Algoritmos y Estructuras de Datos
Tema 5: Sorting (Ordenación)
Grado Imat. Escuela ICAI

January 2024

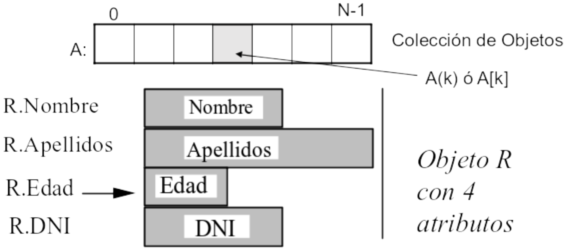
Algoritmos de Ordenación

- **Métodos directos simples**
  - Basados en inserción $\Rightarrow$ Inserción directa, binaria
  - Basados en selección $\Rightarrow$ Selección directa
  - Basados en intercambio $\Rightarrow$ Intercambio directo (burbuja y sacudida)
  - Basados en inserción $\Rightarrow$ Algoritmo de Shell
- **Sistemas Sofisticados**
  - Basados en partición recursiva: Quicksort
  - Basados en algoritmos no comparativos: BucketSort
  - Algoritmos de ordenación múltiple: Radix Sort
  - Basados en selección$\Rightarrow$ Montículo (Heapsort) (Basados en árboles)

COMILLAS ICAI

Ordenando Secuencias de Objectos o Estructuras

La ordenación ser realiza en base a una clave (o jerarquía de claves, para resolver los empates)



Sort by Edad, o DNI
Sort por Apellidos, luego por Nombre

Python hint

- Añade el método *__gt__(object)* a la clase, que retorna True is self es mayor que object
- alternativamente, da la función de comparación a la función de ordenación
  Ejemplo: `sorted(student_objects, key=attrgetter('age'))`

Part I

Algoritmos Simples de Ordenación

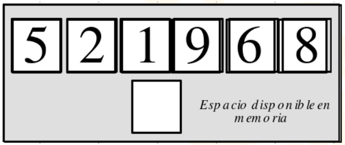
Section 1

Algoritmos de Inserción y Selección

Direct Insertion Sort I

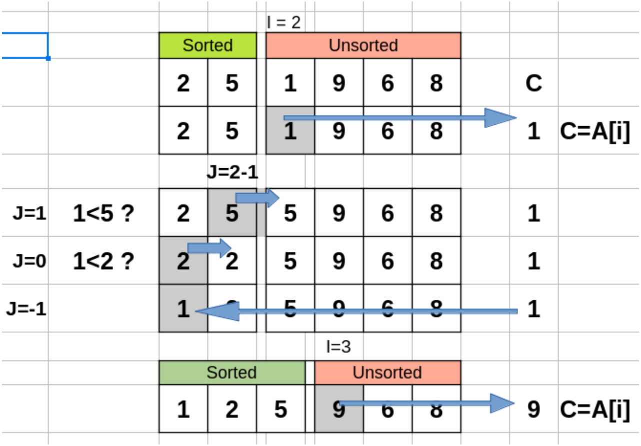
IDEA: toma un elemento de la conjunto desordenado e insértalo en la posición correcta de la zona ordenada

Sorting In-place Utilizando un vector de registros y ordenando sobre el mismo utilizando espacio auxiliar para un registro



Insertion Sort II

Insertion Sort Avanza la seccion ordenada, cogiendo el primero de la no ordenada y haciendo una búsqueda secuencial de la ubicación donde insertar el elemento seleccionado.



Insertion Sort II: algoritmo

```
function DirectInsertionSort(aList)
    for i=1 to len(aList)-1 do
        carta ← aList[i]                                  ▷ selecciona siguiente carta
        j ← i − 1                                         ▷ busca punto de inserción
        while j ≥ 0 and carta < aList[j] do               ▷ comparando y ...
            aList[j + 1] ← aList[j]                       ▷ shifting data
            j ← j − 1
        end while
        aList[j + 1] ← carta                              ▷ Inserta carta en el punto
    end for
    return
end function
```

Insertion Sort II: algoritmo

```
1:  function DirectInsertionSort(aList)
2:      for i=1 to len(aList)-1 do
3:          carta ← aList[i]                              ▷ selecciona siguiente carta
4:          j ← i − 1                                     ▷ busca punto de inserción
5:          while j ≥ 0 and carta < aList[j] do           ▷ comparando y ...
6:              aList[j + 1] ← aList[j]                   ▷ shifting data
7:              j ← j − 1
8:          end while
9:          aList[j + 1] ← carta                          ▷ Inserta carta en el punto
10:     end for
11:     return
12: end function
```

Complejidad ??

- **For** en línea 2 es de $\mathbf{O}(N)$
- **While** en línea 5 es de $\mathbf{O}(N)$
- Insertion Sort is de $\mathbf{O}(N) \times \mathbf{O}(N) = \mathbf{O}(N^2)$

Insertion Sort Mejorado: Usando búsqueda binaria

La inserción directa puede ser mejorada utilizando la búsqueda binaria, para encontrar la posición de la inserción más rápidamente.

```
function BinaryInsertionSort(aList)
    for i=1 to len(aList)-1 do
        carta ← aList[i]
        iz ← 0 ;  de ← i − 1
        while iz ≤ de  do  ▷ bisection Search
            m ← ⌊(iz + de)/2⌋
            if carta < aList[m] then
                de ← m − 1
            else
                iz ← m + 1
            end if
        end while
        for j=i-1 to iz by -1 do
            aList[j+1] ← aList[j]
        end for
        aList[iz] ← carta
    end for
    return
end function
```

Insertion Sort Mejorado: Usando búsqueda binaria

La inserción directa puede ser mejorada utilizando la búsqueda binaria, para encontrar la posición de la inserción más rápidamente.

- **Cuál es la complejidad ahora ?**
  - Iteración en línea 2 es $\mathbf{O}(N)$
  - búsqueda binaria del punto de inserción Líneas 5 a 12 es $\mathbf{O}(\log N)$
  - Data shift en líneas 13 a 15 es $\mathbf{O}(N)$
- **Total**

$$
\begin{aligned}
&= \mathbf{O}(N) \times \big(\mathbf{O}(\log N) + \mathbf{O}(N)\big) \\
&= \mathbf{O}(N^2)
\end{aligned}
$$

```
1:  function BinaryInsertionSort(aList)
2:      for i=1 to len(aList)-1 do
3:          carta ← aList[i]
4:          iz ← 0 ;  de ← i − 1
5:          while iz ≤ de  do              ▷ bisection Search
6:              m ← ⌊(iz + de)/2⌋
7:              if carta < aList[m] then
8:                  de ← m − 1
9:              else
10:                 iz ← m + 1
11:             end if
12:         end while
13:         for j=i-1 to iz by -1 do
14:             aList[j+1] ← aList[j]
15:         end for
16:         aList[iz] ← carta
17:     end for
18:     return
19: end function
```

Selection Sort I

IDEA: Busca el elemento **menor** del conjunto no ordenado e insértalo **al final** del subconjunto Ordenado

Sorting In-place Utilizando un vector de registros y ordenando sobre el mismo utilizando espacio auxiliar para un registro

Selection Sort II: algoritmo

```
function SELECTIONSORT(aList)
    for i=0 to N-2 do                          ▷ separación secuencia origen/destino
        k ← i , x ← A[i]                       ▷ inicializa búsqueda mínimo en origen
        for j=i+1 to N-1 do
            if (A[j] < x then
                k ← j ; x ← A[j]               ▷ busca mínimo en origen
            end if
        end for
        A[k] ← A[i] ; A[i] ← x                 ▷ Intercambio
    end for
    return
end function
```

Selection Sort II: algoritmo

```
1:  function SELECTIONSORT(aList)
2:      for i=0 to N-2 do                      ▷ separación secuencia origen/destino
3:          k ← i , x ← A[i]                   ▷ inicializa búsqueda mínimo en origen
4:          for j=i+1 to N-1 do
5:              if (A[j] < x then
6:                  k ← j ; x ← A[j]           ▷ busca mínimo en origen
7:              end if
8:          end for
9:          A[k] ← A[i] ; A[i] ← x             ▷ Intercambio
10:     end for
11:     return
12: end function
```

Complejidad ??

- **For** en línea 2 es de $\mathbf{O}(N)$
- Búsqueda secuencial **For** en líneas 3 a 8 es de $\mathbf{O}(N)$
- Intercambio (línea 9) es $\mathbf{O}(1)$

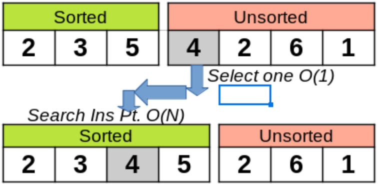
$\Rightarrow$ Selection Sort es de $\mathbf{O}(N) \times \left(\mathbf{O}(N) + \mathbf{O}(1)\right) = \mathbf{O}(N^2)$

Selection Sort and Insertion Sort: Similaridades

El trabajo duro por cada elemento es encontrar la posición de inserción que mantiene el orden del **sorted**, o buscar el mínimo valor en la secuencia **unsorted** (ambas tareas de orden $\mathbf{O}(N)$)

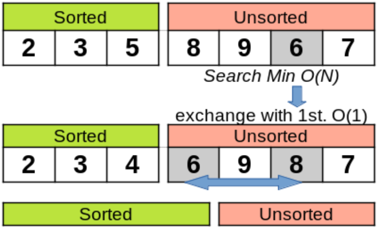
Insertion Sort

1. toma uno del **un-sorted**
2. busca la posición de inserción que garantiza el mantenimiento del orden en el **sorted** e inserta desplazando los elementos



Selection Sort

1. Busca el mínimo entre los **unsorted**
2. Intercambia con el ppio del **unsorted**, expandiendo el sorted una posición



Section 2

Merge Sort: Usando Divide and Conquer

Utiliza divide y vencerás:

- Divide el array en dos partes más pequeñas
- Se ordenan recursivamente. Si son suficientemente pequeñas se resuelve directamente
- Se combinan esas soluciones para ordenar finalmente el array

Mezcla Directa, Merge Sort: Algoritmo

La ordenación por Mezcla Directa, o **"Merge Sort"** es un ejemplo de Divide y Vencerás. La mayoría del trabajo se realiza en la composición de dos sublistas ya preordenadas para obtener una lista única ordenada.

Algorithm 1 Merge Sort D & C

```
function MergeSort(seq)
    if len(seq) = 2 then
        BasicSort(seg)
        return
    else
        left ← LeftHalf(seq)
        right ← RightHalf(seq)
        MergeSort(left)
        MergeSort(right)
        seq ← Merge(left,half)
    end if
    return seq
end function
```

Mezcla Directa, Merge Sort: Algoritmo

La ordenación por Mezcla Directa, o **"Merge Sort"** es un ejemplo de Divide y Vencerás. La mayoría del trabajo se realiza en la composición de dos sublistas ya preordenadas para obtener una lista única ordenada.

Algorithm 3 Merge Sort D & C

```
function MERGESORT(seq)
    if len(seq) = 2 then
        BASICSORT(seg)
        return
    else
        left ← LEFTHALF(seq)
        right ← RIGHTHALF(seq)
        MERGESORT(left)
        MERGESORT(right)
        seq ← MERGE(left,half)
    end if
    return seq
end function
```

Algorithm 4 Merge subfunction

```
function MERGE(left, right)
    merged ← empty sequence
    while left is not empty and right is not empty do
        if left is not empty then
            l0 ← PEEK(left)
            if right is not empty then
                r0 ← PEEK(right)
                if l0 < r0 then
                    l0 ← POP(left)
                    APPEND(merged,l0)
                else
                    r0 ← POP(right)
                    APPEND(merged,r0)
                end if
                APPEND(merged,left)
            end if
            APPEND(merged,right)
        end if
    end while
    return merged
end function
```

Complejidad Lineal-Logarítmica: Ejemplo MergeSort

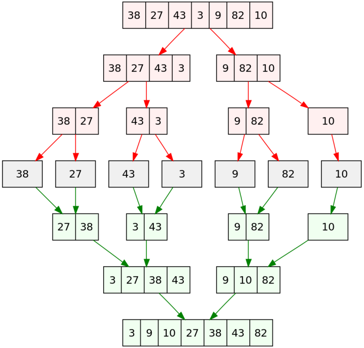
MergeSort

Divide la secuencia en dos, sobre los que invoca el Merge_sort de forma recursiva, para luego realizar un merge de la secuencias previamente ordenadas

Complejidad Lineal-Logarítmica: Ejemplo MergeSort

MergeSort

Divide la secuencia en dos, sobre los que invoca el Merge_sort de forma recursiva, para luego realizar un merge de la secuencias previamente ordenadas



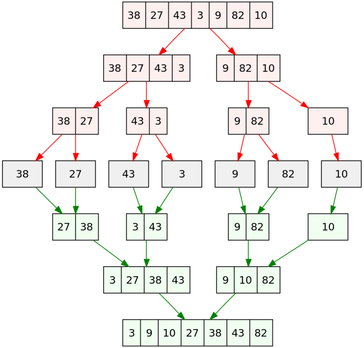
Complejidad Lineal-Logarítmica: Ejemplo MergeSort

MergeSort

Divide la secuencia en dos, sobre los que invoca el Merge_sort de forma recursiva, para luego realizar un merge de la secuencias previamente ordenadas

Cuál es la complejidad ?

- Fase de división



Complejidad Lineal-Logarítmica: Ejemplo MergeSort

MergeSort

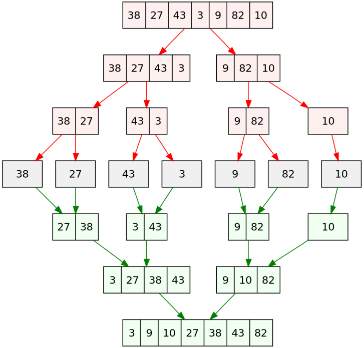
Divide la secuencia en dos, sobre los que invoca el Merge_sort de forma recursiva, para luego realizar un merge de la secuencias previamente ordenadas

Cuál es la complejidad ?

- Fase de división

  $\rightarrow$ $log_2(N) \times \mathbf{O}(1)$

- Cuántos niveles hay el el arbol durante el merge ?



Complejidad Lineal-Logarítmica: Ejemplo MergeSort

MergeSort

Divide la secuencia en dos, sobre los que invoca el Merge_sort de forma recursiva, para luego realizar un merge de la secuencias previamente ordenadas

Cuál es la complejidad ?

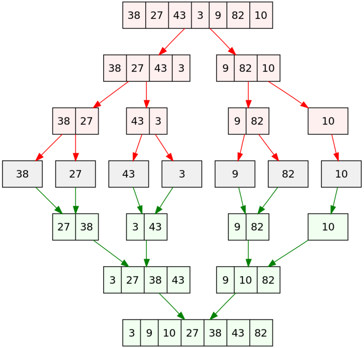
- Fase de división

  $\rightarrow$ $log_2(N) \times \mathbf{O}(1)$

- Cuántos niveles hay el el arbol durante el merge ?

  $\rightarrow$ $log_2(N)$

- Cuál es el coste de cada uno de ellos



Complejidad Lineal-Logarítmica: Ejemplo MergeSort

MergeSort

Divide la secuencia en dos, sobre los que invoca el Merge_sort de forma recursiva, para luego realizar un merge de la secuencias previamente ordenadas

Cuál es la complejidad ?

- Fase de división

  $\rightarrow$ $log_2(N) \times \mathbf{O}(1)$

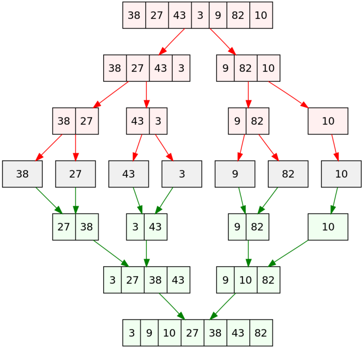
- Cuántos niveles hay el el arbol durante el merge ?

  $\rightarrow$ $log_2(N)$

- Cuál es el coste de cada uno de ellos

  $\rightarrow$ $\mathbf{O}(N)$

- Cuál es la complejidad resultante ?



Complejidad Lineal-Logarítmica: Ejemplo MergeSort

MergeSort

Divide la secuencia en dos, sobre los que invoca el Merge_sort de forma recursiva, para luego realizar un merge de la secuencias previamente ordenadas

Cuál es la complejidad ?

- Fase de división

  $\rightarrow$ $log_2(N) \times \mathbf{O}(1)$

- Cuántos niveles hay el el arbol durante el merge ?

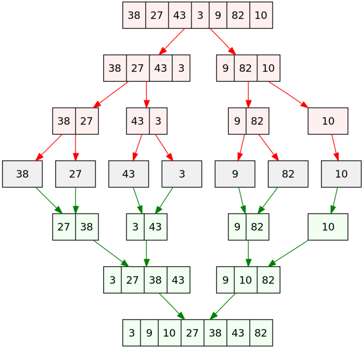
  $\rightarrow$ $log_2(N)$

- Cuál es el coste de cada uno de ellos

  $\rightarrow$ $\mathbf{O}(N)$

- Cuál es la complejidad resultante ?

  $\rightarrow$ $\log_2(N) * (\mathbf{O}(1) + \mathbf{O}(N))$
  $\rightarrow$ $\mathbf{O}(N * \log N)$



Section 3

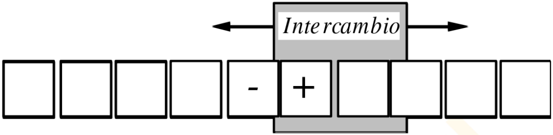
Algoritmos de Intercambio

- Bubble short (Intercambiando elementos adjacentes)
- Shaker Sort (Bidireccional Bubble)
- Shell Sort (Ordena subconjuntos de distancia decreciente)

Algoritmos de Intercambio

Característica dominante Intercambio entre pares de elementos. Mezcla de inserción y selección directa

Idea: Hacer repetidas pasadas sobre el array ordenando parejas de elementos contiguos



Ejemplos

- bubble Sort (Burbuja)
- Shake Sort (Sacudida)
- Shell Sort

Bubble short: Algoritmo

Bubble Sort realizada sucesivas pasadas intercambiando elementos anexos si no están relativamente ordenados, hasta que la ordenación es completa.

Bubble short: Algoritmo

Bubble Sort realizada sucesivas pasadas intercambiando elementos anexos si no están relativamente ordenados, hasta que la ordenación es completa.

```
function BUBBLESORT1(vec)
    n ← len(vec)
    repeat
        swap ← False
        for i=1 to n-1 do
            ▷ Perform Xchange as needed ◁
            if vec[i-1] > vec[i] then
                exchange vec[i-1] and vec[i]
                swap ← True
            end if
        end for
    until swap is False
    return
end function
```

Bubble short: Algoritmo

Bubble Sort realizada sucesivas pasadas intercambiando elementos anexos si no están relativamente ordenados, hasta que la ordenación es completa.

```
function BUBBLESORT1(vec)
    n ← len(vec)
    repeat
        swap ← False
        for i=1 to n-1 do
            ▷ Perform Xchange as needed ◁
            if vec[i-1] > vec[i] then
                exchange vec[i-1] and vec[i]
                swap ← True
            end if
        end for
    until swap is False
    return
end function
```

En cada pasada, el siguiente máximo elemento es empujado a su posición, y algunos otros elementos son reajustados.

```
pass: 0: [4, 2, 6, 7, 0, 3, 5]
pass: 1: [2, 4, 6, 0, 3, 5, 7] swaps=True
pass: 2: [2, 4, 0, 3, 5, 6, 7] swaps=True
pass: 3: [2, 0, 3, 4, 5, 6, 7] swaps=True
pass: 4: [0, 2, 3, 4, 5, 6, 7] swaps=True
pass: 5: [0, 2, 3, 4, 5, 6, 7] swaps=False
```

Bubble Sort: versión mejorada. Complejidad

El algoritmo puede ser optimizado al reconocer que la zona ordenada se incrementa en cada pasada. Por ejemplo:

```
function BubbleSort2(vec)
    n ← len(vec)
    for i=1 to n by 1 do                          ▷ Separar Origen y Destino
        for j=n-1 to i step -1 do                 ▷ Backward scan unsorted region
            if vec[j-1] > vec[j] then
                exchange vec[j-1] and vec[j]      ▷ perform exchange
            end if
        end for
    end for
    return
end function
```

Bubble Sort: versión mejorada. Complejidad

El algoritmo puede ser optimizado al reconocer que la zona ordenada se incrementa en cada pasada. Por ejemplo:

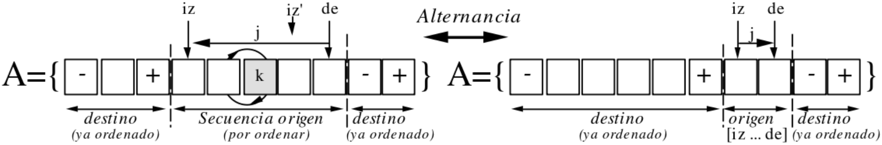
```
function BubbleSort2(vec)
    n ← len(vec)
    for i=1 to n by 1 do                          ▷ Separar Origen y Destino
        for j=n-1 to i step -1 do                 ▷ Backward scan unsorted region
            if vec[j-1] > vec[j] then
                exchange vec[j-1] and vec[j]      ▷ perform exchange
            end if
        end for
    end for
    return
end function
```

Complexity

En el peor de los casos, el Algoritmo es de $\mathbf{O}(N^2)$ ya sea por el anidamiento de dos loops de Orden N o por la N pasadas de orden N.
El coste depende grandemente del grado de ordenación previa de los datos.

Shaker Sort

La ordenación, por Sacudida, (Shaker Sort) es una optimización de la ordenación por intercambio, realizando simultáneamente el proceso por ambos extremos (hacia adelante y hacia atrás) hasta que las zonas ordenadas convergen en una.



Shaker Sort: Algoritmo

```
function ShakerSort(vec)
    de ← k ← len(vec)
    iz ← 1
    repeat
        for j = de to iz step -1 do                          ▷ Backward scan
            if vec[j-1] > vec[j] then
                exchange vec[j-1] and vec[j]                 ▷ perform exchange
                k ← j                                        ▷ Mark exchange position
            end if
        end for
        iz ← (k + 1)                                         ▷ Fija inicio del forward Scan
        for j = iz to de do                                  ▷ Forward scan
            if vec[j-1] > vec[j] then
                exchange vec[j-1] and vec[j]                 ▷ perform exchange
                k ← j                                        ▷ Mark exchange position
            end if
        end for
        de ← (k − 1)                                         ▷ resets right boundary
    until iz > de
    return
end function
```

Shaker Sort: Algoritmo

```
function ShakerSort(vec)
    de ← k ← len(vec)
    iz ← 1
    repeat
        for j = de to iz step -1 do                          ▷ Backward scan
            if vec[j-1] > vec[j] then
                exchange vec[j-1] and vec[j]                 ▷ perform exchange
                k ← j                                        ▷ Mark exchange position
            end if
        end for
        iz ← (k + 1)                                         ▷ Fija inicio del forward Scan
        for j = iz to de do                                  ▷ Forward scan
            if vec[j-1] > vec[j] then
                exchange vec[j-1] and vec[j]                 ▷ perform exchange
                k ← j                                        ▷ Mark exchange position
            end if
        end for
        de ← (k − 1)                                         ▷ resets right boundary
    until iz > de
    return
end function
```

Complejidad

Las mejoras del algoritmo son parciales, $\rightarrow$ **no cambia la escalabilidad del algoritmo**.
Al ser la sección `Repeat` de **O**(N) y $iz \sim de \sim$ **O**(N) $\Rightarrow$ Shaker Sort es de **O**(N^2)
El coste efectivo real es muy dependiente de la ordenación previo o parcial de los datos

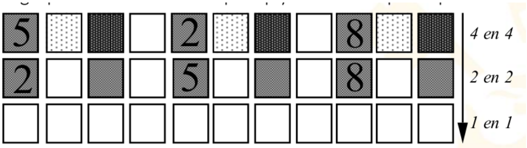
Shaker Sort: Ejemplo

- First Back Scan ends with j = 1 setting iz=2 for the Fow Scan
- Forward scan last exchange is at j =4, setting de=3
- Try again since iz =2 y de=3
- BackScan makes one change y fija iz =4
- ForwScan no hace cambios
- con iz=4 y de = 3, while breaks, hemos acabado

```
Initial              [4, 2, 6, 3, 8, 1]
Range [iz=1,de=5]

BackScan,[1<->8] [4, 2, 6, 3, 1, 8]
BackScan,[1<->3] [4, 2, 6, 1, 3, 8]
BackScan,[1<->6] [4, 2, 1, 6, 3, 8]
BackScan,[1<->2] [4, 1, 2, 6, 3, 8]
BackScan,[1<->4] [1, 4, 2, 6, 3, 8]

BackScan Done: k=1  Range [iz=2,de=5]

ForwScan,[2<->4] [1, 2, 4, 6, 3, 8]
ForwScan,[3<->6] [1, 2, 4, 3, 6, 8]

ForwScan, k=4  Range [iz=2,de=3]

BackScan,[3<->4] [1, 2, 3, 4, 6, 8]

BackScan Done: k=3  Range [iz=4,de=3]
ForwScan Done: k=3  Range [iz=4,de=2]
=> Done
```

Shell Sort

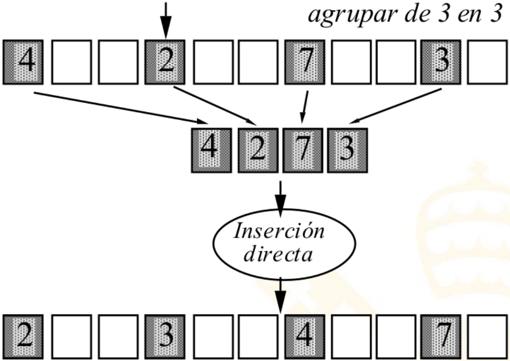
- Combina la inserción directa con el concepto de Divide y Vencerás
- Aplica la inserción directa ordenando subsets de datos con un **step** dado. Lo que aumenta la velocidad de los datos a través de la secuencia a velocidad de **step**.
- Se aplica en secuencia de *Steps* decrecientes, terminando necesariamente con *step* = *1*



- La secuencia de Steps óptima es clave en el proceso. Una Opciones válidas son comenzar en $\log_2(N)$ o $N/2$ y dividir por dos en cada iteración hasta que Step $= 1$
- Ejempo, para $n = 10,000$ una secuencia sería $25, 13, 7, 3, 1$

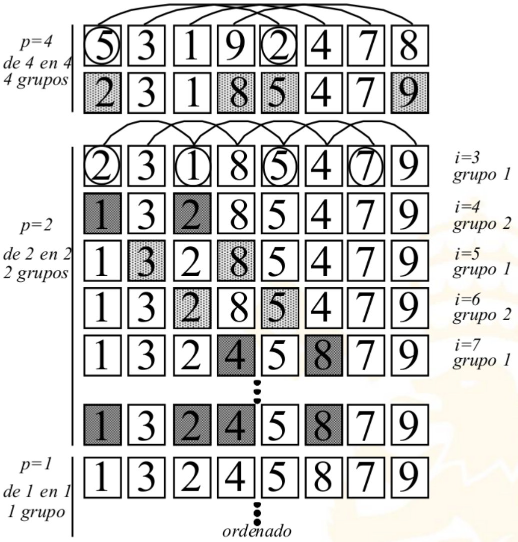
Shell Sort: Idea y Paralelizabilidad

Idea básica:



Al tratar con subsets disjuntos de datos, el algoritmo es altamente PARALELIZABLE, esto es un conjunto de CPUs puede trabajar de forma simultánea sobre los subsets $[i, i + step, i + 2 * step, i + 3 * step, ...]$ de forma simultánea y sin contención

Shell Sort: Ejemplo



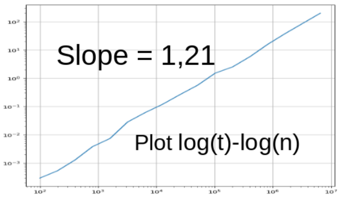
Shell Sort: Algorithm

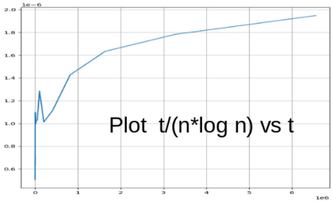
líneas 8 a 12 implementan una ordenación por inserción con elementos separadas por distancia = step.
Itera en el step, con valores decrecientes, hasta step = 1

```
1:  function ShellSort(vec,steps)
2:      ▷ steps: Decreasing sequence step < len(vec) ending in 1 ◁
3:      n ← len(vec)
4:      for step in steps do                          ▷ iterate along decreasing steps
5:          for i=step to n-1 by 1 do                 ▷ Start at step, till the end
6:              aux ← vec[i]
7:              j ← i − step                          ▷ search backward, stride = step
8:              while j >= 0 & (aux < vec[j]) do
9:                  vec[j + step] ← vec[j]            ▷ shifting if required
10:                 decrement j by step               ▷ Stride = step, backwards
11:             end while
12:             if i ≠ (j + step) then                ▷ Insertion at selected point
13:                 vec[j + step] ← aux
14:             end if
15:         end for
16:     end for
17:     return
18: end function
```

Shell Sort: Complejidad

- Siendo una mezcla de Insertion sort y de Divide y Vencerás, su análisis es complejo
- Depende de la preordenación de los datos, y de los steps seleccionados
- Es polinómica, i.e. $\sim n^a \; a \in [1, 2]$ esto es:
  - En el peor de los casos es como un algoritmo ingenuo de inserción $\sim n^2$
  - En el mejor de los casos se comportará algo peor que el algoritmo de mezcla recursiva $\sim n * \log n$
- Experimentalmente, se puede ver que el algoritmo de la página anterior es $\sim n^{1.21}$ y ciertamente no escala como $n * \log n$ según se indican en las figuras siguientes:





Part II

Algoritmos Avanzados de Ordenación

Quicksort: Algoritmo de particionado

Counting Sort: Algoritmo basado en frecuencia para claves de universo finito

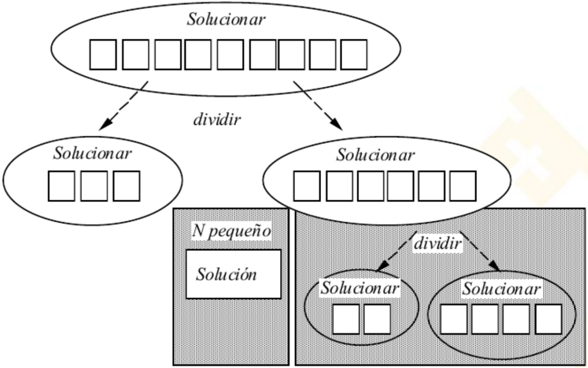
BinSort: algoritmo de agrupación para claves continuas o categorizables

RadixSort: Algoritmo multiclave con ordenación jerárquica

‘

QuickSort

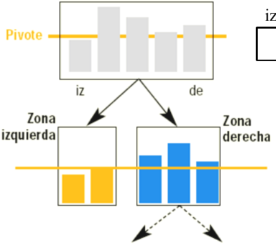
- QuickSort es un método de Ordenación recursivo que funciona por partición recursiva en segmentos cuyos elementos son siempre superiores o inferiores a cualquier elemento de otro segmento
- El proceso de partición es análogo al método de intercambio, pero aquí las distancias son tan grandes como posible.
- En base a las propiedades del proceso de partición $(\forall x, y\,;\, x \in p_l\,;\, y \in p_r \Rightarrow x > y)$ no es necesario realizar funciones de merge, como el algoritmo de mecla directa.
- Recursividad finaliza cuando la partición es de tamaño 1 o 2, que es ordenada de forma trivial.

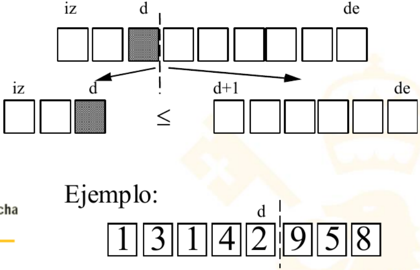


QuickSort: particionado

Cada sub-array A[iz ... de] se particiona en dos más pequeñas A[iz ... d] y A[d+1...de], a partir del índice d, tal que:

$$i \in [iz, .., d-1] \, ; \, j \in [d, ..., de-1] \rightarrow A[i] < A[j]$$





El índice d se determina durante el proceso de división a partir de un valor umbral o "pivote"

La clave está en la selección del umbral *d*.

partition divide

```
function QS_DIVIDE(vec,pivot)
    ▷ partitions vec, vec[pivot] as the threshold ◁
    threshold ← vec[pivot]
    vec[0] ↔ vec[pivot]                                ▷ place pivot first
    l ← 1
    r ← len(vec) − 1
    while l < r do                                     ▷ work on both sides
        L_r ← (vec[l] > threshold)                     ▷ Right and left tests...
        R_r ← (vec[r] =< threshold)
        if L_r ∧ R_r then
            vec[l] ↔ vec[r]                            ▷ swap elements
        end if
        increment l ; decrement r                      ▷ in all cases advance pointers
        if L_r ∧ ¬R_r then
            decrement l                                ▷ undo left ptr advance
        else if ¬L_r ∧ R_r then
            increment r                                ▷ undo right ptr advance
        end if
    end while
    if vec[j] > threshold then                         ▷ final adjusment
        return l
    else
        return l + 1
    end if
end function
```

Este algoritmo es claramente $\mathbf{O}(N)$

```
function QUICKSORT(vec,start,stop)
    if len(vec) ¡= 2 then
        Manually sort vec
        return vec
    else
        d ← QS_DIVIDE(vec)                            ▷ perform quick Sort Divide
        QUICKSORT(vec,start, start+d-1)               ▷ Recurse on Left
        QUICKSORT(vec,start+d, stop)                  ▷ Recurse on right
    end if
end function
```

La profundidad del algoritmo depende del equilibrio del proceso de divide

qs_divide Ejemplo

Take 5, vec[0] as the pivot, randomly

		l = 1						r=7	
Take 5, vec[0] as the pivot, randomly	4	3	1	9	8	5	1	2	Vec[7] <= umbral es True
		→			no swap			■	

			l=2					r=7	
Vec[2] > umbral es Falso	4	3	1	9	8	4	1	2	Vec[7] <= umbral es True
			→		No swap			■	

				L=3				r=7	
Vec[3] > umbral es True	4	3	1	9	8	4	1	2	Vec[7] > umbral es True
				→		Swap		←	

					l=4		r=6		
Vec[3] > umbral es True	4	3	1	2	8	4	1	9	Vec[6] > umbral es True
					→	⇔ swap	←		

Finish condition

						l=5 r=6			
not(vec[l] > vec[0] → d= l +1	4	3	1	2	1	4	8	9	
							d=6		return d=6

All elements in d[6:] are > than pivot. All elements in d[:d} are <= than pivot. Pivot = vec[0]

Quick Sort: Pivot Choice

Hay varias opciones, algunas simples, otras costosas.

1. **Random Choice**. Tomar un elemento al azar. No garantiza que la división sea equilibrada
2. **Primero o último elemento**. Equivalente a lo anterior, pero puede caer en el peligro de ordenaciones previas aproximadas de los datos, lo que perjudica la performance
3. **Mediana** Es la solución óptima, pero costosa

Finding the Median

Respuesta obvia: tomar el elemento central de la serie una vez ordenada [con un coste de $\mathbf{O}(n \log n)$] **Pero si lo que queremos es ordenarla !**
$\Rightarrow$ Afortunadamente, el algoritmo quickSelect[a] es capaz de producir la mediana en $\mathbf{O}(N)$

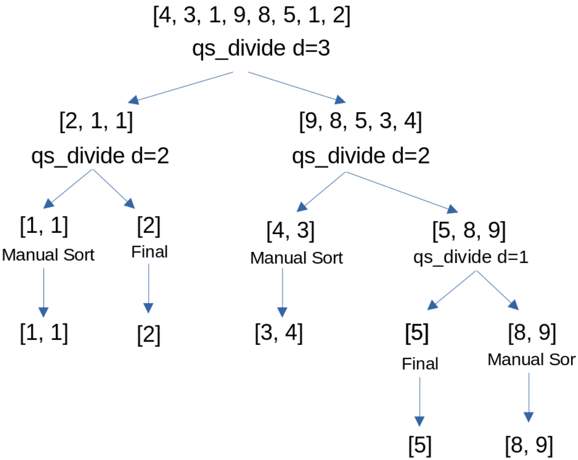
[a] de T. Hoare, el inventor del quicksort

QuickSort. QuickSelect Algorithm

QuickSelect Median algorithm encuentra la mediana aplicando recursivamente el algoritmo de qs_divide hasta que el pivote se encuentre a la mitad de la longitud, asegurando que el pivote es la mediana

```
function FINDMEDIAN(vec,start,stop)
    if len(vec) is odd then
        val ← FINDORDEREDVAL(vec, 0, len(vec)/2)
    else
        val1 ← FINDORDEREDVAL(vec, 0, (len(vec) − 1)/2)
        val2 ← FINDORDEREDVAL(vec, 0, (len(vec) + 1)/2)
        val ← (val1 + val2)/2
    end if
    return val
end function
procedure FINDORDEREDVAL(vec, offset, targetPos)
    if len(vec) = 1 then                                          ▷ Terminal Case
        Return vec[0]
    else
        d ← QS_DIVIDE(()vec)                      ▷ applies qs_divide with random pivot
        if offset + d > targetPos then
            val ← FINDORDEREDVAL(vec[:d], offset, target)             ▷ Busca en izq
        else
            val ← FINDORDEREDVAL(vec[d:], offset+d, target)       ▷ Busca en derecha
        end if
        return val
    end if
end procedure
```

Quick Sort aplica recursivamente el algoritmo qs_divide con pivots aleatorios u, opcionalmente cerca de la mediana, hasta longitudes finales de < 2.



Quick Sort
Complexity, pros and cons

Time Complexity

- **Best Case: $\boldsymbol{\Omega}(Nlog(N))$.** The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.Balanced partitions, leads to efficient Sorting.
- **Average Case: $\boldsymbol{\Theta}(Nlog(N))$** Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
- **Worst Case: O(N2)** The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions.
- **Auxiliary Space: O(1)**, if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O(N).

QuickSort: Pros and cons

Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort

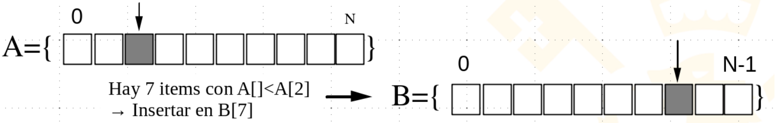
- It has a worst-case time complexity of O(N2), which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output.

Section 4

Counting Sort

Counting Sort

- Counting Sort is un algoritmo de ordenación que no está basado en la comparación de los valores y que se base en la categorización ordenada del rango de valores del vector.
- Funciona bien cuando el rango de los valores a ordenar es muy inferior al tamaño del vector
- El algoritmo se beneficia de la reducida complejidad de los datos. La idea básica es contar la frecuencia de aparición de cada uno de los distintos elementos del vector a ordenar, usando esta información para reposicionar ésto en su correcta posición en el vector ordenado.



Counting Sort How it Works

1. allocar CountVec de tamaño K+1, contar frecuencias y acumular $\Rightarrow \mathbf{O}(N)$
2. Tomando los elementos del vector a ordenar, en orden inverso:
   - Utilizar el valor como clave en vector de frecuencias acumuladas
   - Utilizar la frec accumulada como índice (restar 1) en el vector de destino
   - decrementar la la posicion usada en el vector de frecuencia

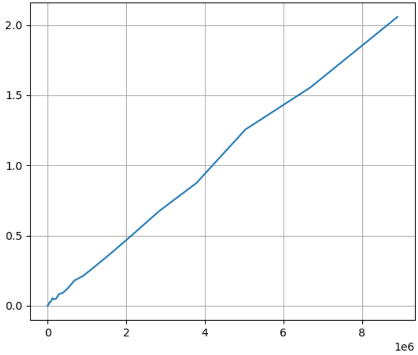
VEC =>	3	6	4	1	3	4	1	4	2	Count Frequencies and Accumulate
	Kmax = 6									
k	0	1	2	3	4	5	6			
VecCount	0	2	1	2	3	0	1			
Acc VecCount	0	2	3	5	8	8	9			
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	
VEC =>	3	6	4	1	3	4	1	4	2	Pos Vec(8)
Acc VecCount	0	2	3	5	8	8	9			3-1 = 2
SortedVec	-	-	2	-	-	-	-	-	-	Dec index 2
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	
VEC =>	3	6	4	1	3	4	1	4	-	Pos Vec(7)
Acc VecCount	0	2	2	5	8	8	9			8-1 = 7
SortedVec	-	-	2	-	-	-	-	4	-	Dec index 4
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	
VEC =>	3	6	4	1	3	4	1	-	-	Pos Vec(6)
Acc VecCount	0	2	2	5	7	8	9			2-1 = 1
SortedVec		1	2	-	-	-	-	4	-	Dec index 1
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	
VEC =>	3	6	4	1	3	4	-	-	-	Pos Vec(5)
Acc VecCount	0	1	2	5	7	8	9			7-1 = 6
SortedVec		1	2	-	-	-	4	4	-	Dec index 4

Counting Sort: Algorithm

```
function COUNTSORT(vec)
Require: vec contiene claves positivas óptimamente de tamaño << len(vec)
    k ← max(vec)
    ▷ allocate counting buffer and output vector ◁
    countVec ← Array() of length k+1, initialized to 0s
    sortecVec ← Array() of length= len(vec)
    ▷ Compute Keys frequencies                        ▷ O(N) ◁
    for item in Vec do
        increment countVec[item]
    end for
    ▷ place the items in proper order as per frequencies ◁
    for item in reversed(vec) do                      ▷ O(N)
        newPos ← (countVec[item]-1)
        sortecVec[newPos] ← item
        decrement countVec[item]
    end for
    return sortedVec
end function
```

Complejidad

- **Time Complexity:** O(N+M), donde N y M son el tamaño de Vec y CountVec respectivamente. Los casos Mejor, Peor y medio son todos de $\mathbf{O}(N+M)$.
- **Auxiliary Space:** $\mathbf{O}(N+M)$ que son los tamaños de los buffers adicionales a alocar.



Debido a las condiciones de las claves a ordenar, podemos hacerlo con compleiidad lineal !!

Counting Sort: Pros, and Cons

Ventajas del Counting Sort

- En general, Counting sort ejecuta más rápidamente que los métodos de ordenación basados en comparación, como merge Sort, o QuickSorticksort, inclusio si el rango de los valores del vector a ordenar es del orden del tamaño del vector $M \sim N$
- Counting sort tiene un código muy sencillo
- Counting sort es un algoritmo Estable (mantiene el orden de los elementos con clave idéntica)

Desventajas del Counting Sort

- Counting sort no funciona si los datos son decimales. Sólo funciona con datos enteros o categorías que se pueden mapear a un rango $[0, \ldots n]$
- Counting sort no es eficiente si el rango de los valores a ordenar es muy grande
- Counting sort no es un algoritmo **"in place"**. Necesita buffer de salida del tamaño de la entrada.

Section 5

BucketSort

BinSort (a.k.a Bucket Sort)

Bucket Sort reagrupa los elementos del vector en varios grupos (buckets), según una partición del rango min/max, para proceder a ordenar cada uno de ellos por separado por otro algoritmo, reagrupando finalmente el vector ordenado como la secuencia de los grupos individuales ordenados.

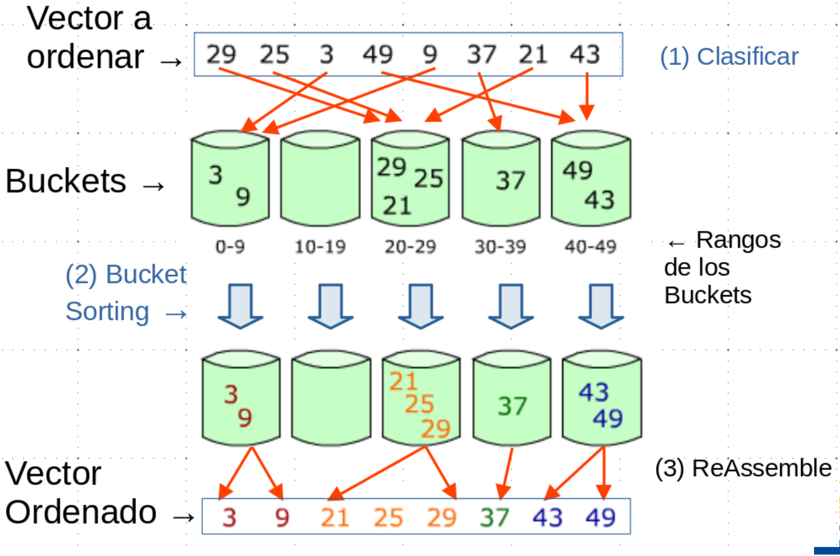
Tres fases:

1. Distribuir los elementos a ordenar en grupos cercanos (bins) segmentando el rango min/max del vector
2. Ordenar éstos grupos utilizando cualquier algoritmo (Insertion, merge, etc..)
3. Recomponer el vector en uno sólo

Hipótesis clave

La clave de ordenación es una magnitud real **distribuída de forma aproximadamente uniforme en el rango** $[min, \ldots max]$ **del vector**.
$\Rightarrow$ el número de elementos en cada grupo es similar, no concentrándose en uno o pocos de los grupos (worst case)

Bucket Sort: Cómo funciona ?



Bucket Sort: Algoritmo

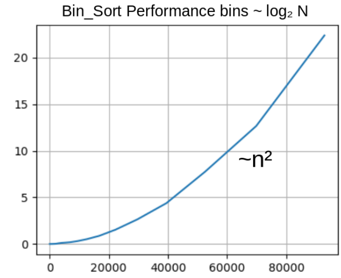
```
function BinSort(vec,nbins)
    ▷ Vec: vector a ordenar. nbins: número de buckets a usar ◁
    bins ← (List of lists, or list of linked lints) of size = nbins
    min,max ← min(vec), max(vec)                ▷ Compute min/max range
    ▷ (1) Distribute elements across bins ◁
    for element in vec do
        binNum ← getBinNum(min,max, element) ▷ Compute appropi-
        ate bucket for each element
        append element to bins[binNum]
    end for
    ▷ (2) Sort individually all the bins ◁
    for bin in bins do
        YourSortAlg(bin)        ▷ Use a sorting algorithm of your choice
    end for
    ▷ (3) Reassemble the vec from the sorted bins content ◁
    for bin in bins do
        place bin in vec                        ▷ at the right place !!
    end for
    return vec
end function
```

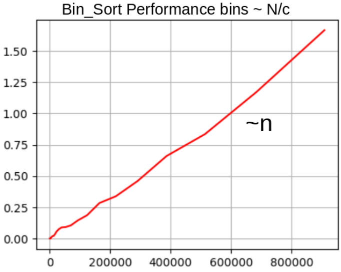
Bucket Sort: Complejidad. Elección of nbins

- BucketSort depende del número de bins utilizado k y del algoritmo de ordenación de los grupos
- Si usamos un método $\mathbf{O}(n^2)$ como inserción la complejidad del algoritmo resultante es del orden de

$$n + k \times (n/k)^2 \approx n + n^2/k$$

- dos opciones para el número de bins:
  1. k fijo, o $k \sim \log n \ll n$ implica $\mathbf{O}(n^2)$ o al menos $\mathbf{O}(n^c)$ con $c \in [1, 2]$
  2. $k \sim n/c$ lo que resulta en algortimo $\mathbf{O}(n)$ lineal





Bucket Sort: Pros and Cons

Pros de Bucket Sort

- Puede alcanzar performance lineal $\mathbf{O}(N)$ con un número alto de bin
- es un algoritmo estable[a]
- La ordenación de los buckets (o bins) es paralelizable
- es un algoritmo estable

[a] preserva el orden original a igualdad de claves

Cons de Bucket Sort

- Es necesario poder distribuir en buckets contiguos (clave de ordenamiento ha de real, más allá del operador comparador)
- Es óptimo sólo cuando la distribución de las claves es approx uniforme sobre el rango, dando lugar a buckets con número de elementos equilibrado

Section 6

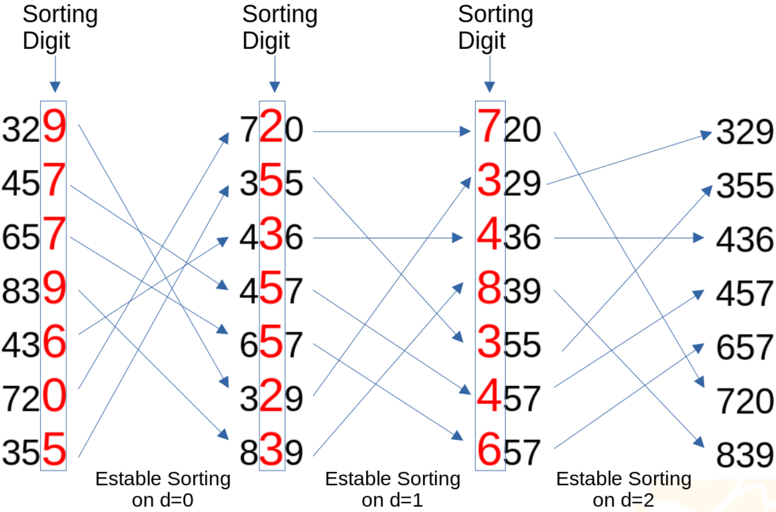
Radix Sort

Radix Sort

Creado en los años 30 para ordenar las tarjetas perforadas utilizadas en los ordenadores, ya sólo en museos (programado mecánicamente)

- Util para ordenar información por múltiples campos como Fechas (año/mes/día) Personas (apellidos) etc..
- Ordenar N items en donde la clave es
  - clave a utilizar = digito1, digito2, digito3, ..., digito d
  - **LSD** (Least Significant Digit) o **MSD** (Most Significant Digit)
- Típica ordenación por múltiples claves en Excel...
- Formas de hacerlo:
  - Utilizar un algoritmo en donde las comparaciones son jerárquicas
  - Ordernar la información d veces con un algoritmo estable, cada vez en un dígito diferente

Radix Sort: How it works !



Radix Sort: Algorithm!

Radix Sort llama de forma repetida a un algoritmo estable, por cada uno de los dígitos, o campos, significativos.
El uso de CountSort es frecuente para los ordenamientos estables

```
function RadixSort(vec)
    digits ← GetDigitsSeq(vec)
    for digit in digits do                          ▷ Must iterate from LSD to MSD
        EstableSort(vec,digit) ▷ Ordena, estable, basado en el dígito
        indicado
    end for
    return vec
end function
procedure EstableSort(vec, digit)
    ▷ Alg for Stable sorting based on given digit                     ◁
    return vec
end procedure
procedure GetDigitsSeq(vec, digit)
    ▷ Return ordered MSD to LSD digits based on data                  ◁
    ▷ ej. 0 . . . log10 max(vec) in case of whole data sequence        ◁
    ▷ ej. max length of word, for a sequence of words                  ◁
    return digits
end procedure
```

Radix Sort: Complejidad

Complejidad Temporal

- Radix Sort tiene una complejidad de $\mathbf{O}(d * (n + b))$, done d es el número de dígitos, n es el tamaño del vector de datos, y b es la base del sistema de representación numérico (10, normalmente, 28 si caracteres)[a]
- Radix sort es habitualmente más rápido que algortimos de comparación, como quickSort, or mergeSort, en sets de datos grandes. Es lineal en el número de dígitos y un gran rango de datos (min/max) puede afectarle negativamente.

[a] $d \sim \log_{10} max(vec)$

Complejidad Espacial

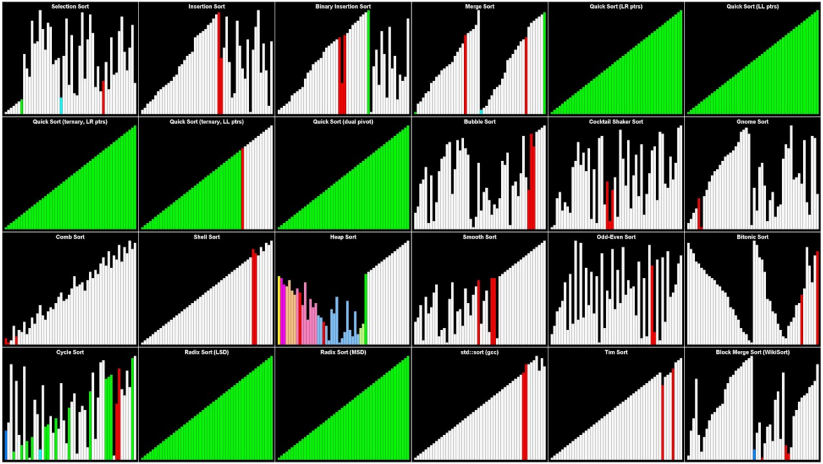
Radix sort tiene una complejidad espacial de $\mathbf{O}(n + b)$. Se debe al uso de algoritmos tipos CountSort donde se han de crear b buckets y duplicar el espacio del vector a ordenar

Performance Compartiva de los Algoritmos de Ordenación

Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Figure: Compararativa Peor/Medio de complejidad de los diferentes Algoritmos de Ordenación . (Source: Cormen et al.)

End: Ver youtube



`https://www.youtube.com/watch?v=BeoCbJPuvSE`

COMILLAS ICAI