020-2

- Intercambios de líneas 3 y 9 se hacen siempre
- Dentro del loop, sólo se hacen cuando A[k] < p
- Si queremos hacer la mayor cantidad de intercambios posibles, queremos que siempre A[k]< p.
- Entonces si tenemos un subarreglo de tamaño m, se hacen 2 + m-1 = m+1 intercambios
- Quedan subarreglos de largos 0 y m-1

```
Partition (A, i, f):
1     x ← índice aleatorio en {i, ..., f}
2     p ← A[x]
3     A[x] ⇄ A[f]
4     j ← i
5     for k = i ... f − 1 :
6         if A[k] < p :
7             A[j] ⇄ A[k]
8             j ← j + 1
9     A[j] ⇄ A[f]
10    return j
```

## Solución 1 a) - I1-2020-2

Con un arreglo de tamaño n tenemos:

$$T(n) = n + (n-1) + (n-2) + \ldots + 1$$

$$= \frac{n \times (n+1)}{2} = \frac{1}{2}\left(n^2 + n\right)$$

$$\in O\left(n^2\right)$$

b) El algoritmo quicker-sort llama a la subrutina pq-partition, que utiliza dos pivotes $\boldsymbol{p}$ y $\boldsymbol{q}$ $(\boldsymbol{p} < \boldsymbol{q})$ para particionar el arreglo en 5 partes: los elementos menores que $\boldsymbol{p}$, el pivote $\boldsymbol{p}$, los elementos entre $\boldsymbol{p}$ y $\boldsymbol{q}$, el pivote $\boldsymbol{q}$, y los elementos mayores que $\boldsymbol{q}$. Escribe el pseudocódigo de pq-partition. ¿Es quicker-sort más eficiente que quick-sort? Justifica.

**Pregunta 1 b) - I1-2020-2**

# Solución P1 b) - I1-2020-2

Ya que la definición dice que $p < q$, entonces es posible asumir que no hay datos repetidos. De todos modos, esto no afecta el análisis.

Lo más simple es implementar partition con listas ligadas como vimos en clases:

```
1:  procedure PQ-PARTITION(lista ligada L)
2:      p, q ← dos nodos de L tal que p < q. Quitar estos nodos de L.
3:      A, B, C ← listas vacias    ▷ Los elementos menores a p, entre p y q y mayores a q respectivamente
4:      for nodo x ∈ L do
5:          if x < p then
6:              agregar x al final de A
7:          else if x < q then
8:              agregar x al final de B
9:          else
10:             agregar x al final de C
11:         end if
12:     end for
13:     return A, p, B, q, C
14: end procedure
```

El paso en la linea 2 es fácil de hacer en $\mathcal{O}(1)$, basta con extraer los primeros dos elementos de $L$, e intercambiarlos si $p > q$.

# Solución P1 b) - I1-2020-2

Recordemos que el peor caso de Quicksort se produce cuando partition separa una secuencia de $m$ datos en dos secuencias disparejas, de $0$ y $m-1$ datos cada una. En ese caso la complejidad para una secuencia inicial de $n$ datos es de:

$$T(n) = n + (n-1) + (n-2) + \cdots + 1$$

$$T(n) \in \mathcal{O}(n^2)$$

En este caso esto también puede suceder, solo que se separa en 3 secuencias disparejas, de $0$, $0$ y $m-2$ elementos cada una. En este caso la complejidad para una secuencia inicial de $n$ datos es de:

$$T(n) = n + (n-2) + (n-4) + \cdots + 1$$

$$T(n) \in \mathcal{O}(n^2)$$

Ambos algoritmos son iguales en el peor caso, así que no podemos afirmar que quickersort sea mejor que quicksort.

**1)** Escribe el algoritmo *quicksort3*, que, en lugar de particionar el arreglo $A$ en dos, como lo hace *quicksort*, lo particiona en tres: datos menores que el pivote, datos iguales al pivote, y datos mayores que el pivote. Puedes suponer que las particiones van a parar a listas diferentes o bien al mismo arreglo —especifica. Usa una notación similar a la usada en las diapositivas.

## Pregunta 1 - C1-2019-1

# Solución 1 - C1-2019-1 - Partition3

**Algorithm 1** Partition3(A, i, f)

```
p ← elemento aleatorio en A[i, f]
m, M, P ← secuencias vacías
for x in A[i, f] do
    if x < p then
        Insertar x en m
    else if x = p then
        Insertar x en P
    else if x > p then
        Insertar x en M
    end if
end for
A[i, f] ← Concat(m, p, P, M)
return i + |m|, i + |m| + |P|
```

# Solución 1 - C1-2019-1 (versión in-place)

**Algorithm 2** Partition3(A, i, f)

```
1:  x ← índice aleatorio en {i, ..., f}
2:  p ← A[x]
3:  swap(A[x], A[f])
4:  j ← i
5:  l ← i
6:  for k in {i, ..., f − 1} do
7:      if A[k] < p then
8:          swap(A[k], A[j])
9:          swap(A[j], A[l])
10:         j ← j + 1
11:         l ← l + 1
12:     end if
13:     if A[k] = p then
14:         swap(A[k], A[j])
15:         j ← j + 1
16:     end if
17: end for
18: swap(A[j], A[f])
19: return l, j
```

# Solución 1 - C1-2019-1 - Llamada a Quicksort

**Algorithm 3** Quicksort3(A, i, f)

```
1: if i ≤ f then
2:     p1, p2 = Partition3(A, i, f)
3:     Quicksort3(A, i, p1 − 1)
4:     Quicksort3(A, p2 + 1, f)
5: end if
```

2. En ocasiones, como vimos con ***QuickSort***, los algoritmos pueden tener una componente aleatoria. Esto puede ser muy útil, ya que en el caso de ***QuickSort*** significa que no podemos forzar el peor caso intencionalmente. En otros algoritmos, por ejemplo, puede ser un problema. Considera el siguiente algoritmo de ordenación que recibe un conjunto $A$ de $n$ elementos:

```
BogoSort(A,n):
    while(true):
        Sea B un arreglo vacío
        while A aún tenga elementos:
            Extraer aleatoriamente un elemento de A y ponerlo al final de B
        if B está ordenado:
            return B
        Devolver todos los elementos de B a A
```

a. Demuestra que ***BogoSort*** no es correcto según la definición vista en clases. ¿Cuál es la lógica detrás de este algoritmo? ¿Qué habría que modificar en el algoritmo para que este sea correcto, manteniendo esta lógica?

b. ¿Qué hace que ***QuickSort*** sea correcto, pese a que toma decisiones aleatorias?

# Pregunta 2: Propuesto - I1-2020-1

# Solución P2 a) - I1-2020-1

**Finitud:** Como se extraen elementos de forma aleatoria de A a B, aun que un orden esté mal, esto no asegura que en un intento futuro no se valla a repetir el mismo orden. Por lo tanto no hay manera de demostrar que hay un avance en dirección al final. Por lo tanto el algoritmo falla en este ámbito.**(Max 0.4 puntos)**

**Lógica:** La lógica detrás de este algoritmo es generar permutaciones aleatorias de los datos, con la esperanza de obtener un arreglo ordenado. **Max 0.3 puntos**

**Arreglo:** La manera de mantener la aleatoriedad pero arreglar el problema de la finitud sería asegurar de alguna forma que una misma permutación, de los elementos del conjunto A, no se genere genere dos veces. En otras palabras, que no se repita la misma permutación más de una vez.**Max 0.3 puntos**

**Criterios:** Para que un algoritmo de ordenación sea correcto necesitamos ver que se cumplan dos cosas:

- Finitud: Termina en una cantidad finita de pasos.
- Correctitud: Cumple con su función al terminar. Como es un algoritmo de ordenación: Que al retornar B, este esté ordenado. **(0.1 puntos)**

**Correctitud:** Como sabemos que para retornar B se tiene que cumplir que "B está ordenado", entonces en esta parte el algoritmo revisa si B está ordenado. Si está ordenado, retorna B ordenado, por lo que cumple con su función. Si no está ordenado, no retorna, devuelve los elementos de B a A y vuelve a correr el while. Por lo tanto sabemos que si B es retornado, estará ordenado.**(0.1 puntos)**

# Solución P2 b) - I1-2020-1

b)

La aleatoriedad de Quicksort se encuentra en la elección del pivote para realizar la partición. Una vez que un elemento es elegido como pivote, éste queda en su posición correcta y no puede volver a ser elegido como pivote. Lo anterior implica que en cada iteración el número de elementos a ordenar va disminuyendo, por lo tanto, con un número de iteraciones finitas, el algoritmo termina para cualquier caso, lo que no sucede con Bogosort. **(1**