¿Qué quiere decir que un algoritmo sea correcto?

1. Termina en una cantidad finita de pasos (buscar donde termina)
2. Cumple su propósito → hace lo que tiene que hacer (Usar inducción)

Teorema Maestro

Si $a_1, a_2, b, c, c_0, d \in \mathbb{R}^+$ y $b \geq 1$, entonces

$$T(n) = \begin{cases} c_0 & 0 \leq n < \frac{b}{b-1} \\ a_1 T(\lceil \frac{n}{b} \rceil) + a_2 T(\lceil \frac{n}{b} \rceil) + cn^d & n \geq \frac{b}{b-1} \end{cases}$$

Se cumple

$$T(n) \in \begin{cases} O(n^d) & a_1 + a_2 < b^d \\ O(n^d \log n) & a_1 + a_2 = b^d \\ O(n^{\log_b(a_1+a_2)}) & a_1 + a_2 > b^d \end{cases}$$

Selection Sort

Siempre buscamos el min

```
SelectionSort(array, size):
  repeat (size-1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
     if element < currentMinimum
        set element as new minimum
  Swap minimum with first unsorted position
end SelectionSort
```

Siempre tiene complejidad $O(n^2)$, independiente de la estructura.
Se tienen siempre dos arrays: el sub-arreglo que ya esta ordenado y otro que no esta

Insertion Sort

```
InsertionSort(array):
  mark first element as sorted
  for each unsorted element X
    "extract" the element X
    for j ← lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end InsertionSort
```

Caso limite → peor caso (orden inverso).
En listas ligadas doblemente tiene complejidad $O(n)$

La complejidad depende de que tan ordenados esten los datos.
Mejor caso. $O(n)$ para datos ya ordenados
Promedio = peor caso. $O(n^2)$ orden inverso
Siempre comparamos con los elementos de la izquierda

Complejidad por sumatoria

$\sum_{i=0}^{3-1} a^i = \frac{1-a^3}{1-a}$

$\sum_{i=0/1}^{3} i = \frac{m(m+1)}{2}$

$\sum_{i=1}^{3} \log i = \log n! \quad \sum_{i=1}^{3} \frac{1}{i} \in O(\log m)$

$\sum_{i=1}^{3} \log(i) \in O(m \log m)$

Merge Sort

```
MergeSort(arr, l, r):  → arr es acotado (l, r)
                       → Divide and Conquer
if r > l:
  1. Find the middle point to divide the array into two halves
        middle m = l + (r-l)/2
  2. Call mergesort for the first half
        call mergeSort(arr, l, m)
  3. Call mergesort for the second half
        call mergeSort(arr, m+1, r)
  4. Merge the two halves sorted:
        call merge(arr, l, m, r)
```

Partimos a la mitad y luego unimos

Si tiene 1 elemento $T(1) = 1$

Para el caso general: $T(n) = 2T(\lceil \frac{n}{2} \rceil) + n$

Se repite $\log_2 n$ veces

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

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

$\vdots$

$T(2)/2 = T(1)/1 + 1$

$T(n)/n = T(1)/1 + \log n$

$T(n) = n \log n + n = O(n \log n)$

Podemos acotar $n \leq k^r \Leftrightarrow k \geq \sqrt[r]{n}$

Merge Sort P2

En terminos de complejidad de tiempo, siempre es O(nlogn) → mergesort siempre divide el array en dos mitades y toma tiempo lineal para fusionar dos mitades.

Divide and Conquer

1. Divide. se divide el problema en sub-problemas más pequeños
2. Conquer. resolvemos los sub-problemas al llamarlo de forma recursiva hasta que sea resuelto
3. Combine. combinamos los subproblemas hasta llegar a la solución del problema.

Binary Search

Binary search solo funciona en un array ordenado

Mejor caso: O(1)

Caso promedio = peor = O(logn)

```
Binary Search(arr, x, low, high):
    if low > high:
        return False
    else:
        mid = (low + high)/2
        if x = arr[mid]
            return mid
        else if x > arr[mid]:        # x en right
            return binarysearch(arr, x, mid+1, high)
        else:                        # x on left
            return binarysearch(arr, x, low, mid-1)
```

Quick Sort

Este algoritmo se basa en dos partes, la aleatoriedad de QuichSort se basa en la elección del pivote para realizar una partición.

```
Partition (array, leftMostIndex, rightmostIndex):
    Set rightmostIndex as pivotIndex
    storeIndex = leftmostIndex - 1
    for i ← leftmostIndex + 1 to rightmostIndex
        if element[i] < pivotelement
            swap element[i] and element[storeIndex]
            storeIndex ++
    swap pivotElement and element[storeIndex + 1]
    return storeIndex + 1
```

De este depende la complejidad

↳ podemos elejir → primer elemento
→ ultimo elemento
→ mediana
→ aleatorio

↳ ponemos el pivote en su posición correcta

```
QuichSort (arr, low, high):
    if (low < high)
        pi = partition (arr, low, high)
        quichsort(arr, low, pi-1) - antes de pi
        quichSort(arr, pi+1, high) - despues de pi
```

Encuentra el elemento llamado pivote que divide al array en dos mitades de forma que los elementos a la izquierda son más pequeños y los de la derecha más grandes.

En terminos generales: $T(n) = T(k) + T(n-k-1) + O(n)$, donde hay k elementos menores al pivote

Peor caso: $T(n) = T(n-1) + O(n) = O(n^2)$ cuando tomamos el elemento mas grande o más pequeño

Mejor Caso: $T(n) = 2T(n/2) + O(n) = O(n \log n)$ cuando elejimos de pivote al elemento del medio

Caso promedio: $O(n \log n)$

Qvichsort es más rapido que mergesort, por la estructura del loop interno