

Algoritmos y Estructuras de Datos

Tema 5: Sorting (Ordenación)

Grado Imat. Escuela ICAI

January 2024



• Métodos directos simples

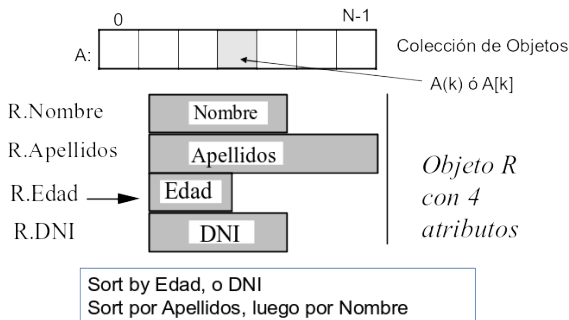
- Basados en inserción \Rightarrow Inserción directa, binaria
- Basados en selección \Rightarrow Selección directa
- Basados en intercambio \Rightarrow Intercambio directo (burbuja y sacudida)
- Basados en inserción \Rightarrow Algoritmo de Shell

• 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)

Ordenando Secuencias de Objetos o Estructuras

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



Python hint

- Añade el método `__gt__(object)` a la clase, que retorna True si self es mayor que object
- alternatively, 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

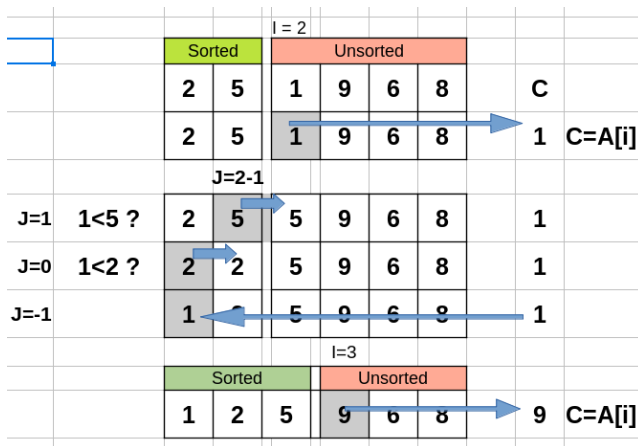


Sorting In-place Utilizando un vector de registros y ordenando sobre el mismo utilizando espacio auxiliar para un registro



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

```
1: function DIRECTINSERTIONSORT(aList)
2:   for i=1 to len(aList)-1 do
3:     carta  $\leftarrow$  aList[i]
4:     j  $\leftarrow$  i - 1
5:     while j  $\geq$  0 and carta < aList[j] do
6:       aList[j + 1]  $\leftarrow$  aList[j]
7:       j  $\leftarrow$  j - 1
8:     end while
9:     aList[j + 1]  $\leftarrow$  carta
10:  end for
11:  return
12: end function
```

▷ *selecciona siguiente carta*

▷ *busca punto de inserción*

▷ *comparando y ...*

▷ *shifting data*

▷ *Inserta carta en el punto*

Insertion Sort II: algoritmo

```
1: function DIRECTINSERTIONSORT(aList)
2:   for i=1 to len(aList)-1 do
3:     carta ← aList[i]
4:     j ← i - 1
5:     while j ≥ 0 and carta < aList[j] do
6:       aList[j + 1] ← aList[j]
7:       j ← j - 1
8:     end while
9:     aList[j + 1] ← carta
10:  end for
11:  return
12: end function
```

▷ selecciona siguiente carta

▷ busca punto de inserción

▷ comparando y ...

▷ shifting data

▷ Inserta carta en el punto

Complejidad ??

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

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.

```
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
```

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 $O(N)$
- búsqueda binaria del punto de inserción Líneas 5 a 12 es $O(\log N)$
- Data shift en líneas 13 a 15 es $O(N)$

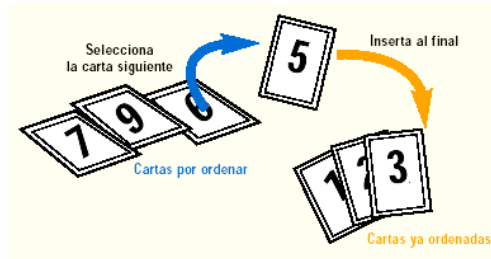
- **Total**

$$= O(N) \times (O(\log N) + O(N))$$
$$= O(N^2)$$

```
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

```
1: function SELECTIONSORT(aList)
2:   for i=0 to N-2 do
3:      $k \leftarrow i$  ,  $x \leftarrow A[i]$ 
4:     for j=i+1 to N-1 do
5:       if ( $A[j] < x$ ) then
6:          $k \leftarrow j$  ;  $x \leftarrow A[j]$ 
7:       end if
8:     end for
9:      $A[k] \leftarrow A[i]$  ;  $A[i] \leftarrow x$ 
10:  end for
11:  return
12: end function
```

▷ separación secuencia origen/destino
▷ inicializa búsqueda mínimo en origen

▷ busca mínimo en origen

▷ Intercambio

Selection Sort II: algoritmo

```
1: function SELECTIONSORT(aList)
2:   for i=0 to N-2 do
3:     k ← i , x ← A[i]
4:     for j=i+1 to N-1 do
5:       if (A[j] < x then
6:         k ← j ; x ← A[j]
7:       end if
8:     end for
9:     A[k] ← A[i] ; A[i] ← x
10:  end for
11:  return
12: end function
```

▷ separación secuencia origen/destino
▷ inicializa búsqueda mínimo en origen

▷ busca mínimo en origen

▷ Intercambio

Complejidad ??

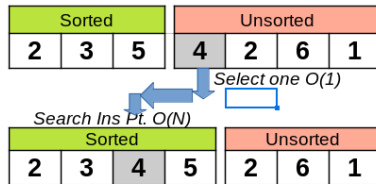
- **For** en línea 2 es de $O(N)$
 - Búsqueda secuencial **For** en líneas 3 a 8 es de $O(N)$
 - Intercambio (línea 9) es $O(1)$
- ⇒ Selection Sort es de $O(N) \times (O(N) + O(1)) = O(N^2)$

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 $O(N)$)

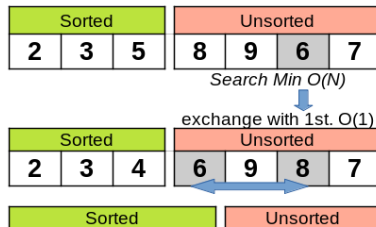
Insertion Sort

- 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

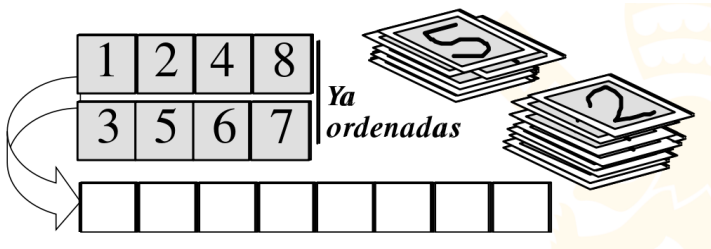


Section 2

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
```

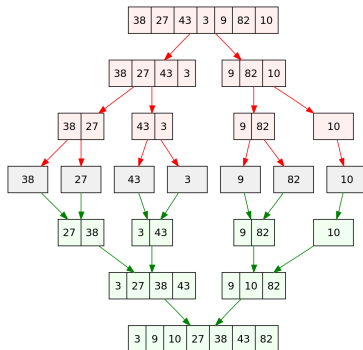
MergeSort

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



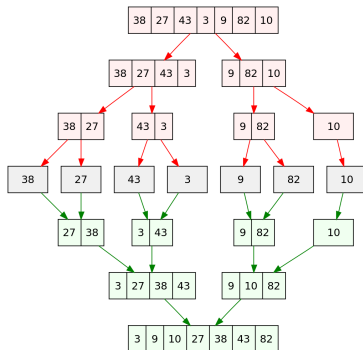
Complejidad Lineal-Logarítmica: Ejemplo MergeSort

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

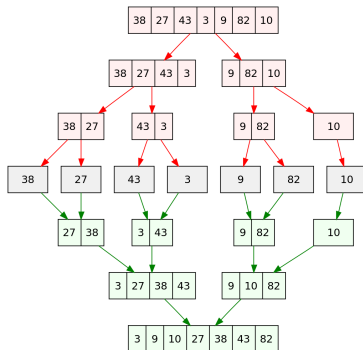
Divide la secuencia en dos, sobre los que invoca el Merge_sort de forma recursiva, para luego realizar un merge de la secuencias previamente ordenadas

Cuál es la complejidad ?

- Fase de división

$$\rightarrow \log_2(N) \times O(1)$$

- Cuántos niveles hay en el árbol durante el merge ?



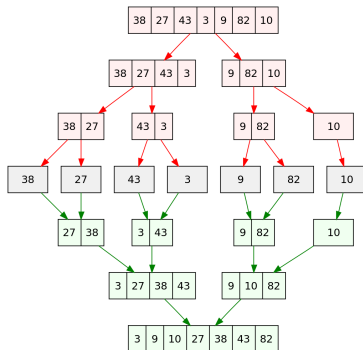
Complejidad Lineal-Logarítmica: Ejemplo MergeSort

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
→ $\log_2(N) \times O(1)$
- Cuántos niveles hay en el árbol durante el merge ?
→ $\log_2(N)$
- Cuál es el coste de cada uno de ellos



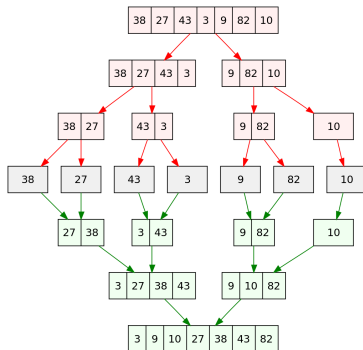
Complejidad Lineal-Logarítmica: Ejemplo MergeSort

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
→ $\log_2(N) \times O(1)$
- Cuántos niveles hay en el árbol durante el merge ?
→ $\log_2(N)$
- Cuál es el coste de cada uno de ellos
→ $O(N)$
- 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

→ $\log_2(N) \times O(1)$

- Cuántos niveles hay en el árbol durante el merge ?

→ $\log_2(N)$

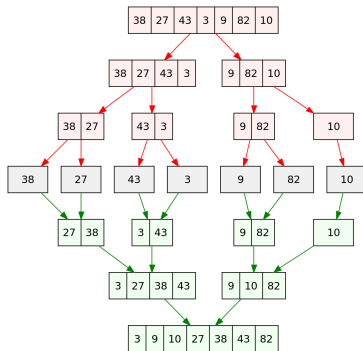
- Cuál es el coste de cada uno de ellos

→ $O(N)$

- Cuál es la complejidad resultante ?

→ $\log_2(N) * (O(1) + O(N))$

→ $O(N * \log N)$



Section 3

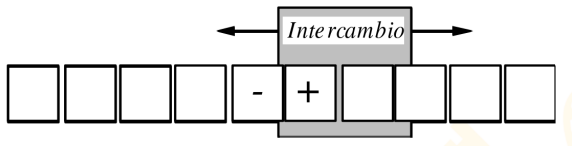
Algoritmos de Intercambio

- Bubble sort (Intercambiando elementos adyacentes)
- 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.

```
1: function BUBBLESORT1(vec)
2:    $n \leftarrow \text{len}(\text{vec})$ 
3:   repeat
4:     swap  $\leftarrow \text{False}$ 
5:     for i=1 to n-1 do
6:       ▷ Perform Xchange as needed
7:       if  $\text{vec}[i-1] > \text{vec}[i]$  then
8:         exchange  $\text{vec}[i-1]$  and  $\text{vec}[i]$ 
9:         swap  $\leftarrow \text{True}$ 
10:      end if
11:    end for
12:  until swap is False
13:  return
14: end function
```

Bubble sort: Algoritmo

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

```
1: function BUBBLESORT1(vec)
2:   n ← len(vec)
3:   repeat
4:     swap ← False
5:     for i=1 to n-1 do
6:       ▷ Perform Xchange as needed
7:       if vec[i-1] > vec[i] then
8:         exchange vec[i-1] and vec[i]
9:         swap ← True
10:      end if
11:    end for
12:  until swap is False
13:  return
14: 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:

```
1: function BUBBLESORT2(vec)
2:    $n \leftarrow \text{len}(\text{vec})$ 
3:   for i=1 to n by 1 do
4:     for j=n-1 to i step -1 do
5:       if vec[j-1] > vec[j] then
6:         exchange vec[j-1] and vec[j]
7:       end if
8:     end for
9:   end for
10:  return
11: end function
```

▷ Separar Origen y Destino
▷ Backward scan unsorted region

▷ perform exchange

Bubble Sort: versión mejorada. Complejidad

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

```
1: function BUBBLESORT2(vec)
2:    $n \leftarrow \text{len}(\text{vec})$ 
3:   for i=1 to n by 1 do
4:     for j=n-1 to i step -1 do
5:       if  $\text{vec}[j-1] > \text{vec}[j]$  then
6:         exchange  $\text{vec}[j-1]$  and  $\text{vec}[j]$ 
7:       end if
8:     end for
9:   end for
10:  return
11: end function
```

▷ Separar Origen y Destino
▷ Backward scan unsorted region

▷ perform exchange

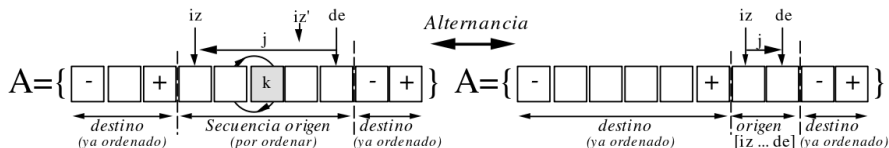
Complexity

En el peor de los casos, el Algoritmo es de $O(N^2)$ 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

```
1: function SHAKERSORT(vec)
2:    $de \leftarrow k \leftarrow \text{len}(\text{vec})$ 
3:    $iz \leftarrow 1$ 
4:   repeat
5:     for  $j = de$  to  $iz$  step -1 do
6:       if  $\text{vec}[j-1] > \text{vec}[j]$  then
7:         exchange  $\text{vec}[j-1]$  and  $\text{vec}[j]$ 
8:          $k \leftarrow j$ 
9:       end if
10:    end for
11:     $iz \leftarrow (k + 1)$ 
12:    for  $j = iz$  to  $de$  do
13:      if  $\text{vec}[j-1] > \text{vec}[j]$  then
14:        exchange  $\text{vec}[j-1]$  and  $\text{vec}[j]$ 
15:         $k \leftarrow j$ 
16:      end if
17:    end for
18:     $de \leftarrow (k - 1)$ 
19:  until  $iz > de$ 
20:  return
21: end function
```

▷ Backward scan

▷ perform exchange

▷ Mark exchange position

▷ Fija inicio del forward Scan

▷ Forward scan

▷ perform exchange

▷ Mark exchange position

▷ resets right boundary

Shaker Sort: Algoritmo

```
1: function SHAKERSORT(vec)
2:   de ← k ← len(vec)
3:   iz ← 1
4:   repeat
5:     for j = de to iz step -1 do
6:       if vec[j-1] > vec[j] then
7:         exchange vec[j-1] and vec[j]
8:         k ← j
9:       end if
10:    end for
11:    iz ← (k + 1)
12:    for j = iz to de do
13:      if vec[j-1] > vec[j] then
14:        exchange vec[j-1] and vec[j]
15:        k ← j
16:      end if
17:    end for
18:    de ← (k - 1)
19:  until iz > de
20:  return
21: end function
```

▷ Backward scan

▷ perform exchange

▷ Mark exchange position

▷ Fija inicio del forward Scan

▷ Forward scan

▷ perform exchange

▷ Mark exchange position

▷ resets right boundary

Complejidad

Las mejoras del algoritmo son parciales, → **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 Forw 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]

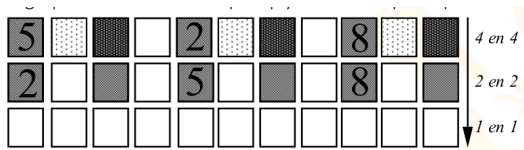
ForwScan, k=4 Range [iz=2,de=3]

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

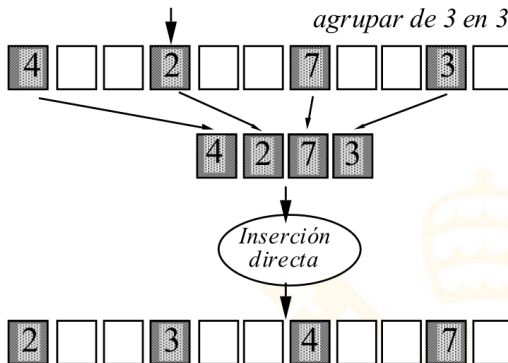
- Combina la inserción directa con el concepto de Divide y Vencerás
- Aplica la inserción directa ordenando subsets de datos con un **step** dado. Lo que aumenta la velocidad de los datos a través de la secuencia a velocidad de **step**.
- Se aplica en secuencia de *Steps* decrecientes, terminando necesariamente 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 $Step = 1$
- Ejemplo, para $n = 10,000$ una secuencia sería 25, 13, 7, 3, 1

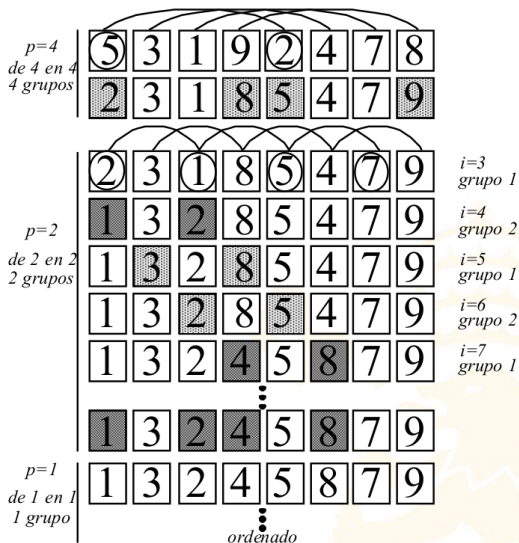
Shell Sort: Idea y Paralelizabilidad

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
5:     for i=step to n-1 by 1 do
6:       aux ← vec[i]
7:       j ← i - step
8:       while j >= 0 & (aux < vec[j]) do
9:         vec[j + step] ← vec[j]
10:        decrement j by step
11:      end while
12:      if i ≠ (j + step) then
13:        vec[j + step] ← aux
14:      end if
15:    end for
16:  end for
17:  return
18: end function
```

▷ iterate along decreasing steps
▷ Start at step, till the end

▷ search backward, stride = step

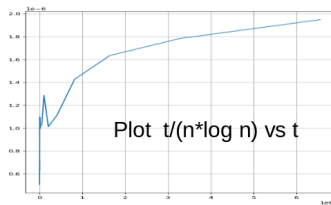
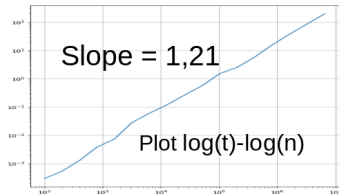
▷ shifting if required
▷ Stride = step, backwards

▷ Insertion at selected point



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

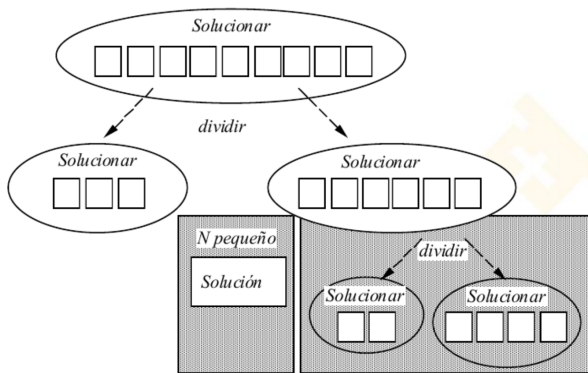
Counting Sort: Algoritmo basado en frecuencia para claves de universo finito

BinSort: algoritmo de agrupación para claves continuas o categorizables

RadixSort: Algoritmo multiclave con ordenación jerárquica

QuickSort

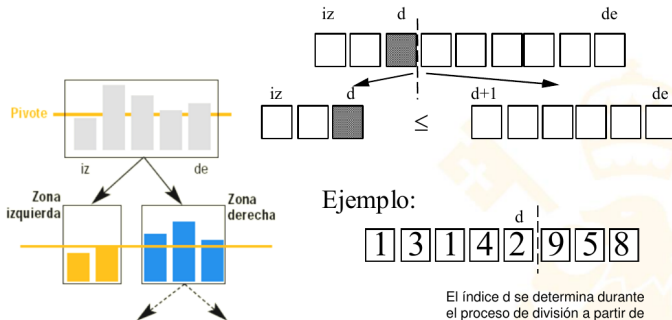
- QuickSort es un método de Ordenación recursivo que funciona por partición recursiva en segmentos cuyos elementos son siempre superiores o inferiores a cualquier elemento de otro segmento
- El proceso de partición es análogo al método de intercambio, pero aquí las distancias son tan grandes como posible.
- 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.



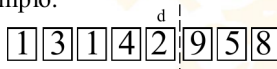
QuickSort: particionado

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

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



Ejemplo:



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

La clave está en la selección del umbral d .

QuickSort Algorithm

```
1: function QUICKSORT(vec,start,stop)
2:   if len(vec) ≤ 2 then
3:     Manually sort vec
4:     return vec
5:   else
6:      $d \leftarrow \text{QS\_DIVIDE}(\text{vec})$ 
7:     QUICKSORT(vec,start, start+d-1)
8:     QUICKSORT(vec,start+d, stop)
9:   end if
10: end function
```

▷ *perform quick Sort Divide*
▷ *Recurse on Left*
▷ *Recurse on right*

La profundidad del algoritmo depende del equilibrio del proceso de divide

qs_divide Ejemplo

El algoritmo de `qs_divide` compara los elementos del vector con el pivote, desde $l = 1$ (fw) y desde $r = n - 1$ (bw) haciendo los intercambios necesarios hasta barrer el vector completo.

		Vec[r] <= Th	
		True	False
Vec[l] > Th	True	Exch L + R -	L = R -
	False	L + R =	L + R -

El algoritmo de qs_divide compara los elementos del vector con el pivote, desde $l = 1$ (fw) y desde $r = n - 1$ (bw) haciendo los intercambios necesarios hasta barrer el vector completo.

		Vec[r] <= Th	
		True	False
Vec[l] > Th	True	Exch L + R -	L = R -
	False	L + R =	L + R -

Aplicación concreta

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

Index	0	1	2	3	4	5	6	7	
		$l = 1$						$r = 7$	
Vec[1] > umbral es	Falso	4	3	1	9	8	5	1	2
									Vec[7] <= umbral es True
									→ no swap
									$l = 2$
Vec[2] > umbral es	Falso	4	3	1	9	8	4	1	2
									Vec[7] <= umbral es True
									→ No swap
									$l = 3$
Vec[3] > umbral es	True	4	3	1	9	8	4	1	2
									Vec[7] > umbral es True
									→ Swap ←
									$l = 4$
Vec[4] > umbral es	True	4	3	1	2	8	4	1	9
									Vec[6] > umbral es True
									→ swap ←
									$l = 5$
Finish condition									$r = 5$
not(vec[l] > vec[0]) → d = l + 1		4	3	1	2	1	4	8	9
									left divide right
return d = 6								d = 6	

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



partition divide Algorithm

```
function QS_DIVIDE(vec,pivot)
  ▷ partitions vec, vec[pivot] as the threshold
  threshold ← vec[pivot]
  vec[0] ↔ vec[pivot]
  l ← 1
  r ← len(vec) - 1
  while l < r do
    L_r ← (vec[l] > threshold)
    R_r ← (vec[r] =< threshold)
    if L_r ∧ R_r then
      | vec[l] ↔ vec[r]
    end if
    increment l ; decrement r
    if L_r ∧ ¬R_r then
      | decrement l
    else if ¬L_r ∧ R_r then
      | increment r
    end if
  end while
  if vec[l] > threshold then
    | return l
  else
    | return l + 1
  end if
end function
```

▷ place pivot first

▷ work on both sides
▷ Right and left tests...

▷ swap elements

▷ in all cases advance pointers

▷ undo left ptr advance

▷ undo right ptr advance

▷ final adjustment

Este algoritmo es claramente $O(N)$

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

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 $O(n \log n)$] **Pero si lo que queremos es ordenarla !**
⇒ Afortunadamente, el algoritmo quickSelect^a es capaz de producir la mediana en $O(N)$

^ade 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
    | Return vec[0]
  else
    | d ← QS_DIVIDE(vec)
    | if offset + d > targetPos then
      | val ← FINDORDEREDVAL(vec[:d], offset, target)
    else
      | val ← FINDORDEREDVAL(vec[d:], offset+d, target)
    end if
    return val
  end if
end procedure
```

▷ Terminal Case

▷ applies `qs_divide` with random pivot

▷ Busca en izq

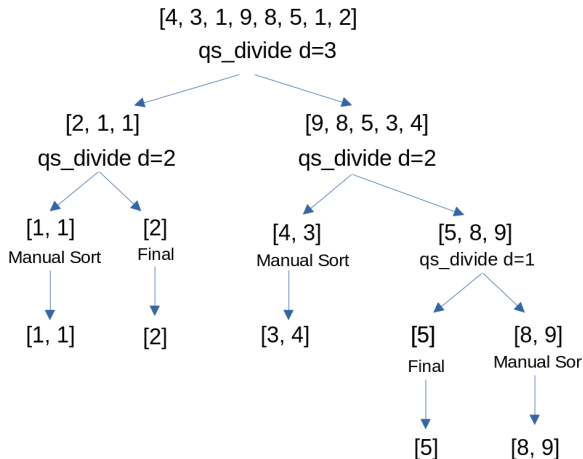
▷ Busca en derecha



COMILLAS
ICAI

qs: Example

Quick Sort aplica recursivamente el algoritmo `qs_divide` con pivots aleatorios u, opcionalmente cerca de la mediana, hasta longitudes finales de < 2 . El ejemplo utiliza el último elemento como pivote.



Complejidad Temporal

- **Caso Óptimo:** $\Omega(N \log(N))$. Este mejor caso para quicksort ocurre cuando el pivo es elegido tal que divide el vector de aproximadamente el mismo tamaño. Esto balancea la división y hace el algoritmo muy eficaz.
- **Caso típico** $\Theta(N \log(N))$ En término medio, la performance de Quicksort's es muy buena y lo hace uno de los algoritmos más eficaces
- **Caso Peor:** $O(N^2)$ El caso peor par Quicksort ocurre cuando la elección del pivote resulta en divisiones consiste y altamente desequilibradas.
- **Espacio Auxiliar** $O(1)$, Sin considerar el coste de recursión. Con él sería de $O(N)$.

QuickSort: Pros y cons

Ventajas del Quick Sort:

- Es un algoritmo divide-and-conquer que hace muy fácil la resolución de los problemas
- Es muy eficiente sobre conjuntos grandes de datos
- Tiene poca complejidad espacial, requiere poco espacio adicional para trabajar.

QuickSort: Pros y cons

Ventajas del Quick Sort:

- Es un algoritmo divide-and-conquer que hace muy fácil la resolución de los problemas
- Es muy eficiente sobre conjuntos grandes de datos
- Tiene poca complejidad espacial, requiere poco espacio adicional para trabajar.

Desventajas of Quick Sort

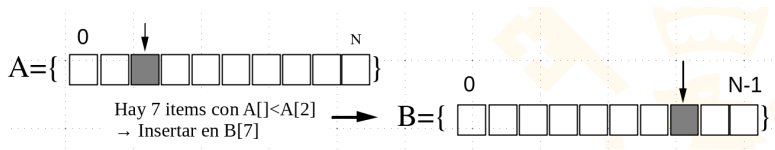
- En su Worst-Case la complejidad puede ser de $O(N^2)$ que puede ocurrir cuando el pivot es malamente elegido
- No es una buena opción para conjuntos pequeños de datos
- No es un algoritmo estable.

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
 $\Rightarrow O(N)$
2. Tomando los elementos del vector a ordenar, en orden inverso:
 - Utilizar el valor como clave en vector de frecuencias acumuladas
 - Utilizar la frecuencia acumulada como índice (restar 1) en el vector de destino
 - decrementar la posición usada en el vector de frecuencia

Counting Sort How it Works

1. allocar CountVec de tamaño $K+1$, contar frecuencias y acumular $\Rightarrow O(N)$
2. Tomando los elementos del vector a ordenar, en orden inverso:
 - Utilizar el valor como clave en vector de frecuencias acumuladas
 - Utilizar la frec acumulada 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

1: **function** COUNTSORT(vec)

Require: vec contiene claves positivas óptimamente de tamaño $\ll \text{len}(\text{vec})$

2: $k \leftarrow \max(\text{vec})$

3: ▷ *allocate counting buffer and output vector* ◁

4: countVec \leftarrow Array() of length $k+1$, initialized to 0s

5: sortecVec \leftarrow Array() of length = len(vec)

6: ▷ *Compute Keys frequencies* ▷ $O(N)$ ◁

7: **for** item in Vec **do**

8: | increment countVec[item]

9: **end for**

10: ▷ *place the items in proper order as per frequencies* ◁

11: **for** item in reversed(vec) **do** ▷ $O(N)$

12: | newPos \leftarrow (countVec[item]-1)

13: | sortecVec[newPos] \leftarrow item

14: | decrement countVec[item]

15: **end for**

16: **return** sortecVec

17: **end function**

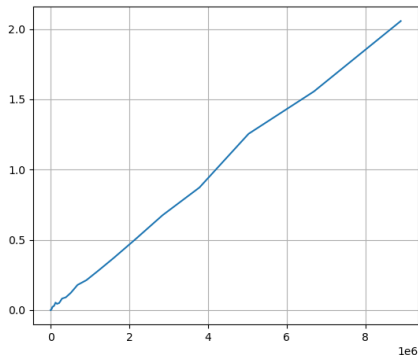


COMILLAS
ICAI

Counting Sort: Complejidad

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 $O(N + M)$.
- **Auxiliary Space:**
 $O(N + M)$ que son los tamaños de los buffers adicionales a alojar.



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



COMILLAS
ICAI

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 QuickSort, incluso 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)

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, incluso 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 $[0, \dots, n]$
- 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:

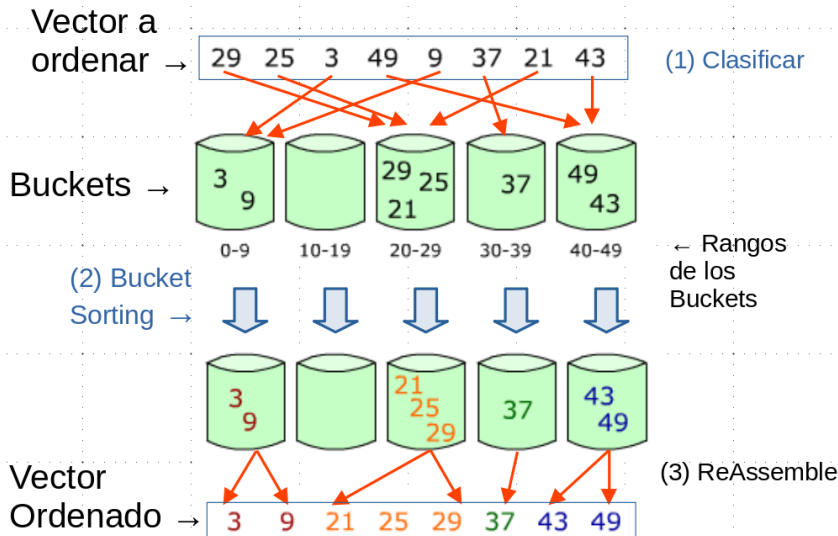
- 1 Distribuir los elementos a ordenar en grupos cercanos (bins) segmentando el rango min/max del vector
- 2 Ordenar éstos grupos utilizando cualquier algoritmo (Insertion, merge, etc..)
- 3 Reconponer el vector en uno sólo

Hipótesis clave

La clave de ordenación es una magnitud real **distribuída de forma aproximadamente uniforme en el rango $[min, \dots max]$ del vector.**

⇒ 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

```
1: function BINSORT(vec,nbins)
2:   ▷ Vec: vector a ordenar. nbins: número de buckets a usar
3:   bins ← (List of lists, or list of linked lists) of size = nbins
4:   min,max ← min(vec), max(vec)      ▷ Compute min/max range
5:   ▷ (1) Distribute elements across bins
6:   for element in vec do
7:     binNum ← GETBINNUM(min,max, element) ▷ Compute appropriate bucket for each element
8:     append element to bins[binNum]
9:   end for
10:  ▷ (2) Sort individually all the bins
11:  for bin in bins do
12:    | YOURSORTALG(bin)      ▷ Use a sorting algorithm of your choice
13:  end for
14:  ▷ (3) Reassemble the vec from the sorted bins content
15:  for bin in bins do
16:    | place bin in vec      ▷ at the right place !!
17:  end for
18:  return vec
19: end function
```

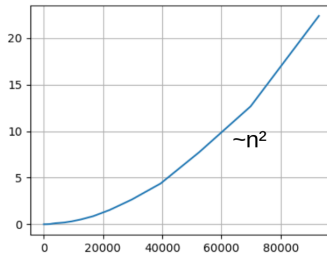
Bucket Sort: Complejidad. Elección of nbins

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

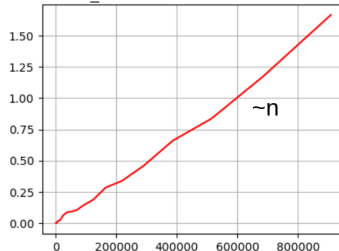
$$n + k \times (n/k)^2 \approx n + n^2/k$$

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

Bin_Sort Performance bins $\sim \log_2 N$



Bin_Sort Performance bins $\sim N/c$



Bucket Sort: Pros and Cons

Pros de Bucket Sort

- Puede alcanzar performance lineal $O(N)$ con un número alto de bin
- es un algoritmo estable^a
- La ordenación de los buckets (o bins) es paralelizable
- es un algoritmo estable

^apreserva el orden original a igualdad de claves

Bucket Sort: Pros and Cons

Pros de Bucket Sort

- Puede alcanzar performance lineal $O(N)$ con un número alto de bin
- es un algoritmo estable^a
- La ordenación de los buckets (o bins) es paralelizable
- es un algoritmo estable

^apreserva 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

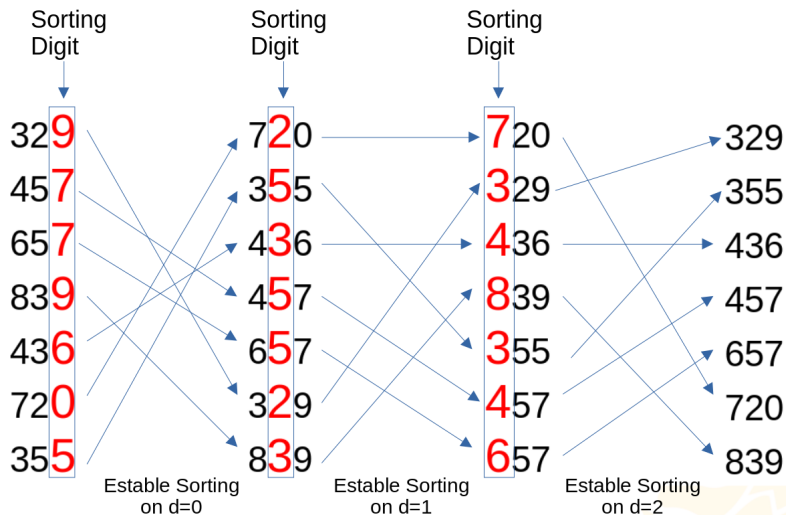
Section 6

Radix Sort

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
 - Ordenar 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

```
1: function RADIXSORT(vec)
2:   digits ← GETDIGITSSEQ(vec)
3:   for digit in digits do                                ▷ Must iterate from LSD to MSD
4:     ESTABLESORT(vec,digit) ▷ Ordena, estable, basado en el dígito
                           indicado
5:   end for
6:   return vec
7: end function
8: procedure ESTABLESORT(vec, digit)
9:   ▷ Alg for Stable sorting based on given digit                                <
10:  return vec
11: end procedure
12: procedure GETDIGITSSEQ(vec, digit)
13:   ▷ Return ordered MSD to LSD digits based on data                                <
14:   ▷ ej.  $0 \dots \log_{10} \max(\text{vec})$  in case of whole data sequence                <
15:   ▷ ej. max length of word, for a sequence of words                            <
16:   return digits
17: end procedure
```

Complejidad Temporal

- Radix Sort tiene una complejidad de $O(d * (n + b))$, donde 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 algoritmos 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.

$$^a d \sim \log_{10} \max(\text{vec})$$

Radix Sort: Complejidad

Complejidad Temporal

- Radix Sort tiene una complejidad de $\mathbf{O}(d * (n + b))$, donde 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 algoritmos 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.

$$^a d \sim \log_{10} \max(\text{vec})$$

Complejidad Espacial

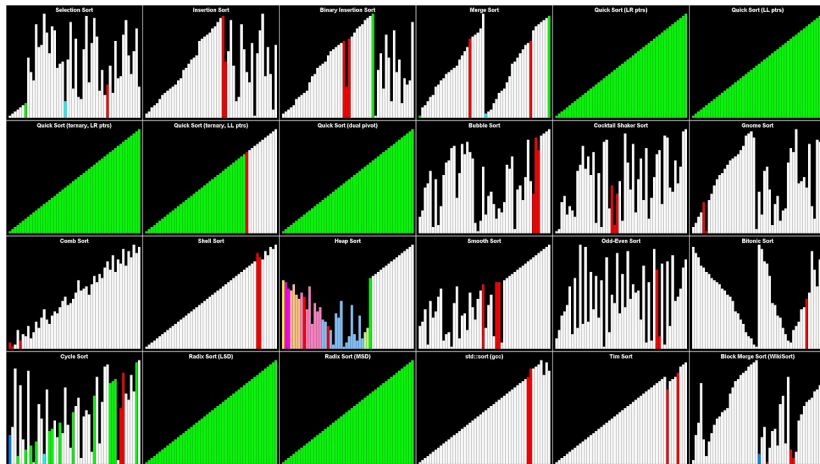
Radix sort tiene una complejidad espacial de $\mathbf{O}(n + b)$. Se debe al uso de algoritmos tipos CountSort donde se han de crear b buckets y duplicar el espacio del vector a ordenar

Performance Comparativa 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: Comparativa 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