Ordenamiento

Algoritmos y Estructuras de Datos
2024

UNRaf UNIVERSIDAD NACIONAL DE RAFAELA

Dada la siguiente lista de números, ¿cómo diseñaría un algoritmo para ordenarlos?

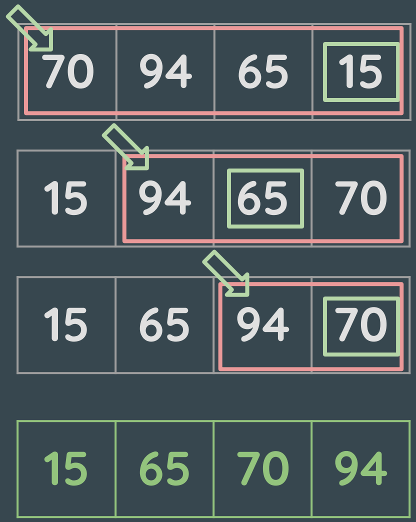
70	94	65	15	85	51	24	22	31	45

Algoritmos de Ordenamiento

- Existen muchas estrategias de ordenamiento:
  - por selección (selection sort)
  - por burbuja (bubble sort)
  - por inserción (insertion sort)
  - por mezcla (merge sort)
  - ordenamiento rápido (quick sort)
  - ....
- En general, los algoritmos presentan una situación de compromiso:
  - los algoritmos **más eficientes** son **más complejos**.
  - los algoritmos **sencillos** **no suelen ser eficientes**.

Selection sort

- Se basa en buscar el menor (o mayor) elemento en la lista y ubicarlo en la primer (última) posición.
- Este procedimiento se repite para todas las sublistas de tamaño n-1




```
para i entre 0, len(A)-1:
  # Encontrar el menor y su posicion
  menor = +inf
  pos = 0
  para j entre i, len(A)-1:
    si A[j] < menor:
      menor = A[j]
      pos = j
  # Intercambiar el menor por el 1er elemento
  aux = A[i]
  A[i] = menor
  A[pos] = aux
```

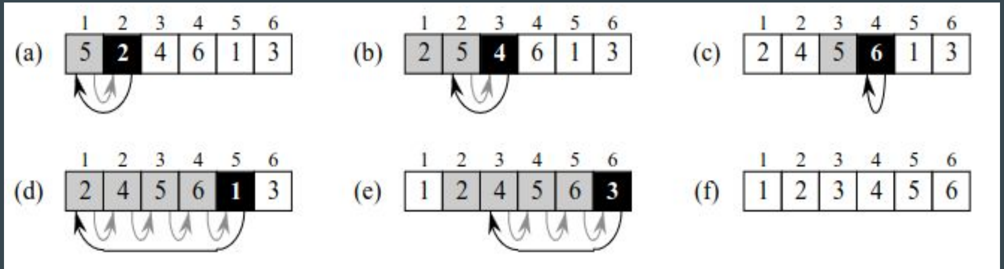
Bubble sort

- Consiste en comparar un elemento con el siguiente: si no están en orden, se invierten las posiciones.


```
para i entre 0, len(A)-1:
  para j entre 0, len(A)-1-i:
    si A[j] > A[j+1]:
      aux = A[j]
      A[j] = A[j+1]
      A[j+1] = aux
```

Insertion sort

- Es un algoritmo sencillo y eficiente para pocos datos.
- Se compara un número con todos los anteriores, hasta ubicar la posición correcta donde **insertarlo**.




```
para i entre 1, len(A):
  insertar = A[i]
  # Encontrar ubicación donde insertar
  ubic = 0
  mientras ubic < i y A[ubic] > insertar:
    ubic += 1
  # Mover todos los elementos
  # yendo de derecha a izquierda
  para j entre i-1 a ubic:
    A[j+1] = A[j]
  # Insertar elemento en su ubicación
  A[ubic] = insertar
```

Complejidad algoritmos

- Al tener bucles anidados que recorren (casi) todo el *array*, podemos establecer que el tiempo requerido por los 3 algoritmos crece de manera **cuadrática**.
- Como los reordenamientos se hacen sobre el mismo arreglo, no se requiere espacio adicional (salvo variables auxiliares).
- Por ello, la complejidad del espacio usado es **constante**

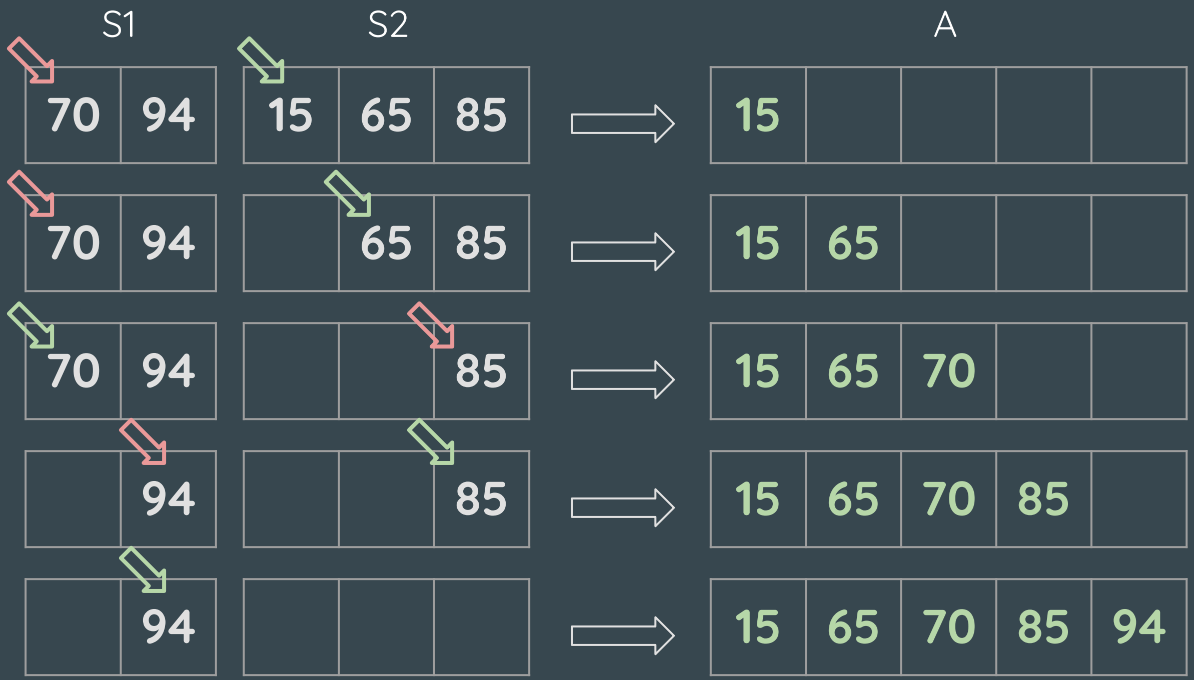
Merge sort

- Se basa en el principio de **divide y vencerás**.
- La lista de valores se divide en listas de menor tamaño (n//2) y estas listas se ordenan recursivamente mientras el número de valores sea >1.
- Cuando la lista tiene un solo valor está ordenada, y este resultado se devuelve a la llamada recursiva anterior.
- El algoritmo luego une o **mezcla** (*merge*) los resultados de las llamadas recursivas que hace.

Merge sort

- La clave de este algoritmo está en la función de mezclado o ***merge***.
- Esta función recibe dos sub-arrays **ordenados**, compara el primer elemento de cada subarray y copia ese elemento en una nueva (y **ordenada**) lista.

Función *merge*



Merge sort

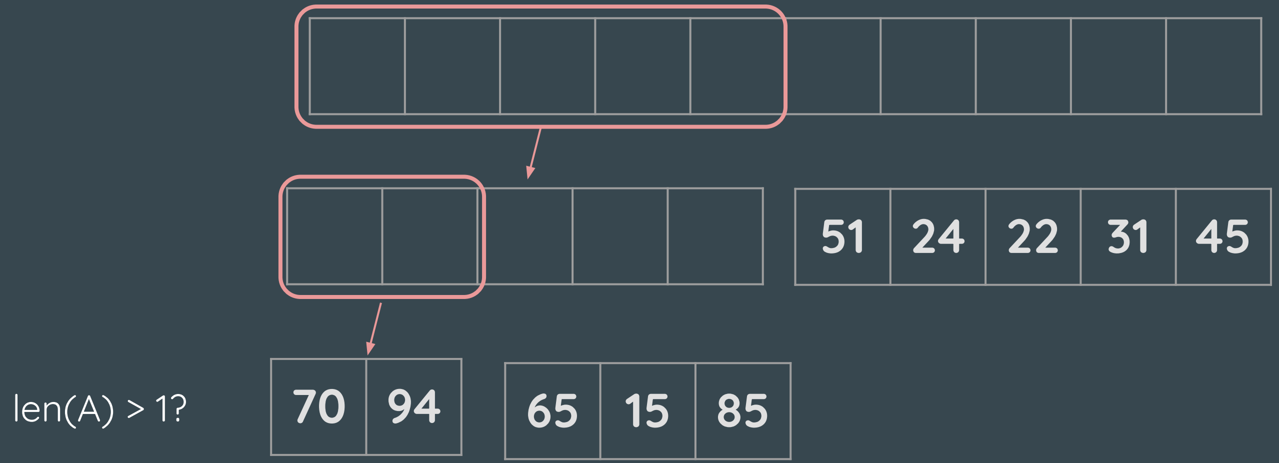
len(A) > 1?

70	94	65	15	85	51	24	22	31	45

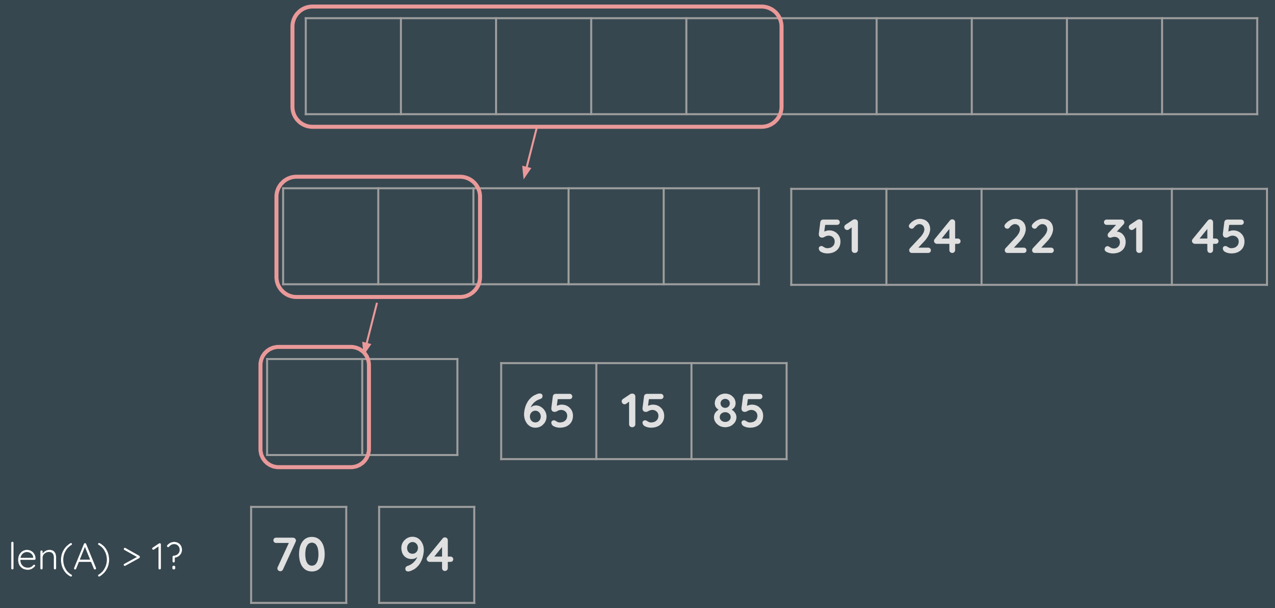
Merge sort

len(A) > 1?

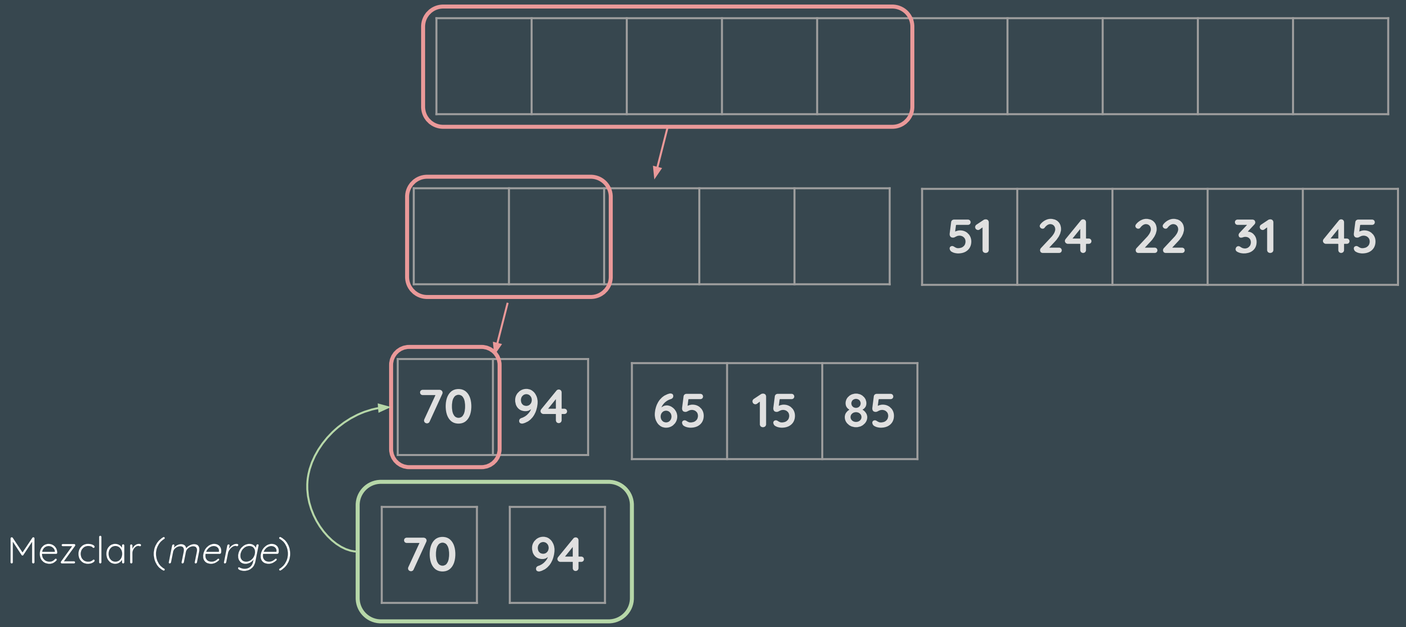
Merge sort



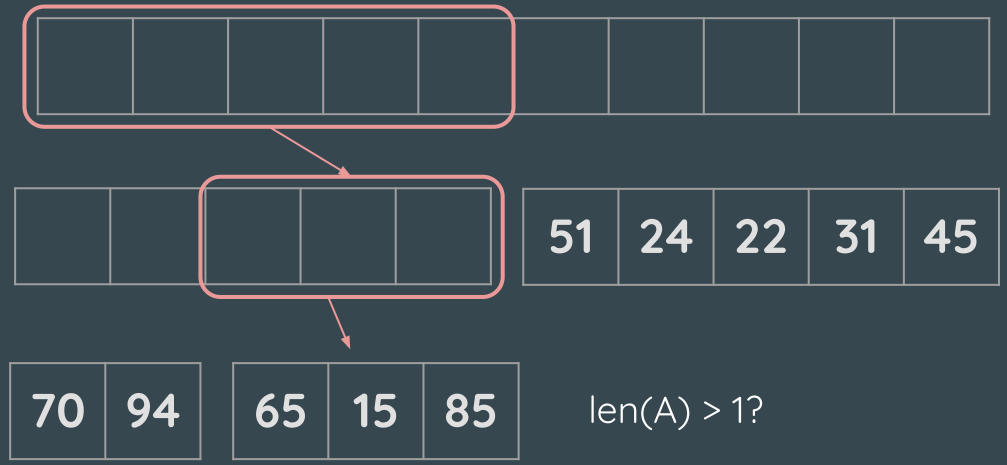
Merge sort



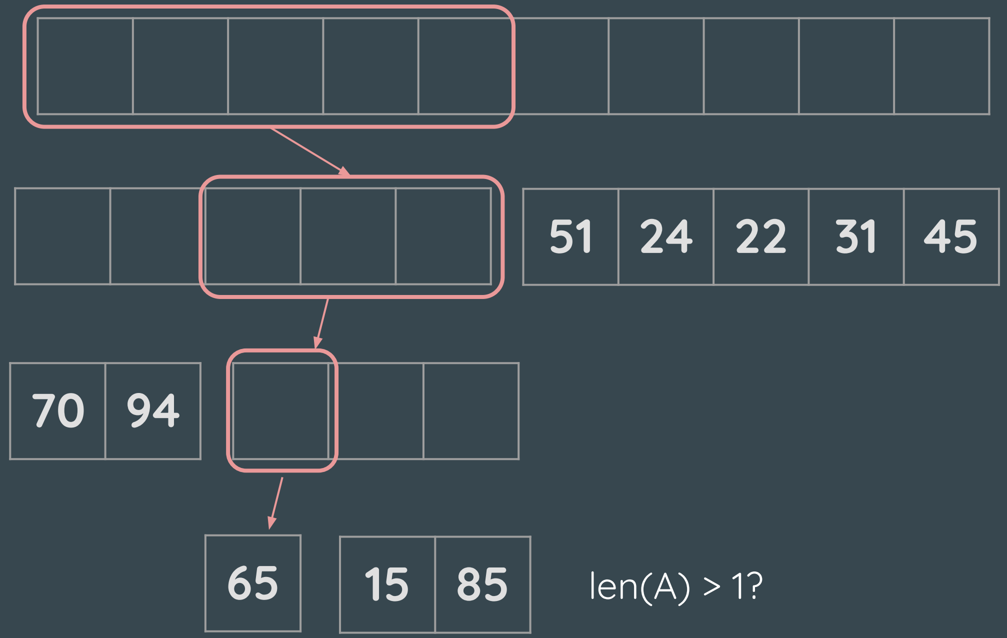
Merge sort



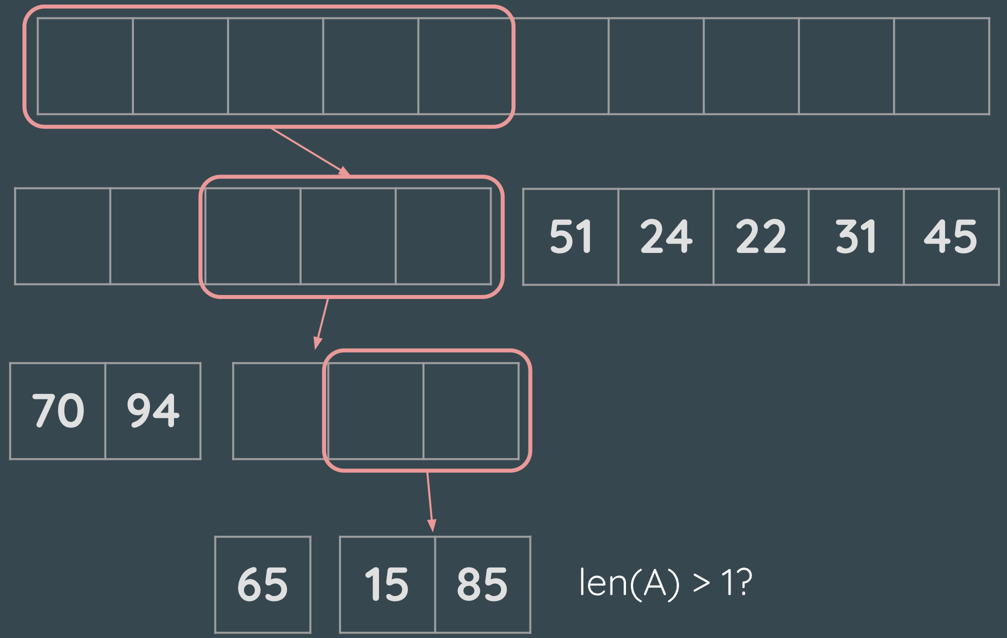
Merge sort



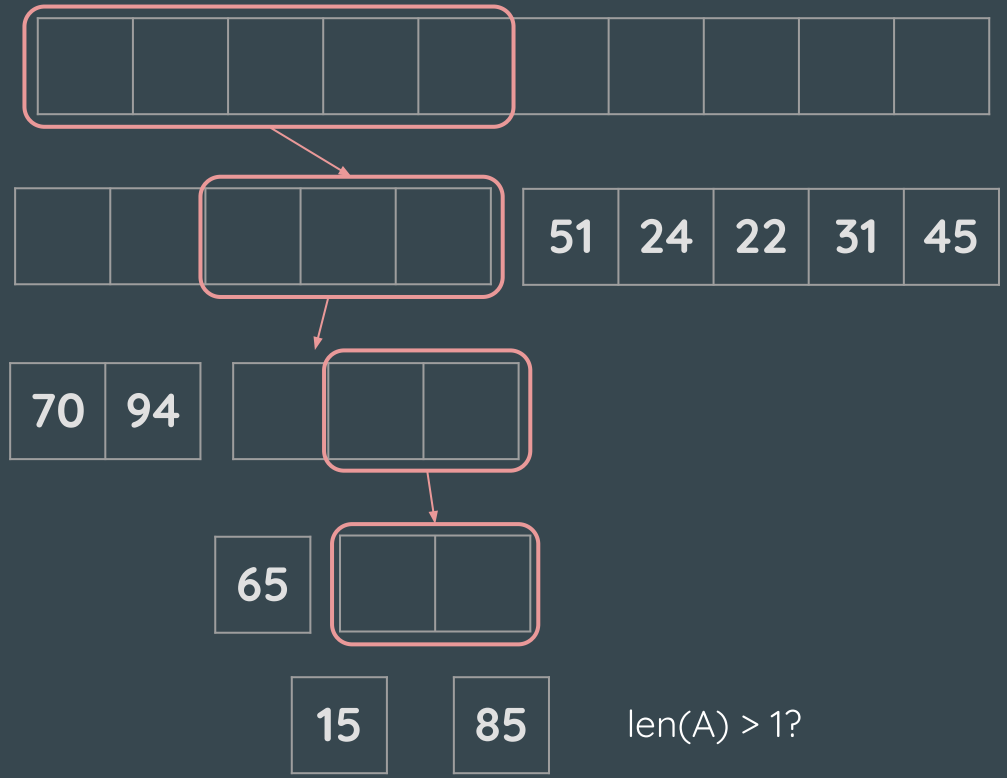
Merge sort



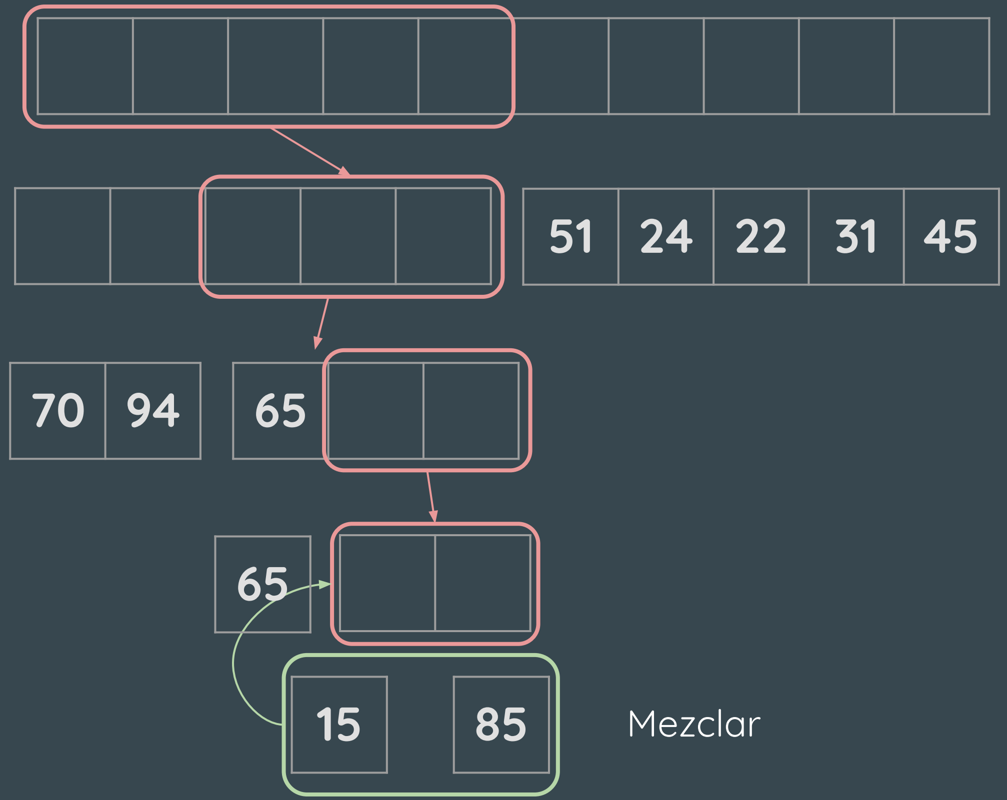
Merge sort



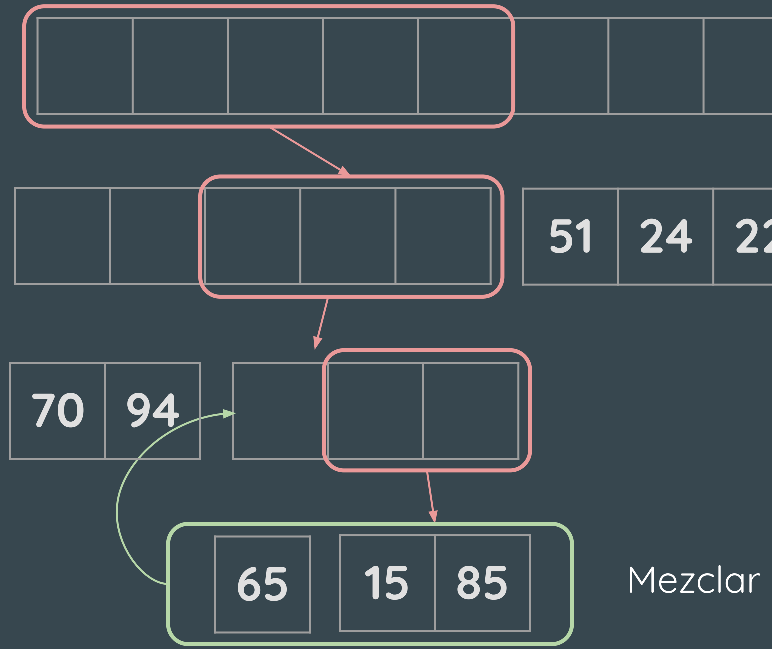
Merge sort



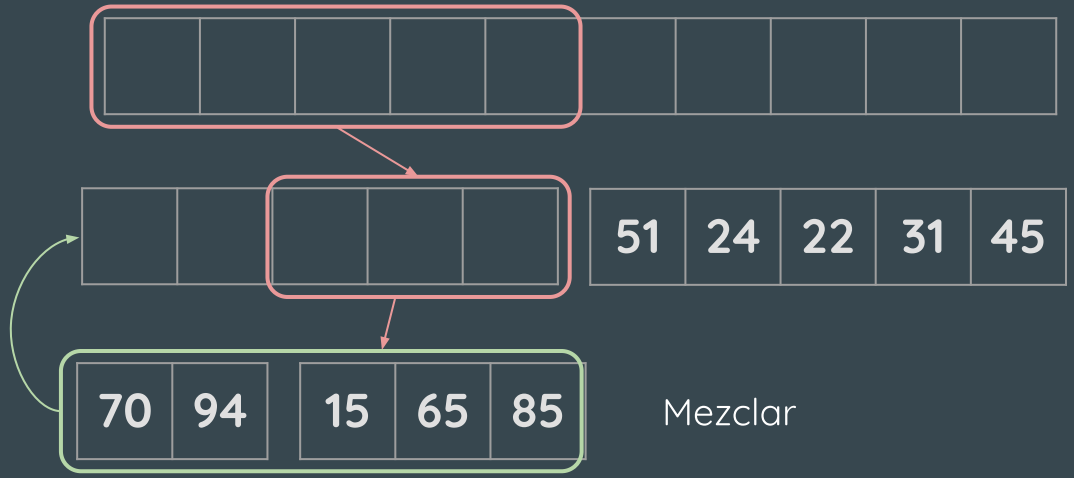
Merge sort



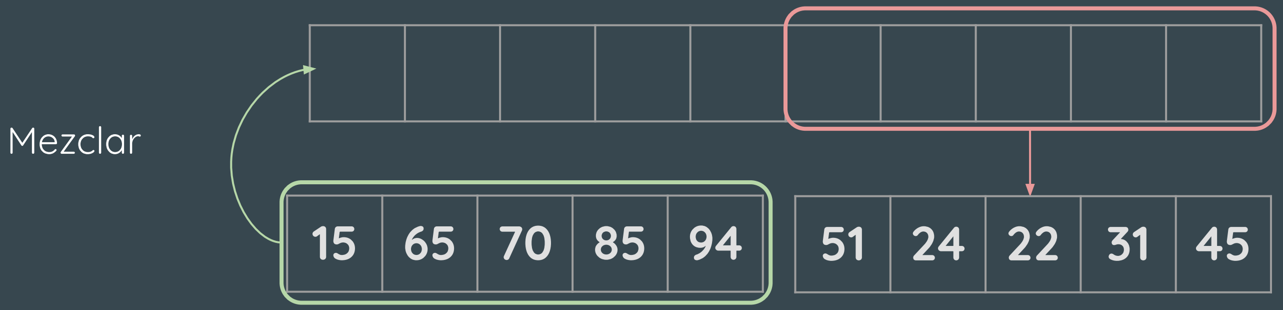
Merge sort



Merge sort



Merge sort



Merge sort: pseudocódigo

```
funcion merge(S1, S2):
  A = []
  i =0; j= 0
  mientras i < len(S1) y j < len(S2):
    si i == len(S1):
      A.append(S2[j]); j += 1
    sino, si j == len(S2):
      A.append(S1[i]); i += 1
    sino, si (S1[i]) < (S2[j]):
      A.append(S2[j]); j += 1
    sino:
      A.append(S1[i]); i += 1
  return A
```

```
funcion merge_sort(A):
  primer_mitad = A[0:len(A)//2]
  segunda_mitad = A[len(A)//2:len(A)]

  merge_sort(primer_mitad)
  merge_sort(segunda_mitad)

  A = merge(primer_mitad, segunda_mitad)
```

Complejidad *merge sort*

- El tiempo usado por el algoritmo depende de dos factores: la cantidad de divisiones y el ordenamiento de los subarrays.
- Al dividir, recursivamente, el arreglo en 2, se generan $\log_2(n)$ niveles.
- Cada nivel requiere un tiempo lineal para ser ordenado

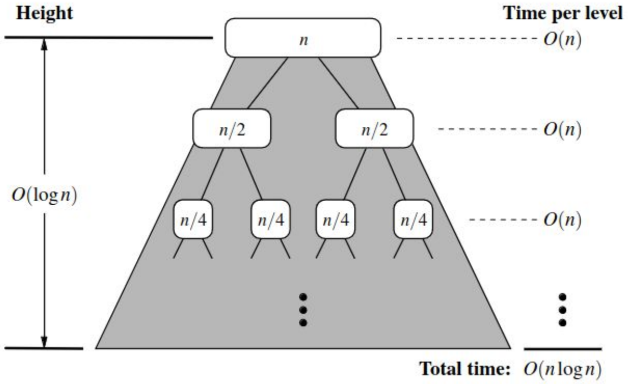


Figure 12.6: A visual analysis of the running time of merge-sort. Each node represents the time spent in a particular recursive call, labeled with the size of its subproblem.

Quick Sort

- También se basa en “divide y vencerás”, pero en este caso se selecciona un **pivot** para dividir la lista (por ejemplo, el último elemento de la lista).
- Los **elementos menores** al pivot se ponen en una lista, que se ordenará **recursivamente**.
- Lo mismo ocurre con los elementos **mayores** al pivot.

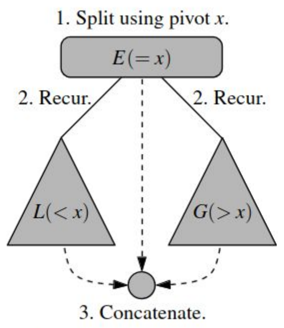


Figure 12.8: A visual schematic of the quick-sort algorithm.

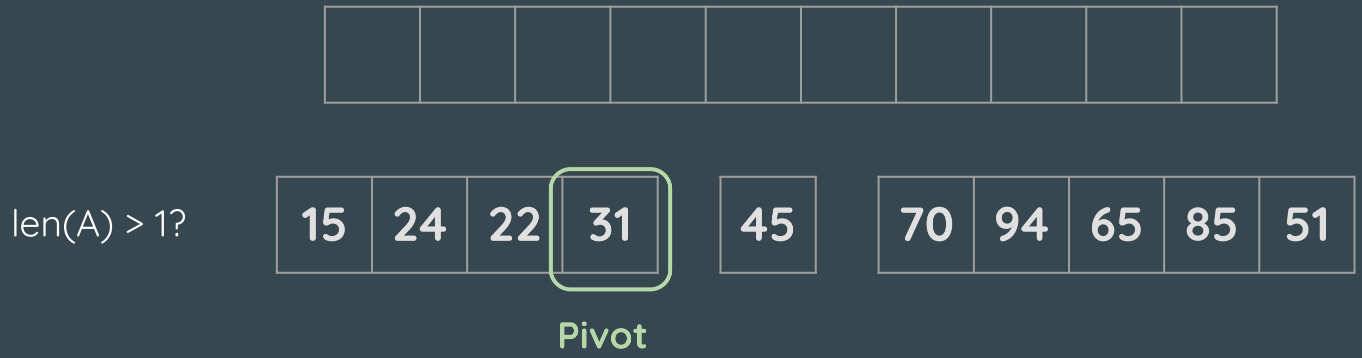
Quick sort

len(A) > 1?

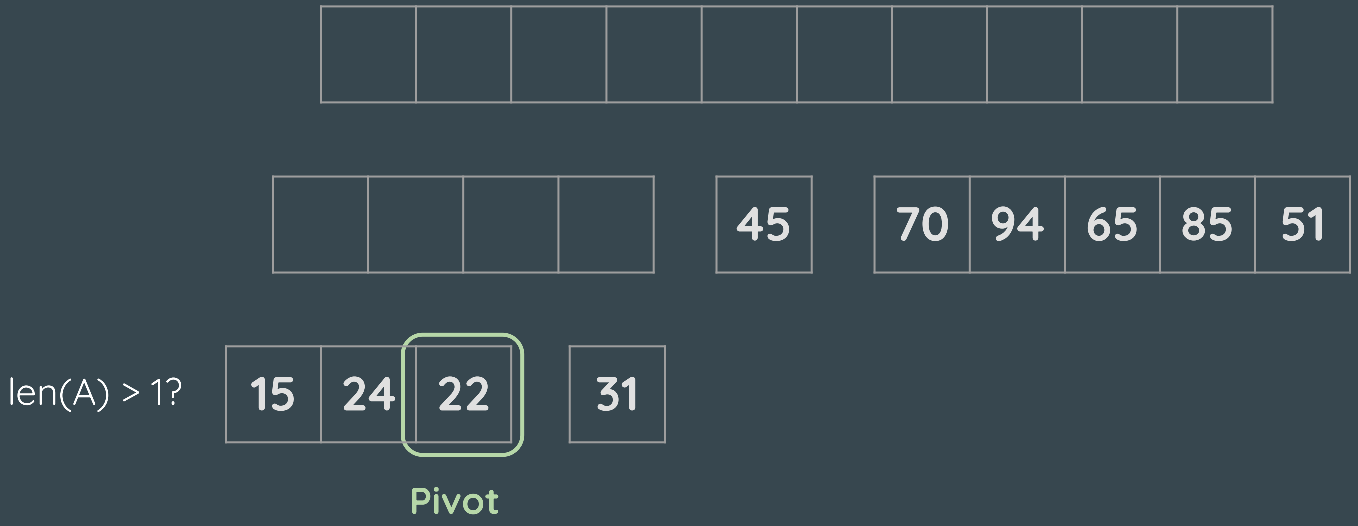
70	94	65	15	85	51	24	22	31	45

Pivot

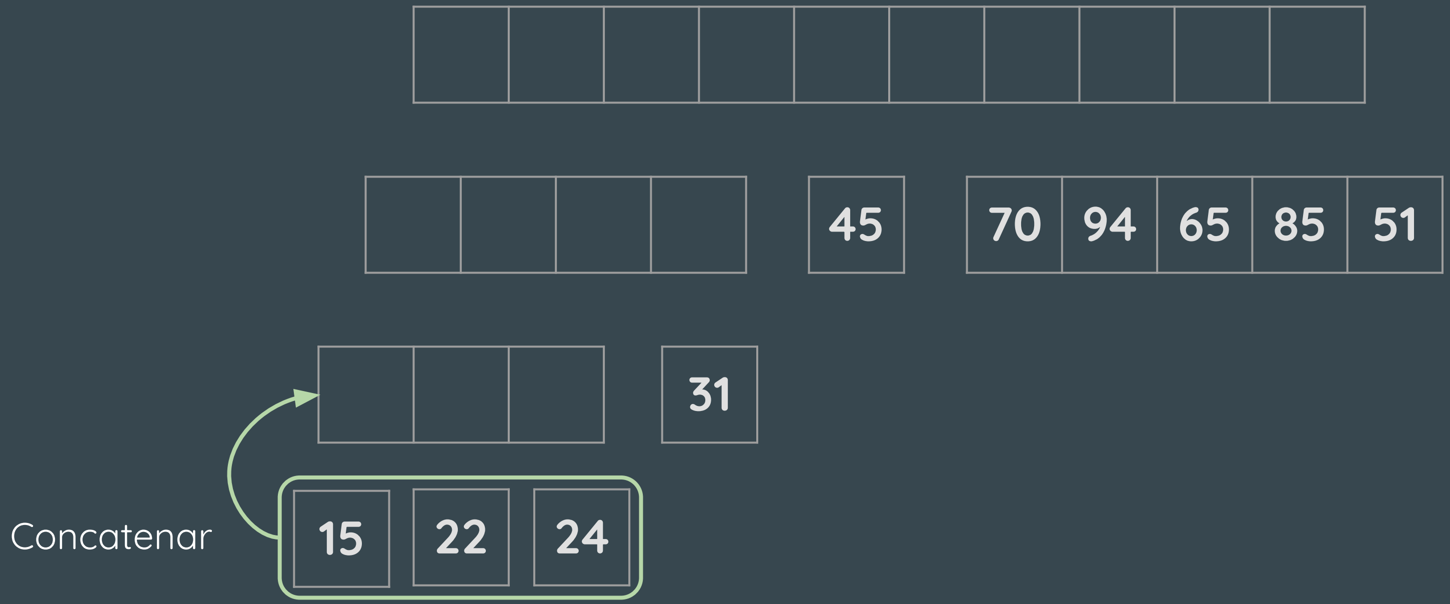
Quick sort



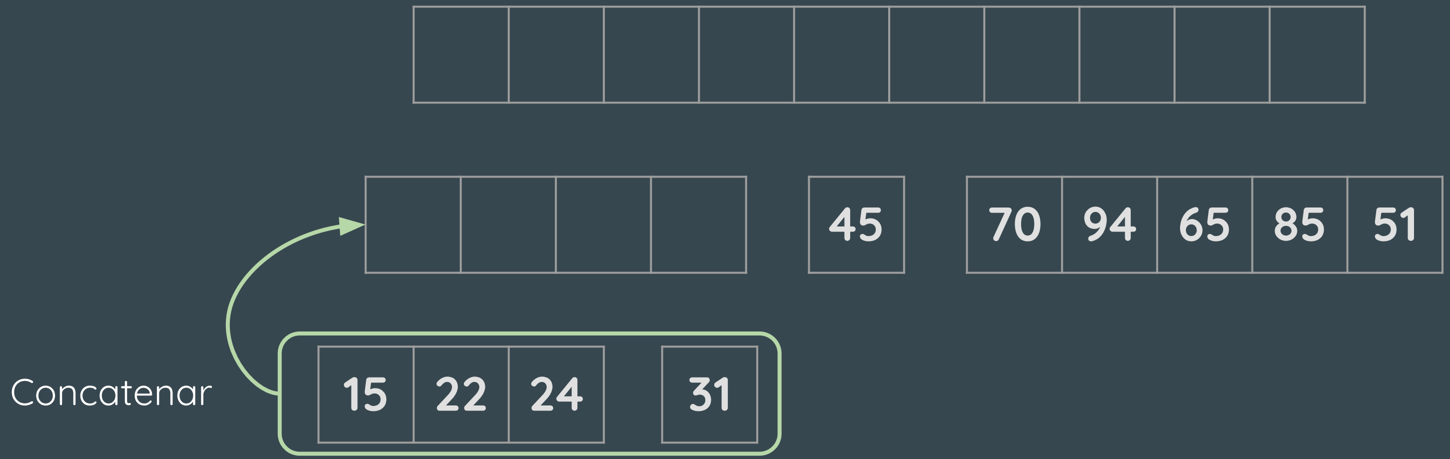
Quick sort



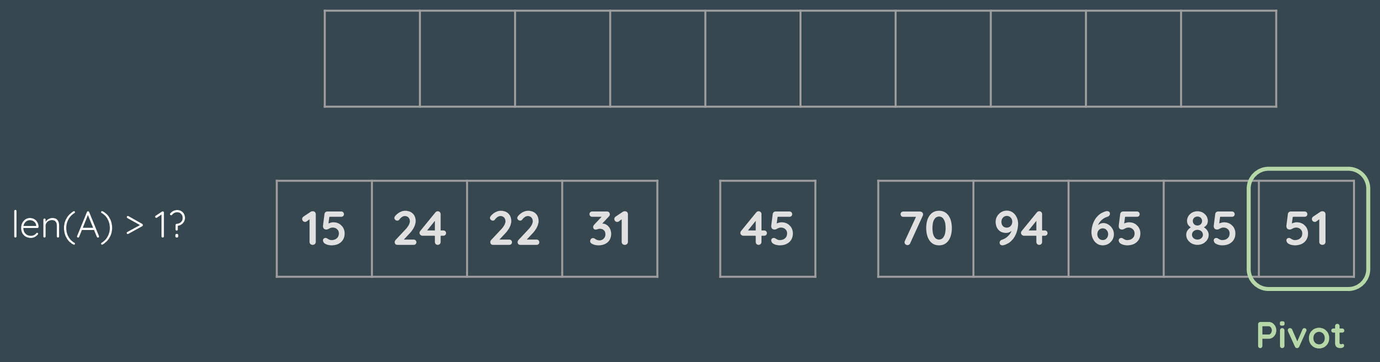
Quick sort



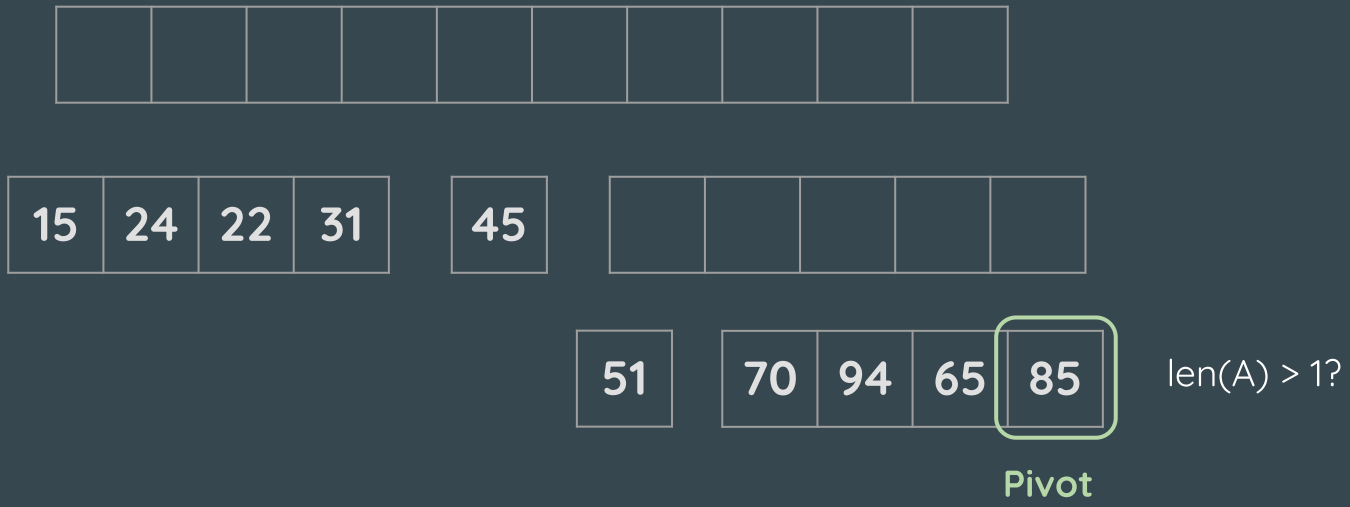
Quick sort



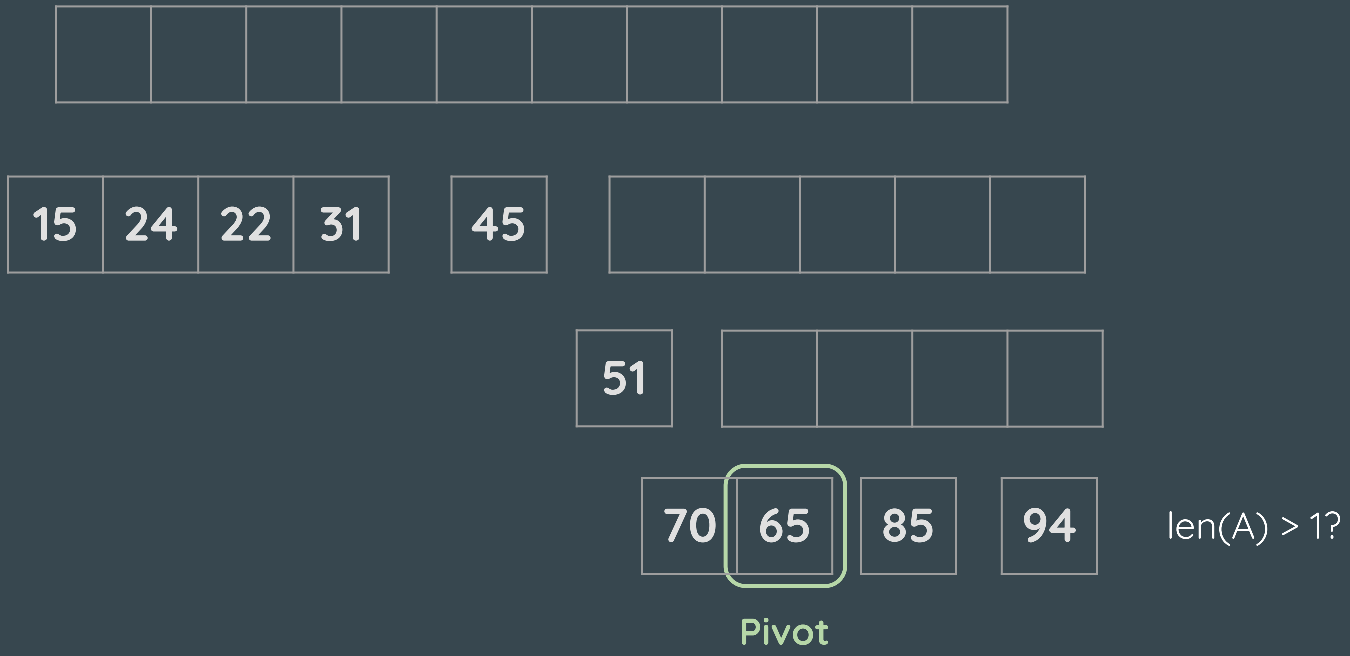
Quick sort



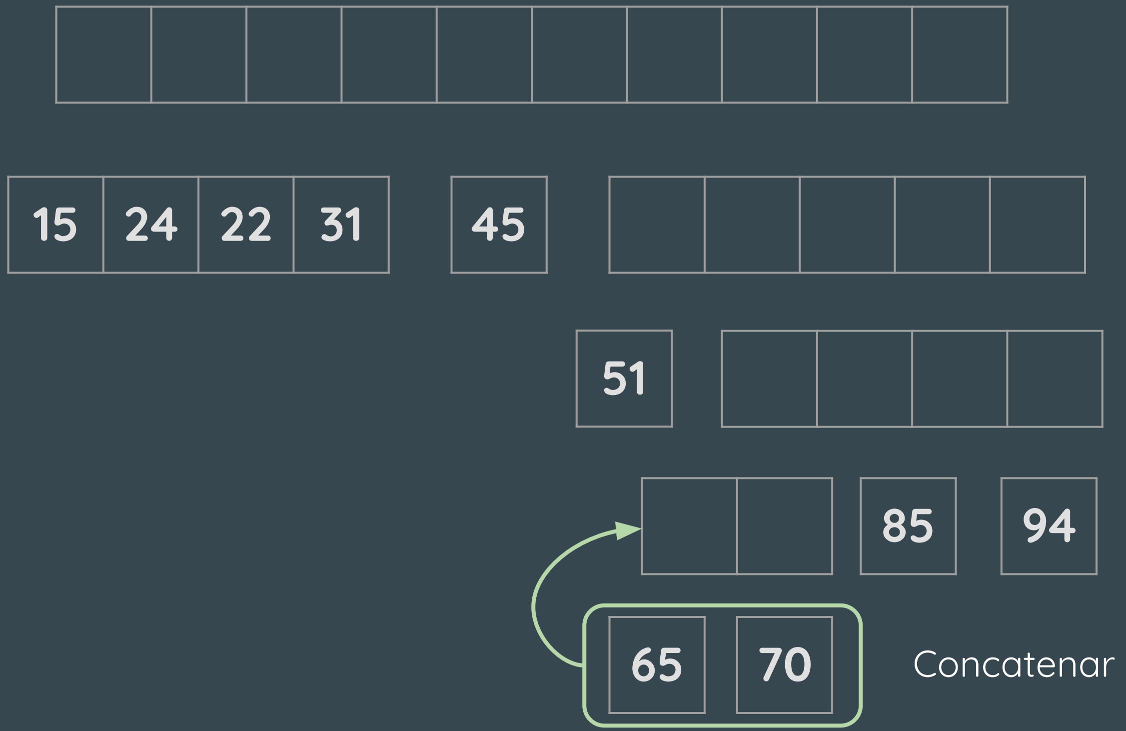
Quick sort



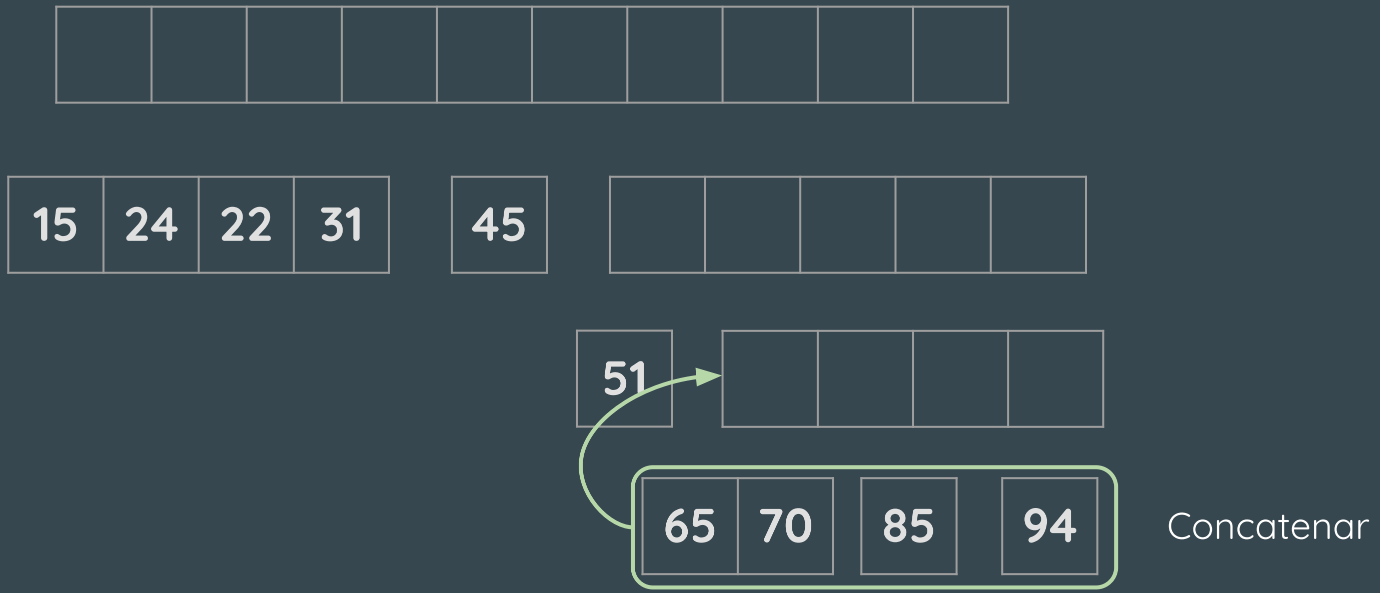
Quick sort



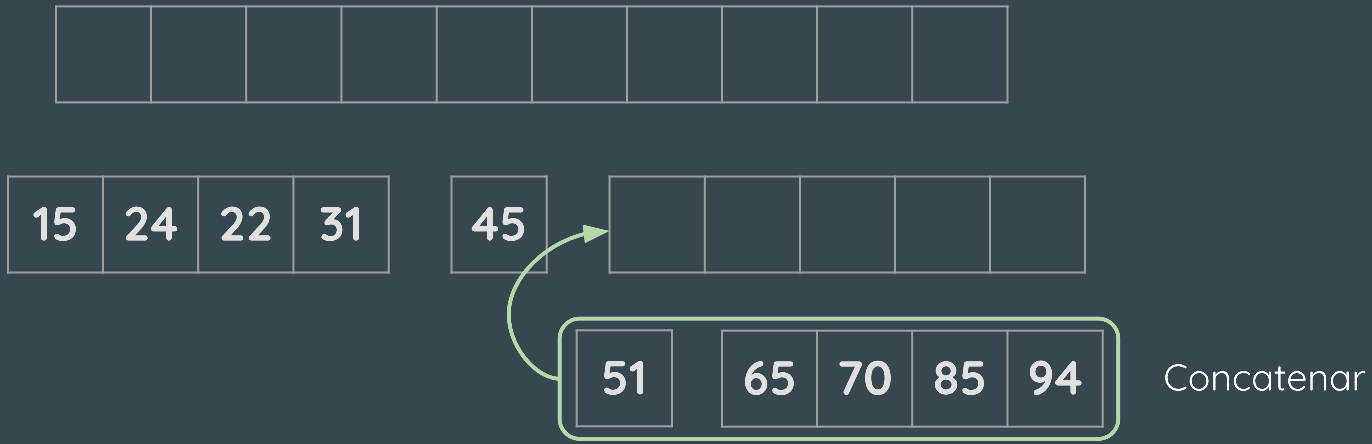
Quick sort



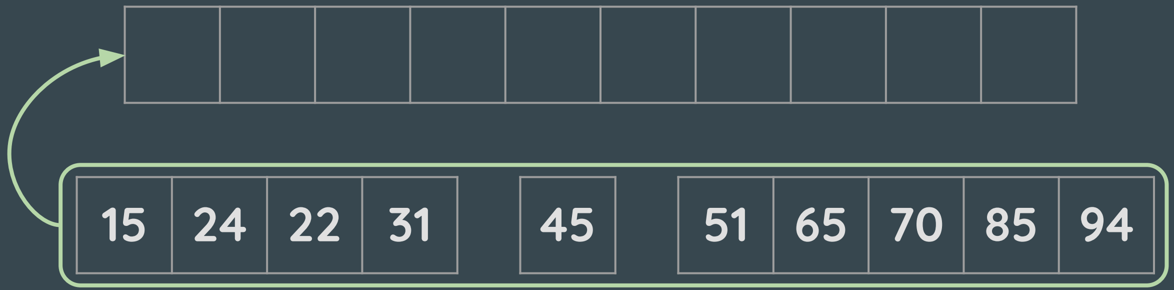
Quick sort



Quick sort



Quick sort



Concatenar

Quick sort: pseudocódigo

```
funcion quick_sort(A):
  si len(A) > 1:
    pivot = A[-1]
    menores = []; mayores = []; iguales = []
    por cada elemento en A:
      si elemento < pivot:
        menores.append(elemento)
      sino, si elemento > pivot:
        mayores.append(elemento)
      sino:
        iguales.append(elemento)

    menores = quick_sort(menores)
    mayores = quick_sort(mayores)

    A = menores + iguales + mayores
    devolver A
```

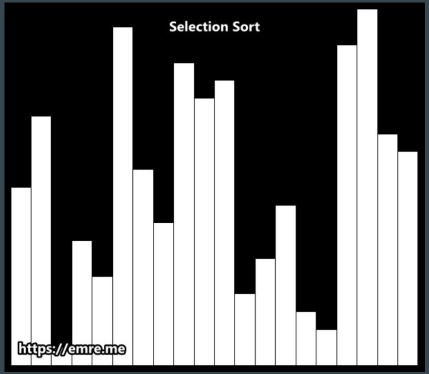
Complejidad de *quick sort*

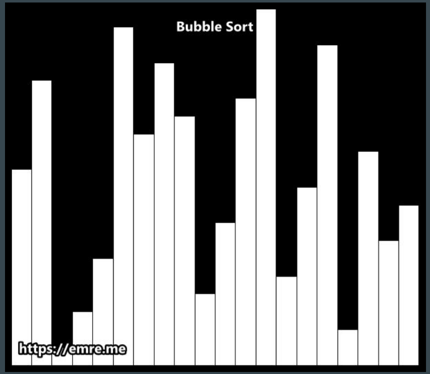
- La *performance* del algoritmo está directamente relacionada al pivot elegido.
- Si el número de elementos antes y después del pivot son similares, el número de llamadas recursivas disminuye.
- En promedio, la complejidad del algoritmo es **cuasi-lineal** ($\log_2(n)*n$), como en ***merge sort***, pero en el peor caso (lista ordenada), el tiempo crece de manera **cuadrática**.

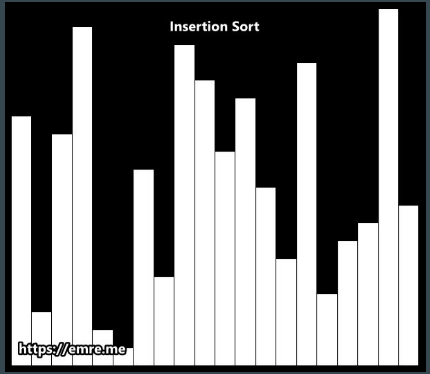
Complejidad de *quick sort*

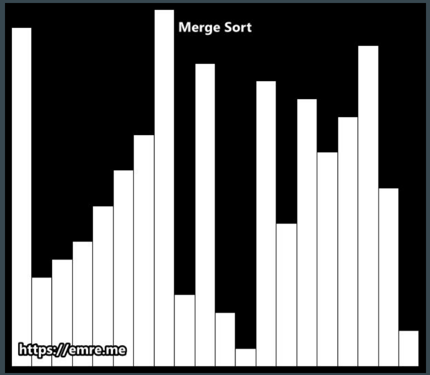
- Ha sido estudiado que la performance puede mejorar cambiando el criterio de selección del pivot por un elemento al azar dentro de la lista.
- El mayor problema del algoritmo rápido (y del de mezcla) es que pueden requerir mucho espacio adicional de memoria.

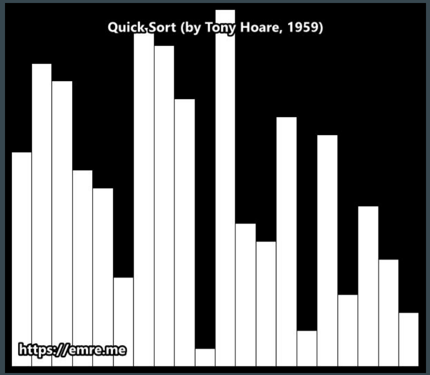
Comparación de tiempos











https://emre.me/algorithms/sorting-algorithms/

Bibliografía

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). Data structures and algorithms in Python. Secciones 5.5.2, 9.4.1, 12.2, 12.3 y 12.4.2
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press. SEcciones 2.1, 2.3.1, 7 y 8.4