

# Análisis de ~~Complejidad~~ Simplicidad de Algoritmos



Celia Rubio Madrigal

### Definición 1.1. (Algoritmo reticente)

$A$  es un algoritmo reticente para el problema  $P$  sí, y solo sí, [1]

$$(\forall M) \mathcal{G}(A, t, M, P) \implies \bigwedge_{o \in \mathcal{I}} \mathcal{E}(M, o)$$

donde  $t$  es el tiempo,  $\mathcal{G}(A, t, M, P)$  es el predicado

*“ $A$  gasta un tiempo  $t$  de la manera  $M$  mientras resuelve  $P$ ”,*

$\mathcal{I}$  es el conjunto de observadores ingenuos y  $\mathcal{E}(M, o)$  es el predicado

*“ $M$  es suficientemente artificiosa como para engañar a  $o$ ”.*

**Ejemplo 1.1.** La búsqueda de la llave correcta a la hora de abrir una cerradura.

**Ejemplo 1.2.** El recorrido de grafos en la Odisea de Homero.

**Ejemplo 1.3.** La vuelta atrás.

**Ejemplo 1.4.** La resolución de buscaminas de María.





Merge



Bubble



Quick





# Algoritmo 1.1. (Bogo)

```
import random

def is_sorted(L) :
    return all(L[i] <= L[i+1] for i in range(len(L)-1))

def bogo(L) :
    while not is_sorted(L) :
        random.shuffle(L)
    return L
```

# Algoritmo 1.1. (Bogo)

(Versión probabilista *Las Vegas*)

```
def LV(x, y) :  
    y = x  
    random.shuffle(y)  
    return is_sorted(y)
```

```
def repetirLV(x) :  
    y = x  
    exito = False  
    while not exito :  
        LV(x, y, exito)  
    return y
```

Si  $|x| = n$ :

$$\blacktriangleright p(x) = \frac{1}{n!}$$

$$\blacktriangleright e(x) = f(x) = n$$

$$\begin{aligned}\blacktriangleright t(x) &= e(x) + \frac{1 - p(x)}{p(x)} f(x) \\ &= n + (n! - 1)n = n \cdot n!\end{aligned}$$

# Algoritmo 1.1. (Bogo)



Tiempo esperado:  $O(n \cdot n!)$



Caso mejor: ¡  $O(n)$  !



Caso peor: ¡¡  $O(\infty)$  !!



¿Generador aleatorio?



## Arreglo 1.1. ( y )

En vez de permutar aleatoriamente,  
recorremos las permutaciones una a una.

```
from itertools import  
    permutations  
  
def bad(L) :  
    P = permutations(L)  
    for X in P :  
        if is_sorted(X) :  
            return X  
    return
```

## Arreglo 1.2. ( SOON )

En vez de recorrer las permutaciones,  
generamos todas ellas al principio.

```
def worse(L) :  
    P = list(permutations(L))  
    for X in P :  
        if is_sorted(X) :  
            return X  
    return
```

**Arreglo 1.3. (🕒)**

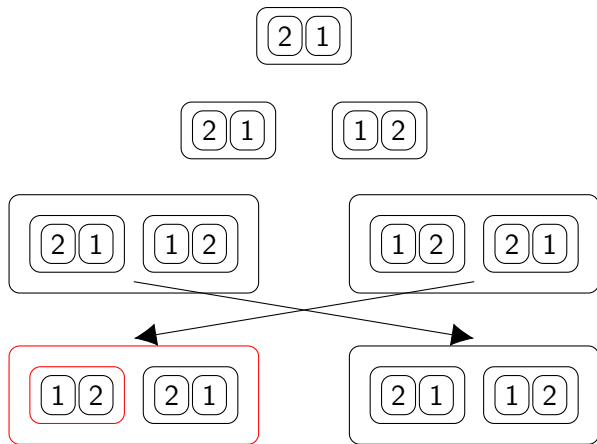
## Arreglo 1.3. ()

La permutación correcta es la lexicográficamente **menor** de ellas  $\implies$  podemos **ordenarlas** y tomar la primera permutación. [2]

## Arreglo 1.3. ()

La permutación correcta es la lexicográficamente **menor** de ellas  $\implies$  podemos **ordenarlas** y tomar la primera permutación. [2]

## ¡Ordenación recursiva!



## Algoritmo 1.2. (Worst Sort)

```
def multi(L, N) :  
    if N == 0 :  
        return sorted(L)  
    P = list(permutations(L))  
    P = multi(P, N-1)  
    return P[0]  
  
def worst(L) :  
    return multi(L, cota(len(L)))
```

$$\blacktriangleright \Omega(n!! \cdots^{f(n)} !!)$$

## Algoritmo 1.2. (Worst Sort)

```
def multi(L, N) :  
    if N == 0 :  
        return sorted(L)  
    P = list(permutations(L))  
    P = multi(P, N-1)  
    return P[0]  
  
def worst(L) :  
    return multi(L, cota(len(L)))
```

$$\blacktriangleright \Omega(n!! \cdots^{f(n)} !!)$$

$$\blacktriangleright \lim_{n \rightarrow \infty} \frac{\boxed{B} \boxed{B}}{(n!! \cdots^{f(n)} !!)} = \infty$$



Input interpretation:

$((3!))! \text{ps}$  (picoseconds)

Results:

$2.601 \times 10^{1746} \text{ps}$  (picoseconds)

$2.601 \times 10^{1746} \text{ps}$  (picoseconds)

Unit conversions:

[More digits](#)

[Exact forms](#)

$2.601218943565795 \times 10^{1734}$  seconds

$8.242934690162076 \times 10^{1726}$  average Gregorian years

Comparisons as age:

$\approx 5.7 \times 10^{1716}$  × Hubble time ( $\approx 4.6 \times 10^{17} \text{ s}$ )

$\approx 6 \times 10^{1716}$  × age of the universe ( $\approx 14 \text{ Gyr}$ )

Interpretations:

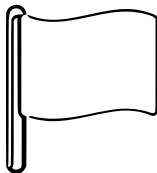
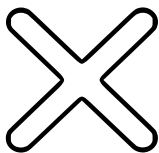
[More](#)

**time**

**age**

 [Download Page](#)

POWERED BY THE WOLFRAM LANGUAGE



(multiplica y ríndete)

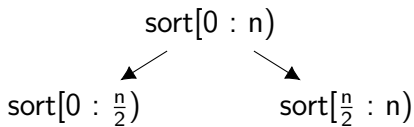


(multiplica y ríndete)

sort[0 : n)

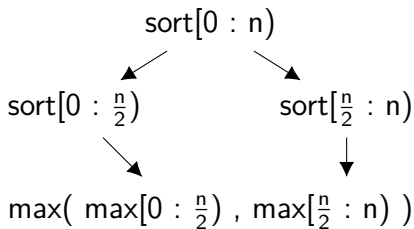


(multiplica y ríndete)



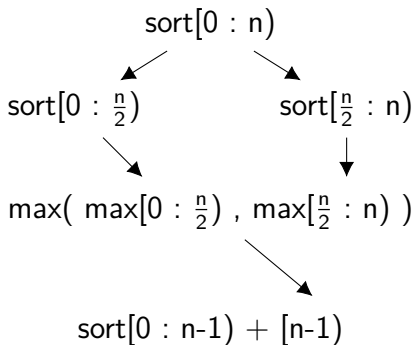


(multiplica y ríndete)



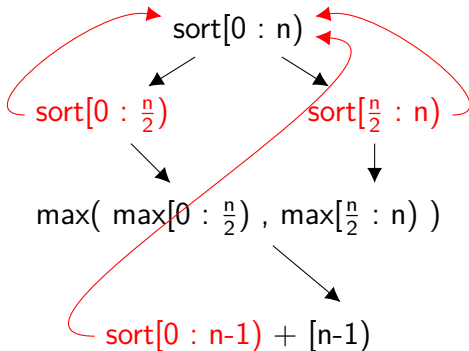


(multiplica y ríndete)





(multiplica y ríndete)



## Algoritmo 1.3. (Slow Sort)

```
def slow(L, i, j) :  
    if i >= j :  
        return  
    m = (i+j) // 2  
    slow(L, i, m)  
    slow(L, m + 1, j)  
    if L[j] < L[m] :  
        L[j], L[m] = L[m], L[j]  
    slow(L, i, j-1)  
    return L
```





## Referencias



- [1] Broder, A., Stolfi, J.  
*Pessimal Algorithms and Simplicity Analysis*, 1984.
- [2] Lerma, M. A.  
*How inefficient can a sort algorithm be?* 2014.



Código de las diapositivas (C++, Python)