Universidad de La Laguna

Tema 3. Ordenación

1. ALGORITMOS CUADRÁTICOS DE ORDENACIÓN
   Inserción, Selección, Intercambio
2. ALGORITMOS LOGARÍTMICOS DE ORDENACIÓN
   HeapSort, QuickSort, MergeSort
3. OTROS ALGORITMOS DE ORDENACIÓN
   Incrementos decrecientes, Radicales, TimSort

Ordenar

- **Ordenar:**

  reorganizar un conjunto de objetos en una secuencia especificada por una ***clave****.*
- **Objetivo** básico de la ordenación:

  facilitar la ***búsqueda*** *de un elemento dado por su* ***clave****.*
- La ordenación está presente en **cualquier** contexto;

  se ***aprende*** *antes a ordenar que a contar.*
- Los métodos se describen para ordenación **ascendente**;

  de menor a mayor clave.

Clasificación de los métodos

- Según el dispositivo de almacenamiento:
  - Los métodos de **ordenación externa:**
    - los datos se almacenan en memoria **secundaria**.
  - Los métodos de **ordenación interna:**
    - los datos se almacenan en memoria **principal**.
- Según la forma de abordar la tarea:
  - Método de **ordenación directa:**
    - el método trabaja directamente con los elementos.
  - Método de **ordenación por descomposición:**
    - el método trabaja con partes de la secuencia.
- Método **estable:** no altera el orden ***previo*** de los elementos de la secuencia que tienen ***igual*** valor de la clave.

Los mejores métodos de ordenación

- Calidad de los métodos de ordenación:
  - se mide por la **complejidad** algorítmica.
- Clasificación:
  - **Algoritmos Cuadráticos:** métodos **sencillos** que son $O(n^2)$ en tiempo de ejecución.
  - **Algoritmos Logarítimicos:** métodos **complejos** que son $O(n \log_2 n)$ en tiempo de ejecución.
  - **Otros algoritmos:** métodos mejorados que alcanzan tiempos de ejecución $O(n^p)$ con $p\downarrow 1$

(n es el número de elementos de la secuencia)

ORDENACIÓN CUADRÁTICA

1. **ALGORITMOS CUADRÁTICOS DE ORDENACIÓN**
   - ➢ **Método de Ordenación por Inserción**
   - Método de Ordenación por Selección
   - Método de Ordenación por Intercambio
2. ALGORITMOS LOGARÍTMICOS DE ORDENACIÓN

   HeapSort, QuickSort, MergeSort
3. OTROS ALGORITMOS DE ORDENACIÓN

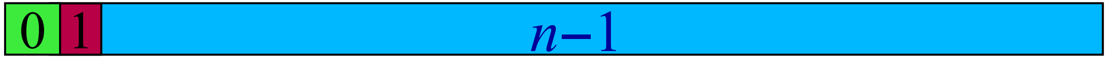
   Incrementos decrecientes, Radicales, TimSort

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

Ordenación por inserción

- Se considera la secuencia de elementos a ordenar **dividida** en dos partes:
  - los i primeros elementos que **ya** están ordenados
  - los $n-i$ últimos que **no** lo están.

- Se **comienza** desde i = 1.



- En cada iteración:
  - se toma el elemento que ocupa la **posición** i y
  - se **inserta** en la posición (j) adecuada entre los i primeros;

Ejemplo:

- **Desde la secuencia:**

```
0   1          j-1  j    j+1          i-1  i        n-1
5   7   . . .  12   42   44   55  . . . 94   18 ...06   67
```

- **Se obtiene la situación:**

```
0 1         j-1  j    j+1             i-1  i       n-1
5 7  . . .  12   18   42   44   55   . . . 94  ...06 67
```

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Ejecución, paso a paso

i=1	44	**55**	12	42	94	18	06	67	j=1
i=2	44	55	**12**	42	94	18	06	67	j=0
i=3	12	44	55	**42**	94	18	06	67	j=1
i=4	12	42	44	55	**94**	18	06	67	j=4
i=5	12	42	44	55	94	**18**	06	67	j=1
i=6	12	18	42	44	55	94	**06**	67	j=0
i=7	06	12	18	42	44	55	94	**67**	j=6
i=8	06	12	18	42	44	55	67	94	

Secuencia Ordenada

El código

- El código del método de ordenación por inserción puede ser sólo:

```
for (int i = 1; i < n; i++)
    insertar(sec,i,x);
```

- El procedimiento:

```
insertar(sec,i,x)
```

deberá insertar en la posición adecuada de la secuencia de tamaño **`i`** el nuevo elemento **`x`**.

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

La posición de inserción

- Para determinar la posición `j` de inserción, se **recorre** la parte ya ordenada de la secuencia comparando la clave del objeto a insertar con la clave de los elementos de la secuencia.

- De forma ascendente desde el principio de la secuencia:

```
j = 0 ;
while ( x > sec[j] )
    j++ ;
```

- De forma descendente desde el índice `i`:

```
j = i - 1 ;
while ( x < sec[j] )
    j-- ;
```

El centinela

- ***Hay que contemplar la posibilidad de llegar al final sin que se de la condición de parada, por lo que debe imponerse una doble condición:***

```
j = i - 1 ;
while ( (x < sec[j]) && (j > 0) )
    j-- ;
```

- La técnica del **centinela** evita la doble condición de parada:
  - *Se coloca como centinela una copia del elemento a insertar antes de la* ***primera*** *posición para que el recorrido se detenga al encontrarla.*

```
sec[-1] = x ;
j = i - 1 ;
while x < sec[j]
    j-- ;
```

La Inserción

- Una vez determinada la posición `j` de inserción, es necesario desplazar la parte de la secuencia entre los índices `i` y `j` para dejar *hueco* al objeto a insertar.

```
for (int k = i-1; k >= j; k--)
    sec[k+1] = sec[k] ;
sec[j] = x ;
```

- Las operaciones de búsqueda descendente de la posición de inserción y la propia inserción ***se pueden realizar a la vez***.

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Procedimiento conjunto

```
void inserta(Tvector &sec, int i, Tdef x){
  sec[-1] = x ;
  j = i - 1 ;
  while ( x < sec[j] ){
    sec[j+1] = sec[j] ;
    j-- ;
  }
  sec[j+1] = x ;
}
```

Inserción con centinela

```
for (int i = 1; i < n; i++ ){
    j = i ;
    x = sec[i] ;
    sec[-1] = x ;
    while ( x < sec[j-1] ){
        sec[j] = sec[j-1] ;
        j-- ;
    }
    sec[j] = x ;
}
```

Análisis de la complejidad

- Se trata de un algoritmo de complejidad $O(n^2)$.
    - *Hay que realizar n veces la búsqueda de la posición de inserción, que es O(n).*
- Una **mejora** significativa se obtendría al realizar una **búsqueda binaria** de la posición de inserción.
    - *Se compara la clave del elemento a insertar con la del elemento* ***medio*** *de la secuencia de búsqueda.*
    - *Según que la clave del elemento a insertar sea mayor o menor que la del elemento medio, la secuencia de búsqueda pasa a ser la mitad superior o la inferior de la anterior.*
- Al procedimiento resultante se le llama ***BinSort***

Procedimiento BinSort

```
for (int i = 1; i < n; i++ ) {
   j = i ;
   x = sec[i] ;
   ini = 0 ; fin = i-1;
   while ( ini <= fin ) {
      med = (ini+fin)/2;
      if (v[med] < x)
         ini = med+1;
      else
         fin = med-1;
   }
   for (int j = i-1; j >= ini; j--)
      sec[j+1] = sec[j] ;
   sec[ini] = x ;
}
```

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

Ejemplo:

- Partiendo de la situación intermedia del ejemplo anterior con el estado de la secuencia parcialmente ordenada siguiente.

  <u>12 42 44 55 94</u> 18 06 67

- La secuencia de los pasos que sigue el método es la siguiente:

Espacio de búsqueda	x	ini	fin	med
12 42 44 55 94	**18**	0	4	2
12 42	**18**	0	1	0
42	**18**	1	1	1
12 18 42 44 55 94	**06**	0	5	2
12 18	**06**	0	1	0
12	**06**	0	0	0
06 12 18 42 44 55 94	**67**	0	6	3
44 55 94	**67**	4	6	5
94	**67**	6	6	6

Complejidad de BinSort

- Si la ***búsqueda* binaria** en la secuencia de tamaño n emplea un tiempo $\boldsymbol{O(\log n)}$ y se realiza una búsqueda binaria de la posición de inserción en la parte de la secuencia ordenada, el algoritmo BinSort es $\boldsymbol{O(n \log n)}$**.**
- Sin embargo, un análisis *cuidadoso* de la implementación anterior muestra que este algoritmo es $\boldsymbol{O(n^2)}$.
  - Observando el procedimiento de la ordenación **BinSort** anterior se deduce que, aunque la **búsqueda binaria** es $\boldsymbol{O(\log n)}$, se aplica un procedimiento $\boldsymbol{O(n)}$ para **dejar hueco** al elemento a insertar.

El análisis detallado

- En particular, un análisis detallado del bucle `while` interno muestra que el número de divisiones (y comparaciones) máximo en la iteración i es $\log i = O(\log n)$.
- A continuación se ejecuta un bucle `for` en el que el número de asignaciones máximo en la iteración i es $i = O(n)$.

- Por tanto el número máximo total de divisiones y comparaciones es:

$$n\,O(\log n) = O(n \log n),$$

pero el número máximo total de asignaciones realizadas es: $O(n^2)$.

Implementación óptima de BinSort

- Con la secuencia representada por una lista enlazada, como la descrita anteriormente, se evita el bucle `for` y el número de asignaciones también sería $O(n \log n)$.
- Sin embargo, la búsqueda binaria tal como se ha expuesto aquí, está implementada para una secuencia representada por `array` lo que permite acceder en tiempo $O(1)$ al elemento medio, lo que no puede realizarse con la lista.
- Para obtener una implementación del algoritmo de ordenación por inserción de complejidad $O(n \log n)$ debe usarse un ***árbol binario de búsqueda*** para mantener la parte ordenada de la secuencia.

ORDENACIÓN CUADRÁTICA

1. ALGORITMOS CUADRÁTICOS DE ORDENACIÓN

- √ Método de Ordenación por Inserción
- ➢ **Método de Ordenación por Selección**
- ● Método de Ordenación por Intercambio

2. ALGORITMOS LOGARÍTMICOS DE ORDENACIÓN

HeapSort, QuickSort, MergeSort

3. OTROS ALGORITMOS DE ORDENACIÓN

Incrementos decrecientes, Radicales, TimSort

Ordenación por selección

- En el método de **ordenación por selección**,
  (*igual que en el método de ordenación por inserción*)
  se considera que, en una situación intermedia
  - los ***i*** primeros elementos **ya** están ordenados y
  - los ***n–i*** últimos **no** están ordenados.

- Sin embargo, en la iteración ***i***,
  - se **selecciona** la posición ***j*** del elemento de **menor** clave,
    (entre los elementos que están de la posición ***i*** a la posición ***n–1***)
  - se intercambian los elementos que están en las posiciones ***j*** e ***i***.

Ordenación por selección

- En la situación de partida:
  - no se asume la ordenación de ningún trozo; ***i* = 0**.

  - En la primera iteración se selecciona el <u>menor</u> de <u>toda</u> la secuencia

  - Se selecciona siempre el elemento de **menor** clave de los no ordenados;

  -
  - por tanto, en la iteración ***i***, los primeros ***i*** elementos, están ordenados,

.... pero además son los *i* elementos de menor clave.

Ejemplo numérico:

- *La ejecución de un paso:*
  - De la situación:

```
0   1       i-1   i                  j             n-1
 6 12 ... 18   44   55 ... 94 42 67 ... 84 87
```

  - se pasa a la situación:

```
0   1       i-1   i                  j             n-1
 6 12 ... 18   42   55 ... 94 44 67 ... 84 87
```

Ejecución, paso a paso

i=0	44	55	12	42	94	18	**06**	67	j=6
i=1	06	55	**12**	42	94	18	**44**	67	j=2
i=2	06	12	**55**	42	94	**18**	44	67	j=5
i=3	06	12	18	**42**	94	**55**	44	67	j=3
i=4	06	12	18	42	94	55	**44**	67	j=6
i=5	06	12	18	42	44	**55**	**94**	67	j=5
i=6	06	12	18	42	44	55	94	**67**	j=7
i=7	06	12	18	42	44	55	67	**94**	

Secuencia Ordenada

El código

- El código es similar al de ordenación por inserción, ***reemplazando*** el procedimiento de inserción por uno de selección:

```
for (int i = 0; i < n-1; i++)
    selecciona(sec,i);
```

- El procedimiento de selección elige el elemento de **menor** clave de la parte no ordenada de la secuencia (desde la posición *i* hasta el final) y los **intercambia**.

```
for (int i = 0; i < n-1; i++){
    min = i ;
    for (int j = i+1; j < n; j++)
        if sec[j] < sec[min]
            min = j ;
    x = sec[min] ;
    sec[min] = sec[i] ;
    sec[i] = x ;
}
```

Análisis del algoritmo

- Tiene dos bucles **`for`** anidados: se trata de un algoritmo **O(*n*²)**.
- Paralelamente al método de ordenación por inserción, se **toma** un elemento de la parte **no ordenada** y se añade a la parte **ordenada**.
- Sin embargo, en el método de ordenación por inserción, la selección del elemento a incorporar es **trivial** mientras que el proceso **inteligente** es el de búsqueda de la posición de ***<u>inserción</u>***.
- En el método de selección, la posición de inserción en la parte ordenada se obtiene **directamente** y el **esfuerzo** computacional se hace en el proceso de ***<u>selección</u>*** del elemento a insertar.

ORDENACIÓN CUADRÁTICA

1. **ALGORITMOS CUADRÁTICOS DE ORDENACIÓN**
   - √ Método de Ordenación por Inserción
   - √ Método de Ordenación por Selección
   - ➢ **Método de Ordenación por Intercambio**

2. ALGORITMOS LOGARÍTMICOS DE ORDENACIÓN
   HeapSort, QuickSort, MergeSort

3. OTROS ALGORITMOS DE ORDENACIÓN
   Incrementos decrecientes, Radicales, TimSort

Ordenación por Intercambio

- **Método de la Burbuja**

  BubbleSort

- **Método de la Sacudida**

  ShakeSort

Ordenación por intercambio

- En el método de **ordenación por intercambio** se recorre sucesivamente la secuencia **intercambiando** pares de elementos consecutivos desordenados.
- Se van comparando los pares consecutivos de elementos *desde el final hacia el principio*.
- Al proceso desde la comparación de los dos últimos elementos hasta los dos primeros se le llama **pasada**.
- La secuencia se suele representar verticalmente y se denomina método de la **burbuja** porque parece que hay un elemento que sube como una burbuja.

Ejemplo

- En la primera pasada, que aquí mostramos con la secuencia, **orientada en vertical**, el elemento burbuja es el **`06`**:

44	44	44	44	44	44	**44**	**06**
55	55	55	55	55	**55**	**06**	44
12	12	12	12	**12**	**06**	55	55
42	42	42	**42**	**06**	12	12	12
94	94	**94**	**06**	42	42	42	42
18	**18**	**06**	94	94	94	94	94
06	**06**	18	18	18	18	18	18
67	67	67	67	67	67	67	67

Criterios de Parada

- Si el elemento ***burbuja*** topa con uno de menor clave, éste elemento pasa a ser el nuevo elemento **burbuja** (que puede continuar subiendo) y su posición queda ocupada por el anterior.
- En cada pasada del método de la burbuja queda **colocado** el último elemento burbuja.
- El método de la burbuja se **termina** cuando en una pasada no se modifica la secuencia.

Las pasadas del ejemplo

- En la segunda pasada del ejemplo, el elemento burbuja es inicialmente el elemento con clave **18** pero acaba siendo el elemento de clave **12**.

- En la tercera pasada el elemento burbuja es el de clave **42**
- Las siguientes pasadas (en **horizontal**) son:

06	44	55	**12**	42	94	***18***	67	
06	**12**	44	55	**18**	42	94	***67***	
06	12	**18**	44	55	**42**	67	94	
06	12	18	**42**	**44**	55	67	94	
06	12	18	42	**44**	**55**	67	94	*no cambia*
06	12	18	42	44	**55**	**67**	94	
06	12	18	42	44	55	**67**	**94**	
06	12	18	42	44	55	67	**94**	

El código

```
for (int i = 1; i < n; i++){
    for (int j = n-1; j >= i; j--)
        if (sec[j] < sec[j-1]){
            swap(sec[j-1],sec[j]) ;
        }
}
```

¿Podríamos ahorrar comprobaciones si en una pasada completa no se produce ningún intercambio de elementos?

Elementos pesados

- Si al aplicar el método de la burbuja,
  sólo está mal colocado un elemento **ligero**,
  el método acaba rápidamente, aunque esté muy profundo.

- Sin embargo,
  si sólo está mal colocado un elemento **pesado,**
  éste se hunde muy lentamente.

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

Burbujas y Piedras

- El único elemento ligero mal colocado (el 02) sube en una pasada.

06	44	55	12	42	94	18	67	**02**
02	06	44	55	12	42	94	18	67

- El elemento pesado (el 97) necesita 8 pasadas para colocarse:

97	06	44	55	12	42	94	18	67
06	**97**	12	44	55	18	42	94	67
06	12	**97**	18	44	55	42	67	94
06	12	18	**97**	42	44	55	67	94
06	12	18	42	**97**	44	55	67	94
06	12	18	42	44	**97**	55	67	94
06	12	18	42	44	55	**97**	67	94
06	12	18	42	44	55	67	**97**	94
06	12	18	42	44	55	67	94	**97**

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

El método de La Sacudida

- Por tanto, este caso se resolvería fácilmente realizando el recorrido en sentido descendente; como una ***piedra*** que se hunde en el agua.
- El método de la **sacudida** evita este fenómeno haciendo recorridos ascendente y descendentes, **alternativamente**.
- Los recorridos pueden **empezar** debajo del **último** elemento burbuja y encima del **último** elemento hundido, acortándose el recorrido.
- A pesar de estas dos mejoras introducidas, tanto el algoritmo de la burbuja como el de la sacudida son $\mathbf{O}(n^2)$.

Ejemplo de la Sacudida

- La sacudida aplicado al ejemplo anterior acaba en 4 pasadas.

	44	55	12	42	94	18	**06**	67	
→	**06**	44	55	12	42	**94**	18	67	→
←	06	44	**12**	42	55	18	67	**94**	←
→	06	**12**	44	18	42	55	67	94	→
←	06	12	18	42	**44**	**55**	**67**	94	←
	06	12	18	42	44	55	67	94	

El código del método de la sacudida

```
ini = 1 ;
fin = n-1 ;
cam = n ;
while (ini < fin){
   for (int j = fin; j >= ini; j--)
       if (sec[j] < sec[j-1]) {
          swap(sec[j-1],sec[j]) ;
          cam = j ;
       }
   ini = cam + 1 ;
   for (int j = ini; j <= fin; j++)
       if (sec[j] < sec[j-1]) {
          swap(sec[j-1],sec[j]) ;
          cam = j;
       }
   fin = cam – 1 ;
}
```

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

ORDENACIÓN LOGARÍTMICA

1. ALGORITMOS CUADRÁTICOS DE ORDENACIÓN
   - √ Inserción, Selección, por Intercambio
2. **ALGORITMOS LOGARÍTMICOS DE ORDENACIÓN**
   - ➢ **Algoritmo de ordenación HeapSort**
   - Algoritmo de ordenación QuickSort
   - Algoritmo de ordenación MergeSort

4. OTROS ALGORITMOS DE ORDENACIÓN

   Incrementos decrecientes, Radicales, TimSort

Algoritmo HeapSort

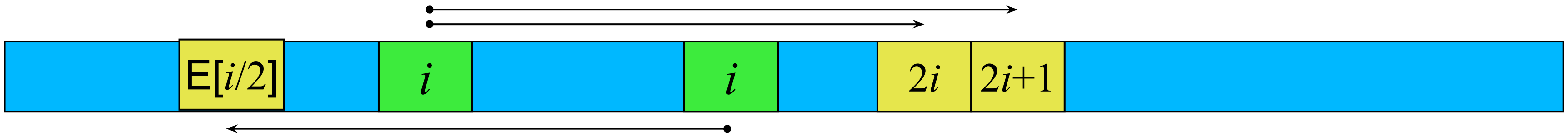
- Ordenación por selección: ***SelSort***
- Estructura de datos muy eficiente: ***Heap* o montón**
- Ordenación por selección con Heap: ***HeapSort***

Mejora del SelSort

- **En el algoritmo de ordenación por selección:**
  - En la iteración ***i*** **= 0, 1, …,** ***n*****–2**: se **selecciona** la posición ***j*** del elemento con **menor** clave, entre los elementos que están de la posición ***i*****+1** a la ***n*****–1**, *y* se intercambian los elementos que están en las posiciones ***j*** e ***i*****+1**.
  - Es de esperar que, si se utiliza un método **inteligente** de selección del menor elemento de la parte no ordenada que resulte más eficiente se podría **mejorar** la complejidad del método de ordenación.
  - Esta **mejora** significativa se consigue si la secuencia de elementos no ordenados se introducen previamente en un **montón**, **montículo** o **Heap** del que posteriormente se van extrayendo ordenadamente.
- Esto da lugar al denominado algoritmo **HeapSort** que es **O(*n* log *n*).**

Concepto de motón o Heap

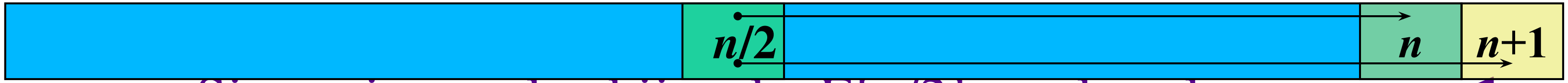
- El Heap se puede implementar de forma más natural con un árbol.
- La formalización de ***Williams*** se implementa en un array que <u>empieza en **1** en lugar de en **0**</u> para simplificar las expresiones.
- Un **Heap** es una estructura de datos basada en la relación **padre/hijo**:
    - Los **hijos** del elemento ***i*** son los elementos **2*i*** y **2*i*+1** (si están presentes en el montón).
    - El **padre** del elemento ***i*** es el elemento **E[*i*/2] = *i*/2**, si no es nulo (**E[.]** es parte entera)



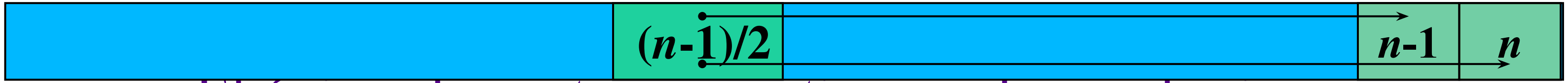
Límites del Heap

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

- Los elementos posteriores a **E[*n*/2]** no tienen hijos:
  - Si *n* es par, el elemento **E[*n*/2]** tiene un sólo hijo; el elemento ***n***.



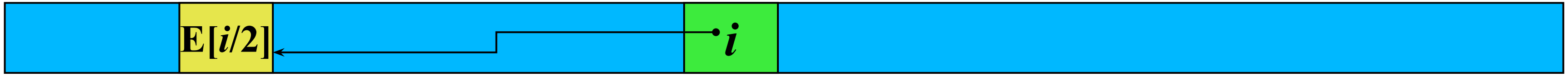
  - Si *n* es impar, los hijos de E[*n*/2] son los elementos ***n*−1** y ***n***.



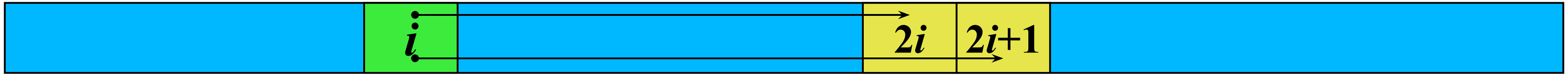
  - El único elemento que no tiene padre es el primero

Heap ordenado

- La noción de ordenación en la secuencia organizada cómo un montón o heap **es menos exigente** que la noción de ordenación estándar.
  - Decimos que el montón está ***ordenado*** si:
    ningún elemento tiene menor clave que su padre.



  - **Equivalentemente**, el montón está ordenado si:
    ningún elemento tiene mayor clave que ninguno de sus dos hijos.

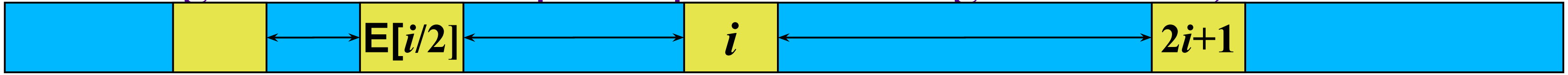


- Una vez que se tiene una parte de la secuencia ya ordenada según un montón, se **inserta** un nuevo elemento o se **elimina** un elemento de la parte ordenada del montón, ***manteniendo la ordenación***.

Insertar un elemento

- Vamos a tener los elementos del montón ordenados hasta una posición ***n***
  - Un nuevo elemento se **inserta** en la posición ***n*+1**, y si hace falta se sube para mantenerlo ordenado.

  - Si el elemento queda mal colocado se **sube**, intercambiándolo recursivamente con su padre, mientras haga falta (es decir, mientras tenga menor clave que su padre o se llegue a la raíz).



- La inserción implica **actualizar** el tamaño ***n*** del montón; ***n* = *n* + 1**

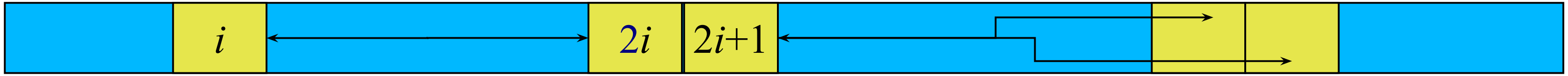
Eliminar un elemento

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

- Para **eliminar** un elemento de un montón ordenado:
  - se intercambia con el último elemento; y

si éste queda mal colocado se sube o se baja, según haga falta (hasta que no tenga hijos o de tenerlos, ninguno tenga menor clave).

- Para **bajar** un elemento mal colocado se intercambia recursivamente con el hijo de menor clave, mientras haga falta.



- La eliminación implica **actualizar** el tamaño n del montón. $n = n - 1$

Ordenar un Heap

- En un montón ordenado el elemento **mínimo** (el de menor clave) está siempre en la primera posición.

- Para **ordenar** los elementos de una secuencia por el método de ordenación por selección con el uso de un ***heap*** o montón, en una ***<u>primera fase</u>***, se ordena la secuencia como un montón incorporando al montón los elementos uno a uno, que se **sube** mientras haga falta.

- Luego, en la ***<u>segunda fase</u>***, se va seleccionando iterativamente el elemento de menor clave, que será siempre el que está en la raíz o primera posición.

- La raíz se reemplaza por el último que, mientras haga falta, se baja.

Ejemplo

- En este ejemplo se muestra, en primer lugar, cómo se realiza la primera **fase de introducción** de los elementos en el heap.
- Se denota por ***m*** al elemento que se incorpora al montón y por ***p*** al padre del elemento ***i***.
- En la **fase de extracción**, para sacar los elementos del montón de la secuencia ordenada se va extrayendo el primer elemento de forma iterativa, intercambiándose con el último y, si hace falta, se baja.
- Se rebaja el tamaño ***n*** del montón y se denota por ***h*** el hijo del elemento ***i*** de menor clave.

Introducciones

	1	2	3	4	5	6	7	8	
m=1	**44**	55	12	42	94	18	06	67	i=1 p=0 => colocado
m=2	44	**55**	12	42	94	18	06	67	i=2 p=1 55≥44 => colocado
m=3	44	55	**12**	42	94	18	06	67	i=3 p=1 12<44 => sube
	12	55	44	42	94	18	06	67	i=1 p=0 => colocado
m=4	12	55	44	**42**	94	18	06	67	i=4 p=2 42<55 => sube
	12	**42**	44	55	94	18	06	67	i=2 p=1 42≥12 => colocado
m=5	12	42	44	55	**94**	18	06	67	i=5 p=2 94≥42 => colocado
m=6	12	42	44	55	94	**18**	06	67	i=6 p=3 18<44 => sube
	12	42	**18**	55	94	44	06	67	i=3 p=1 18≥12 => colocado
m=7	12	42	18	55	94	44	**06**	67	i=7 p=3 06<18 => sube
	12	42	**06**	55	94	44	18	67	i=3 p=1 06<12 => sube
	06	42	12	55	94	44	18	67	i=1 p=0 => colocado
m=8	06	42	12	55	94	44	18	**67**	i=8 p=4 67≥55 => colocado

Extracciones (I)

	1	2	3	4	5	6	7	8	
n=8	**06**	42	12	55	94	44	18	67	
n=7	**67**	42	12	55	94	44	18	06	i=1 h=2,3 67>12 => baja
	12	42	**67**	55	94	44	18	06	i=3 h=6,7 67>18 => baja
	12	42	18	55	94	44	67	06	i=7 h=∅ (14>n=7) colocado
	12	42	18	55	94	44	67	06	
n=6	**67**	42	18	55	94	44	12	06	i=1 h=2,3 67>18 => baja
	18	42	**67**	55	94	44	12	06	i=3 h=6 67>44 => baja
	18	42	44	55	94	**67**	12	06	i=6 h=∅ (12>n=6) colocado
	18	42	44	55	94	67	12	06	
n=5	**67**	42	44	55	94	18	12	06	i=1 h=2,3 67>42 => baja
	42	**67**	44	55	94	18	12	06	i=2 h=4,5 67>55 => baja
	42	55	44	67	94	18	12	06	i=4 h=∅ (8>n=5) colocado
	42	55	44	67	94	18	12	06	

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Extracciones (II)

	1	2	3	4	5	6	7	8		
	42	55	44	67	94	18	12	06		
n=4	**94**	55	44	67	42	18	12	06	i=1	h=2,3 94 >44 => baja
	44	55	**94**	67	42	18	12	06	i=3	h=∅ (6>n=4) colocado
	44	55	94	67	42	18	12	06		
n=3	**67**	55	94	44	42	18	12	06	i=1	h=2,3 67>55 => baja
	55	67	**94**	44	42	18	12	06	i=3	h=∅ (6>n=3) colocado
	55	67	94	44	42	18	12	06		
n=2	**94**	67	55	44	42	18	12	06	i=1	h=2 94>67 => baja
	67	**94**	55	44	42	18	12	06	i=2	h=∅ (4>n=2) colocado
	67	94	55	44	42	18	12	06		
n=1	**94**	67	55	44	42	18	12	06	i=1	h=∅ (2>n=1) colocado

Inconvenientes

- **Inconvenientes** que podrían solventarse:
  - En primer lugar, la secuencia queda al final ordenada en sentido contrario y hay que invertirla.
    - Esto se soluciona considerando el orden dentro del montón o heap en sentido inverso; de ***mayor a menor***.
  - En segundo lugar, la parte que se considera que inicialmente ya está ordenada es sólo la raíz del montón porque no tiene padre.
    - Se puede también empezar suponiendo que los elementos de la mitad ***final de la secuencia*** son los que está inicialmente bien ordenados, porque no tienen hijos.

Método de Floyd (*HeapSort*)

- Se parte de que los elementos inicialmente bien colocados son los de **la mitad final** que son padres de elementos de fuera de la secuencia.
- El montón se completa incorporando cada vez un elemento a la derecha que es recolocado bajándolo si hace falta. De esta forma, tanto al incorporar como al excluir elementos del montón ***nunca es necesario subir un elemento, sólo bajarlo***.
- Finalmente, en la segunda fase del método, es siempre un elemento del final del heap (sin hijos) el que se coloca en la primera posición para ser bajado ***una y otra vez***. Para solventar esta cuestión no se ha realizado una propuesta satisfactoria.

Heap ordenado para *HeapSort*

- El *Heap* o montón está ***ordenado*** si:
  ningún elemento tiene mayor clave que su padre.
- El *Heap* o montón está ***ordenado*** si:
  ningún elemento tiene menor clave que alguno de sus dos hijos.

- Un elemento mal colocado se recoloca, bajándolo o subiéndolo recursivamente mientras haga falta:
  - Se **baja** intercambiándolo <u>*recursivamente*</u> con su hijo mayor,
  - Se **sube** intercambiándolo <u>*recursivamente*</u> con su padre.

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Insertar y Eliminar

- Dada la secuencia ya ordenada según un montón desde la posición ***i*+1** hasta la posición ***n***, se **inserta** el nuevo elemento de la posición ***i*** y, para mantener la ordenación, se baja *mientras haga falta* (es decir; hasta que no tenga hijos o, de tenerlos, ninguno tenga menor clave).
- Para **eliminar** el primer elemento del montón ordenado, se intercambia con el último elemento; y, si queda mal colocado, se baja *mientras haga falta*.
- La eliminación de elementos implica **actualizar** el tamaño ***n*** del heap o montón.

Ordenar con Heap

- Para **ordenar** los elementos de una secuencia por el método de ordenación por selección con el uso de un ***heap***:
  - En una ***primera*** fase,
    - se ordena la secuencia como un montón incorporando al montón los elementos uno a uno, bajándolo si hace falta.
  - En la ***segunda*** fase,
    - se va seleccionando iterativamente el elemento de mayor clave, que será siempre el que está en la raíz o primera posición, para eliminarlo y bajar mientras haga falta el que ocupa su lugar

Ejemplo del HeapSort:

Inserciones:

1	2	3	4	5	6	7	8
44	55	12	42	94	18	06	67
44	55	12	**67**	94	18	06	**42**
44	55	**18**	67	94	**12**	06	42
44	**94**	18	67	**55**	12	06	42
94	**67**	18	**44**	55	12	06	42

Extracciones:

1	2	3	4	5	6	7	8		1	2	3	4	5	6	7	8
42	67	18	44	55	12	06	**94**	→	**67**	**55**	18	44	**42**	12	06	94
06	55	18	44	42	12	**67**	94	→	**55**	**44**	18	**06**	42	12	67	94
12	44	18	06	42	**55**	67	94	→	**44**	**42**	18	06	**12**	55	67	94
12	42	18	06	**44**	55	67	94	→	**42**	**12**	18	06	44	55	67	94
06	12	18	**42**	44	55	67	94	→	**18**	12	**06**	42	44	55	67	94
06	12	**18**	42	44	55	67	94	→	**12**	**06**	18	42	44	55	67	94
06	**12**	18	42	44	55	67	94	→	06	12	18	42	44	55	67	94

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

Análisis del Algoritmo

- El algoritmo **HeapSort**, que implementa el método de ordenación por selección con heap siguiendo la propuesta de Floyd, es **O(*n* log *n*)**.
  - Sólo se utiliza el procedimiento baja, que es **O(log *n*)**.
  - El número de veces que se baja un elemento en cada fase es:

    n*/2** en la primera fase y ***n en la segunda fase.

    por tanto el procedimiento baja se aplica **O(*n*)** veces en total.

- La complejidad total resultante es: **O(*n*) · O(log *n*) = O(*n* log *n*)**

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

El código de baja

```
void baja( int i ; Tvector &sec ; int n ) {
   while ( 2*i <= n ){
       h1 = 2*i ;
       h2 = h1 + 1 ;
       if (h1 == n)
          h = h1
       else if (sec[h1] > sec[h2])
               h = h1
            else h = h2 ;
       if (sec[h] <= sec[i])
           break ;
       else {
          swap(sec[i],sec[h]) ;
          i = h ;
       } ;
   } ;
} ;
```

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

El código de *HeapSort*

```
void heapsort( Tvector sec ; int n ) {
   for (int i = n/2; i > 0; i--)
       baja(i, sec, n) ;
   for (int i = n; i > 1; i--) {
       swap(sec[1],sec[i]);
       baja(1,sec,i-1) ;
   } ;
}
```

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

ORDENACIÓN LOGARÍTMICA

1. ALGORITMOS CUADRÁTICOS DE ORDENACIÓN
   - √ Inserción, Selección, por Intercambio
2. **ALGORITMOS LOGARÍTMICOS DE ORDENACIÓN**
   - √ Algoritmo de ordenación HeapSort
   - ➢ **Algoritmo de ordenación QuickSort**
   - ● Algoritmo de ordenación MergeSort
4. OTROS ALGORITMOS DE ORDENACIÓN

   Incrementos decrecientes, Radicales, TimSort

Ordenación por descomposición

- Existen varios métodos de ordenación que se basan en la descomposición del problema para aplicar la técnica denominada ***divide y vencerás***.
- Los más importantes son:
  - El método de descomposición por pivote denominado **ordenación rápida** o **QuickSort** y
  - El método de descomposición por posición denominado **ordenación por mezcla** o **MergeSort**.

AMBOS SON LOGARÍTMICOS $O(n \log_2 n)$

Descomposición por pivote

- El algoritmo denominado **QuickSort** es originalmente debido a ***Hoare*** y utiliza una clave **pivote** para descomponer la secuencia.
- Se recorre la secuencia desde sus extremos en **ambos sentidos** comparando las claves de los elementos con el pivote.
  - El recorrido ascendente se para en el primer elemento con clave **mayor** que el pivote.
  - El recorrido descendente se para en el primer elemento con clave **menor** que el pivote.
- Se **intercambian** estos pares de elementos y se **continúan** los recorridos hasta que se encuentren.
- En ese momento, la secuencia está dividida entre dos subsecuencias,
  - la primera con elementos de clave **menor** que el pivote y
  - la segunda con elementos de clave **mayor** que el pivote.

Ejemplo

44 55 12 42 94 18 06 67 PIV: 42

Los recorridos ascendente y descendente se paran en **44** y **06**

44 55 12 42 94 18 06 67

Se intercambian estos elementos:

06 55 12 42 94 18 **44** 67

Se continúan los recorridos hasta que vuelven a detenerse en **55** y **18**

06 55 12 42 94 18 44 67

que se intercambian

06 **18** 12 42 94 **55** 44 67

Se continúan los recorridos hasta que se encuentran en **42**

06 18 12 42 94 55 44 67

La secuencia queda dividida en dos subsecuencias por la posición de **cruce**, una con elementos menores que **42** y la otra con elementos mayores que **42**

06 18 12 42 94 55 44 67

Recursión

- Con cada una de las dos secuencias en que queda dividida se itera **recursivamente** el proceso.
- Aunque se puede usar como pivote un valor que no coincida con ninguna clave, se suele usar como pivote la **clave** de un elemento, generalmente el primero o el del medio.
- ***Otras propuestas***:
  - Uno en particular: el primero, el último, el del medio
  - Una media: entre el primero y el último, entre el primero, el último y el del medio, ¿ponderada?
  - La mediana, la mediana entre varios, ...

Ejemplo

El proceso es el siguiente:

(42)	44	55	12	42	94	18	06	67
PIVOTE 42	44	55	12	42	94	18	06	67
posición 3	**06**	55	12	42	94	18	**44**	67
= (0 + 7)/2	06	**18**	12	42	94	**55**	44	67
	06	18	12	42	94	55	44	67

- *¿qué hacemos cuando nos encontremos justo con el pivote?*
- *¿qué ocurre si no hay ninguno mayor que el pivote?*

(18) 06 18 12 42 (55) 94 55 44 67

06 12 18 44 55 94 67

(12) 06 12 18 42 44 55 (94) 94 67

- *¿en qué orden se hacen las divisiones?*

Resultado: 06 12 18 42 44 55 67 94

Revisión - QuickSort

- Se utiliza una clave **pivote** para descomponer la secuencia.
- Se recorre la secuencia desde sus extremos en **ambos sentidos** comparando las claves de los elementos con el pivote.
  - El recorrido ascendente se para en el primer elemento con clave **mayor *o igual*** que el pivote.
  - El recorrido descendente se para en el primer elemento con clave **menor *o igual*** que el pivote.
- Se **intercambian** estos pares y se **continúan** los recorridos hasta que se ***crucen***.
- En ese momento, la secuencia está dividida entre dos subsecuencias,
  - la primera con elementos de clave **menor *o igual*** que el pivote y
  - la segunda con elementos de clave **mayor *o igual*** que el pivote.

Mejoras

- Una mejora práctica se obtiene al ***estimar*** la mediana (*el valor desconocido que dividiría la secuencia en dos partes del mismo tamaño*); como la media de las claves de los extremos y la del medio.

- El procedimiento natural que implementa el método de ordenación rápida o **QuickSort** es claramente **recursivo**.

- El algoritmo de ordenación rápida o **QuickSort** resultante es:
  $\mathbf{O}(n^2)$ en el peor caso pero es $\mathbf{O}(n \log n)$ en el caso medio.

Código

```
void Qsort( sec, ini, fin) {
    i = ini ; f = fin ;
    p = sec[(i+f)/2] ;
    while (i <= f){
        while (sec[i] < p) i++ ;
        while (sec[f] > p) f-- ;
        if (i <= f) {
            swap(sec[i],sec[f]) ;
            i++ ;
            f-- ;
        } ;
    } ;
    if (ini < f) Qsort(sec, ini, f) ;
    if (i < fin) Qsort(sec, i, fin) ;
} ;
```

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Ejemplo (I)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	
13	3	4	12	14	10	5	1	8	2	7	9	11	6	Pivote: 5
13	3	4	12	14	10	5	1	8	2	7	9	11	6	i=0; f=9
2	3	4	12	14	10	5	1	8	**13**	7	9	11	6	i=3; f=7
2	3	4	**1**	14	10	5	**12**	8	13	7	9	11	6	i=4; f=6
2	3	4	1	**5**	10	**14**	12	8	13	7	9	11	6	i=5; f=5

2	**3**	**4**	**1**	**5**	10	14	12	8	13	7	9	11	6	Pivote: 4
2	3	**4**	**1**	5	10	14	12	8	13	7	9	11	6	i=2; f=3
2	3	1	4	5	10	14	12	8	13	7	9	11	6	i=3; f=2
2	3	1	4	5	10	14	12	8	13	7	9	11	6	

Ejemplo (II)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	
13	3	4	12	14	10	5	1	8	2	7	9	11	6	Pivote: 5
2	**3**	**4**	**1**	**5**	10	14	12	8	13	7	9	11	6	Pivote: 4
2	**3**	**1**	4	5	10	14	12	8	13	7	9	11	6	Pivote: 3
2	**1**	3	4	5	10	14	12	8	13	7	9	11	6	Pivote: 2
1	2	3	**4**	**5**	10	14	12	8	13	7	9	11	6	Pivote: 4
1	2	3	4	5	**10**	**14**	**12**	**8**	**13**	**7**	**9**	**11**	**6**	Pivote: 13
1	2	3	4	5	**10**	**6**	**12**	**8**	**11**	**7**	**9**	13	14	Pivote: 8
1	2	3	4	5	**7**	**6**	**8**	12	11	10	9	13	14	Pivote: 6
1	2	3	4	5	6	**7**	**8**	12	11	10	9	13	14	Pivote: 7
1	2	3	4	5	6	7	8	**12**	**11**	**10**	**9**	13	14	Pivote: 11
1	2	3	4	5	6	7	8	**9**	**10**	11	12	13	14	Pivote: 9
1	2	3	4	5	6	7	8	9	10	**11**	**12**	13	14	Pivote: 11
1	2	3	4	5	6	7	8	9	10	11	12	**13**	**14**	Pivote: 13

ORDENACIÓN LOGARÍTMICA

1. ALGORITMOS CUADRÁTICOS DE ORDENACIÓN
   - √ Inserción, Selección, por Intercambio
2. **ALGORITMOS LOGARÍTMICOS DE ORDENACIÓN**
   - √ Algoritmo de ordenación HeapSort
   - √ Algoritmo de ordenación QuickSort
   - ➢ **Algoritmo de ordenación MergeSort**
4. OTROS ALGORITMOS DE ORDENACIÓN

   Incrementos decrecientes, Radicales, TimSort

Descomposición por posición

- El método de **ordenación por mezcla** o **MergeSort** se basa en la ***descomposición*** de la secuencia a ordenar en dos *subsecuencias* que, una vez ordenadas, se mezclan ordenadamente.

- La división se realiza por la **posición media** y la mezcla se realiza seleccionando el elemento ***cabecera*** de las dos subsecuencias de ***menor clave*** hasta que una de ellas quede vacía.

- Entonces se pega la *cola* de la otra subsecuencia.

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Ejemplo

Secuencia:	44	55	12	42	94	18	06	67
División:	44	55	12	42	94	18	06	67
División:	44	55	12	42	94	18	06	67
División:	44	55	12	42	94	18	06	67
Mezclado:	44	55	12	42	94	18	06	67
División:	44	55	12	42	94	18	06	67
Mezclado:	44	55	12	42	94	18	06	67
Mezclado:	12	42	44	55	94	18	06	67
División:	12	42	44	55	94	18	06	67
División:	12	42	44	55	94	18	06	67
Mezclado:	12	42	44	55	18	94	06	67
División:	12	42	44	55	18	94	06	67
Mezclado:	12	42	44	55	18	94	06	67
Mezclado:	12	42	44	55	06	18	67	94
Mezclado:	06	12	18	42	44	55	67	94

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Detalle de la mezcla:

```
                                          |-12---42---44---55--
1)                                    |-- |-06---18---67---94--
                                          |-12---42---44---55
2)                                 |-06-  |-18---67---94
                                          |-42---44---55--
3)                              |-06-12-  |-18---67---94--
                                          |-42---44---55
4)                           |-06-12-18-  |-67---94
                                          |-44---55--
5)                        |-06-12-18-42-  |-67---94--
                                          |-55
6)                     |-06-12-18-42-44-  |-67---94
                                          |--
7)                  |-06-12-18-42-44-55-  |-67---94--
                                          |--
8)|-06-12-18-42-44-55-67-94-              |--
```

Implementación de MergeSort

- Para la operación de mezcla se necesita un array auxiliar.
- El procedimiento es recursivo y el algoritmo **MergeSort** resultante es: **O(*n* log *n*)**.

```
  void Msort (sec, ini, fin){
    if (ini < fin){
       cen = (ini + fin) / 2 ;
       Msort(sec, ini, cen) ;
       Msort(sec, cen+1, fin) ;
       Mezcla(sec, ini, cen, fin) ;
    } ;
} ;
```

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Código de la Mezcla (I)

```
void Mezcla (sec, ini, cen, fin){
   i = ini ; j = cen + 1 ; k = ini ;
   while ((i <= cen) && (j <= fin)){
      if (sec[i] < sec[j]){
         aux[k] = sec[i] ;
         i++ ;
      }
      else{
         aux[k] = sec[j] ;
         j++ ;
      }
   k++ ;
   }
   . . . .
} ;
```

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

Código de la Mezcla (II)

```
void Mezcla (sec, ini, cen, fin)
{...
  if (i > cen)
      while (j <= fin){
          aux[k] = sec[j] ;
          j++ ; k++ ;
      }
  else
      while (i <= cen){
          aux[k] = sec[i] ;
          i++ ; k++ ;
      } ;
  for (int k = ini; k <= fin; k++)
      sec[k] = aux[k] ;
} ;
```

Mezcla alternativa

```
void Mezcla (sec, ini, cen, fin){
   i = ini; j = cen+1;
   for (int k = ini, k <= fin; k++){
      if (i < cen &&
         (j > fin || sec[i] < sec[j])){
         aux[k] = sec[i];
         i++;
      }
      else {
         aux[k] = sec[j];
         j++;
      }
   };
   for (int k = ini; k <= fin; k++)
     sec[k] = aux[k] ;
};
```

OTROS ALGORITMOS

1. ALGORITMOS CUADRÁTICOS DE ORDENACIÓN
   - √ Inserción, Selección, por Intercambio
2. ALGORITMOS LOGARÍTMICOS DE ORDENACIÓN
   - √ HeapSort, QuickSort, MergeSort
3. **OTROS ALGORITMOS DE ORDENACIÓN**
   - ➢ **Ordenación por Incrementos decrecientes;** *ShellSort*
   - ● Ordenación por Radicales; *RadixSort*
   - ● Ordenación de Java y Python; o *TimSort*

Ordenación por Incrementos Decrecientes

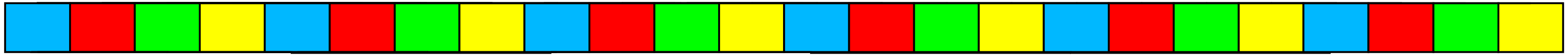
- **Inserción con intercambios más alejados**

- **ShellSort**: dividir por 2. $\mathbf{O}(n^2)$

- Mejoras: $\mathbf{O}(n^{1.5})$, $\mathbf{O}(n^{1.3})$, $\mathbf{O}(n^{1.15})$ *Hibbard*, *Sedgewick*

Ordenación por Incrementos Decrecientes

- En el método de **incrementos decrecientes:** se ordenan por inserción los elementos de la secuencia que están entre sí a una **distancia** o ***incremento* δ**, **rebajando** sucesivamente el incremento **δ** hasta llegar a **1**
- Se considera que la secuencia está descompuesta en δ subsecuencias:
  - La subsecuencia k-ésima (k **= 0,…, δ−1**) está formada por los elementos en las posiciones ***k*, *k*+δ, *k*+2δ, *k*+3δ, …**
- Se denomina **δ-ordenación** al proceso de ordenación de todas las subsecuencias, para cada valor de **δ**.



Ejemplo (primera pasada).

Tomamos las distancias o incrementos decrecientes $\delta = 4, 2, 1$:

44	**55**	**12**	**42**	**94**	**18**	**06**	**67**

Primera pasada $\delta = 4$.

Descomposición	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Subsecuencia 1:	44	---	---	---	94	---	---	---
Subsecuencia 2:	---	55	---	---	---	18	---	---
Subsecuencia 3:	---	---	12	---	---	---	06	---
Subsecuencia 4:	---	---	---	42	---	---	---	67

4-ordenaciones	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
Subsecuencia 1:	44	---	---	---	94	---	---	---
Subsecuencia 2:	---	**18**	---	---	---	**55**	---	---
Subsecuencia 3:	---	---	**06**	---	---	---	**12**	---
Subsecuencia 4:	---	---	---	42	---	---	---	67

Sec. 4-ordenada	44	18	06	42	94	55	12	67

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

Ejemplo (otras pasadas)

Segunda pasada $\delta = 2$:

Descomposición

0	1	2	3	4	5	6	7
44	------	06	------	94	------	12	----
----	18	------	42	------	55	------	67

2-ordenaciones

0	1	2	3	4	5	6	7
06	------	**12**	------	**44**	------	**94**	----
----	18	------	42	------	55	------	67

2-ordenada

06	18	12	42	44	55	94	67

Tercera pasada $\delta = 1$:

Descomposición

0	1	2	3	4	5	6	7
06	18	12	42	44	55	94	67

1-ordenaciones

0	1	2	3	4	5	6	7
06	**12**	**18**	**42**	**44**	**55**	**67**	**94**

Condiciones de los incrementos

- La efectividad del método queda garantizada porque ***termina*** en una <u>1-ordenación</u> que equivale a la aplicación del método de ordenación por inserción aplicado a toda la secuencia.

- La ventaja consiste en que con incrementos *grandes* se van realizando ***grandes*** desplazamientos de los elementos muy mal colocados y posteriormente se ***refinan*** las colocaciones a menor distancia.

- Cualquier sucesión de valores <u>**decrecientes**</u> para los incrementos δ, que acabe con $\boldsymbol{\delta = 1}$ puede ser aplicada.

- Conviene no usar sucesiones de incrementos que sean ***múltiplos*** o que tengan ***divisores*** comunes porque se repiten comparaciones.

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Ejemplo

Sucesión de incrementos `5,3,1`:

	13	3	4	12	14	10	5	1	8	2	7	9	11	6
δ = 5	13	3	4	12	14	10	5	1	8	2	7	9	11	6
	7	3	4	12	14	**10**	5	1	8	2	**13**	9	11	6
	7	**3**	4	12	14	10	**5**	1	8	2	13	**9**	11	6
	7	3	**1**	12	14	10	5	**4**	8	2	13	9	**11**	6
	7	3	1	**6**	14	10	5	4	**8**	2	13	9	11	**12**
	7	3	1	6	**2**	10	5	4	8	**14**	13	9	11	12
δ = 3	7	3	1	6	2	10	5	4	8	14	13	9	11	12
	5	3	1	**6**	2	10	**7**	4	8	**11**	13	9	**14**	12
	5	**2**	1	6	**3**	10	7	**4**	8	11	**12**	9	14	**13**
	5	2	**1**	6	3	**8**	7	4	**9**	11	12	**10**	14	13
δ = 1	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Notas:

- Se subrayan los elementos de cada una de las subsecuencias
- Se ponen en negrita los elementos una vez ordenados entre sí

Implementación basada en la Ordenación por Inserción

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

- La implementación está basada en la **ordenación por inserción** de cada subsecuencia, pero se realiza un recorrido ***simultáneo*** de todas ellas y no de forma sucesiva, de manera que se compara cada objeto de la secuencia con el objeto que ocupa **δ** lugares más adelante.

- **Se compara**, de *izquierda a derecha*, ***cada elemento con el que está δ posiciones más adelante,*** intercambiándolos *si es necesario*, hasta que *no deba adelantarse más*.

- Se actúa como en la ordenación por inserción pero tomando siempre un paso de ***δ*** posiciones en lugar de ir de uno en uno

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

Detalles: δ = 5

Sucesión de incrementos 5,3,1:

Secuencia	13	3	4	12	14	10	5	1	8	2	7	9	11	6
δ = 5	**13**	3	4	12	14	**10**	5	1	8	2	7	9	11	6
	10	**3**	4	12	14	13	**5**	1	8	2	7	9	11	6
	10	3	**4**	12	14	13	5	**1**	8	2	7	9	11	6
	10	3	1	**12**	14	13	5	4	**8**	2	7	9	11	6
	10	3	1	8	**14**	13	5	4	12	**2**	7	9	11	6
	10	3	1	8	2	**13**	5	4	12	14	**7**	9	11	6
	7	**3**	1	8	2	10	**5**	4	12	14	13	**9**	11	6
	7	3	**1**	8	2	10	5	**4**	12	14	13	9	**11**	6
	7	3	1	**8**	2	10	5	4	**12**	14	13	9	11	**6**
	7	3	1	6	2	10	5	4	8	14	13	9	11	12
5-orden:	7	3	1	6	2	10	5	4	8	14	13	9	11	12

Notas:
- Se subrayan los elementos que ya están **δ** (= 5) ordenados
- En negrita están los elementos afectados por cada **δ** inserción

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

Detalles: δ = 3

Sucesión de incrementos **5, 3, 1**:

5-orden:	7	3	1	6	2	10	5	4	8	14	13	9	11	12
δ = 3	**7**	3	1	**6**	2	10	5	4	8	14	13	9	11	12
	6	**3**	1	7	**2**	10	5	4	8	14	13	9	11	12
	6	2	**1**	7	3	**10**	5	4	8	14	13	9	11	12
	6	2	1	**7**	3	10	**5**	4	8	14	13	9	11	12
	5	**2**	1	6	**3**	10	7	**4**	8	14	13	9	11	12
	5	2	**1**	6	3	**10**	7	4	**8**	14	13	9	11	12
	5	2	1	**6**	3	8	**7**	4	10	**14**	13	9	11	12
	5	**2**	1	6	**3**	8	7	**4**	10	14	**13**	9	11	12
	5	2	**1**	6	3	**8**	7	4	**10**	14	13	**9**	11	12
	5	2	1	**6**	3	8	**7**	4	9	**14**	13	10	**11**	12
	5	**2**	1	6	**3**	8	7	**4**	9	11	**13**	10	14	**12**
	5	2	1	6	3	8	7	4	9	11	12	10	14	13
3-orden:	5	2	1	6	3	8	7	4	9	11	12	10	14	13

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

Detalles: δ = 1

3-orden:	5	2	1	6	3	8	7	4	9	11	12	10	14	13
δ = 1	5	**2**	1	6	3	8	7	4	9	11	12	10	14	13
	2	5	**1**	6	3	8	7	4	9	11	12	10	14	13
	1	2	5	**6**	3	8	7	4	9	11	12	10	14	13
	1	2	5	6	**3**	8	7	4	9	11	12	10	14	13
	1	2	3	5	6	**8**	7	4	9	11	12	10	14	13
	1	2	3	5	6	8	**7**	4	9	11	12	10	14	13
	1	2	3	5	6	7	8	**4**	9	11	12	10	14	13
	1	2	3	4	5	6	7	8	**9**	11	12	10	14	13
	1	2	3	4	5	6	7	8	9	**11**	12	10	14	13
	1	2	3	4	5	6	7	8	9	11	**12**	10	14	13
	1	2	3	4	5	6	7	8	9	11	12	**10**	14	13
	1	2	3	4	5	6	7	8	9	10	11	12	**14**	13
	1	2	3	4	5	6	7	8	9	10	11	12	14	**13**
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1-orden:	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Reducción de los incrementos

- La ***propuesta de Shell*** para una secuencia de tamaño $\boldsymbol{n}$ consiste en:
  - iniciar las $\boldsymbol{\delta}$-ordenaciones con $\boldsymbol{\delta} = \mathrm{E}[\boldsymbol{n}/\mathbf{2}]$, y
  - continuar con $\boldsymbol{\delta} \leftarrow \mathrm{E}[\boldsymbol{n}/\mathbf{2}]$ (E[$\boldsymbol{x}$] es la parte entera de $\boldsymbol{x}$)
  - hasta que $\boldsymbol{\delta} = 1$.

- Se han realizado propuestas similares para:
  - iniciar los incrementos con $\boldsymbol{\delta} = \mathrm{E}[\boldsymbol{n} \cdot \boldsymbol{a}]$, y
  - seguir con $\boldsymbol{\delta} \leftarrow \mathrm{E}[\boldsymbol{\delta} \cdot \boldsymbol{a}]$ $(\mathbf{0} < \boldsymbol{a} < \mathbf{1})$
  - hasta $\boldsymbol{\delta} = \mathbf{1}$.

- Una de las sucesiones que *mejores* resultados proporciona es la consistente en usar los valores $\boldsymbol{a} = \mathbf{1/3}$ ó $\boldsymbol{a} = \mathbf{0.45454}$; esta última posibilidad es equivalente a aplicar la ***fórmula*** $\boldsymbol{\delta} \leftarrow \mathrm{E}[\,(\mathbf{5}\boldsymbol{\delta} - \mathbf{1}) / \mathbf{11}\,]$.

El código ShellSort

```
delta = n ;
while (delta > 1){
   delta = delta / 2 ;
   deltasort(delta,sec, n) ;
} ;
void deltasort( int delta ; TVector Sec ; int n ) {
   for (int i = delta; i < n; i++){
      x = sec[i] ;
      j = i ;
      while ((j >= delta) && (x < sec[j - delta])){
         sec[j] = sec[j - delta] ;
         j = j - delta ;
      } ;
   sec[j] = x ;
   } ;
} ;
```

Sucesiones de Incrementos

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

- Varios **autores** proponen: *sucesiones de incrementos* $\boldsymbol{\delta_k}$ *que se construyen en orden creciente, con* $\boldsymbol{\delta_0 = 1}$, aunque se utilizan en <u>orden decreciente</u> desde <u>el mayor incremento que es menor que el tamaño</u> $\boldsymbol{n}$ de la secuencia.
  - La fórmula propuesta por ***Hibbard*** es $\boldsymbol{\delta_k = 2^k - 1}$ que da lugar a la sucesión de incrementos: **{ 1, 3, 7, 15, 31, 63, 127, 255, 511, ... }**
  - Entre los **mejores** resultados empíricos con estas sucesiones están los de la sucesión de incrementos **{1, 4, 13, 40, 121, 364, 1093, ... }** obtenida con la fórmula $\boldsymbol{\delta_k = (3^k-1)/2}$ ó con $\boldsymbol{\delta \leftarrow 3\delta + 1}$.
  - La propuesta de ***Sedgewick*** es usar la fórmula $\boldsymbol{\delta_k = (4^{k+1} + 3\cdot 2^k + 1)}$ para obtener la sucesión de incrementos **{1, 8, 23, 77, 281, 1073, ... }**
  - Entre diversas propuestas similares de ***Sedgewick***, la que da mejores resultados prácticos es la de usar <u>*conjuntamente*</u> los incrementos de $\boldsymbol{\delta_k = (4^k - 3\cdot 2^k + 1)}$ y de $\boldsymbol{\delta_k = (9\cdot 4^k - 9\cdot 2^k + 1)}$ que es la sucesión: **{1, 5, 19, 41, 209, 233, 505, 921, …}**.

Aplicación de una sucesión de Incrementos

- Para iniciar las ***δ***-ordenaciones, se busca el **mayor valor** de la sucesión de incrementos que es **menor** que el tamaño ***n*** de la secuencia

- En las implementaciones se *recomienda* utilizar los valores de la sucesión de incrementos dados **explícitamente** como datos, sin necesidad de aplicar ninguna *fórmula* para calcularlos.

Análisis del Algoritmo

- El tiempo de ejecución del método de ordenación por incrementos decrecientes **depende** de la elección de la ***sucesión de incrementos***.
- Se han probado multitud de *sucesiones explícitas* u obtenidas mediante diferentes **fórmulas** como las mostradas anteriormente.
- Con la sucesión **propuesta *por*** *Shell* es un algoritmo $\boldsymbol{\Theta(n^2)}$.
- Con la fórmula $\boldsymbol{\delta_k = 2^k – 1}$ propuesta por *Hibbard* es un algoritmo:

  $\boldsymbol{\Theta(n^{3/2})}$ en el *peor* caso y se cree que es $\boldsymbol{O(n^{5/4})}$ en el caso *medio*.
- Con la sucesión $\boldsymbol{\delta_k = (4^{k+1} + 3 \cdot 2^k + 1)}$, propuesta por *Sedgewick*, resulta:

  $\boldsymbol{O(n^{4/3})}$ en el *peor* caso y parece que es $\boldsymbol{O(n^{7/6})}$ en el caso *medio*.

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

OTROS ALGORITMOS

1. ALGORITMOS CUADRÁTICOS DE ORDENACIÓN
   - √ Inserción, Selección, por Intercambio
2. ALGORITMOS LOGARÍTMICOS DE ORDENACIÓN
   - √ HeapSort, QuickSort, MergeSort
3. **OTROS ALGORITMOS DE ORDENACIÓN**
   - √ Ordenación por Incrementos decrecientes o *ShellSort*
   - ➢ **Ordenación por Radicales** o *RadixSort*
   - ● Ordenación de Java y Python o *TimSort*

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

La ordenación por radicales

- La ordenación por radicales de la que se deriva el método RadixSort parte de la idea de que para los números de una secuencia se pueden ordenar ***sucesivamente*** por cada uno de sus ***dígitos*** desde el último al primero manteniendo la ordenación previa en caso de empate.

- En este **ejemplo** se ordenan **9** números de **3** cifras:

Secuencia original	**516**	**223**	**323**	**413**	**416**	**723**	**813**	**626**	**616**
Ordenada por el último dígito:	**223**	**323**	**413**	**723**	**813**	**516**	**416**	**626**	**616**
Ordenada por el segundo dígito:	**413**	**813**	**516**	**416**	**616**	**223**	**323**	**723**	**626**
Ordenada por el primer dígito:	**223**	**323**	**413**	**416**	**516**	**616**	**626**	**723**	**813**

Ordenación por apilamiento

- El método básico **RadixSort** se describe utilizando la **expresión decimal** de una clave entera, pero se pueden usar otras muchas representaciones: binaria, octal, hexadecimal, …
- Este método de ordenación llamado por **apilamiento** o por **cubetas** consiste en **repartir** en sucesivas pasadas los elementos en pilas o cubetas según cada dígito desde el último al primero.
- Se tienen **10 cubetas** con los números del **0** al **9**.
- Se **apilan** desde la secuencia dada cada elemento según el **dígito *i***-esimo y luego se **recogen** ordenadamente, variando ***i*** desde el **último** al **primero**.
- Si el número de dígitos se considera acotado por una ***constante***, entonces **el algoritmo es $O(n)$** ya que se realiza una pasada por cada posible dígito en la que son necesarias $O(n)$ operaciones.

Ejemplo. (Fase I)

```
345   721   425   572   836   467   672   194   365
236   891   746   431   834   247   529   216   389      ==>
```

						216			
	431				365	746			
	891	672		834	425	236	247		389
	721	572		194	345	836	467		529
0	1	2	3	4	5	6	7	8	9

```
==>   721   891   431   572   672   194   834   345   425
      365   836   236   746   216   467   247   529   389
```

Ejemplo. (Fase II)

```
721 891 431 572 672 194 834 345 425
365 836 236 746 216 467 247 529 389   ==>
```

			236						
		529	836	247					
		425	834	746		467	672		194
	216	721	431	345		365	572	389	891
0	1	2	3	4	5	6	7	8	9

```
==>     216 721 425 529 431 834 836 236 345
        746 247 365 467 572 672 389 891 194
```

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Ejemplo. (Fase III)

216 721 425 529 431 834 836 236 245
746 247 365 467 572 672 389 891 194 ==>

		247	389	467				891	
		236	365	431	572		746	836	
	194	216	345	425	529	672	721	834	
0	1	2	3	4	5	6	7	8	9

=> 194 216 236 247 345 365 389 425 431
467 529 572 672 721 746 834 836 891

Ejemplo (Fase II): Detalles

```
721 891 431 572 672 194 834 345 425

365 836 236 746 216 467 247 529 389   ==>
```

```
____  ____  ____  ____  ____  ____  ____  ____  ____  ____
 0     1     2     3     4     5     6     7     8     9
```

Ejemplo (Fase II): Detalles

```
     891 431 572 672 194 834 345 425
365 836 236 746 216 467 247 529 389  ==>
```

0	1	2	3	4	5	6	7	8	9
		721							

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Ejemplo (Fase II): Detalles

431 572 672 194 834 345 425

365 836 236 746 216 467 247 529 389 ==>

0	1	2	3	4	5	6	7	8	9
		721							891

Ejemplo (Fase II): Detalles

572 672 194 834 345 425

365 836 236 746 216 467 247 529 389 ==>

0	1	2	3	4	5	6	7	8	9
		721	431						891

Ejemplo (Fase II): Detalles

672 194 834 345 425

365 836 236 746 216 467 247 529 389 ==>

0	1	2	3	4	5	6	7	8	9
		721	431				572		891

Ejemplo (Fase II): Detalles

194 834 345 425

365 836 236 746 216 467 247 529 389 ==>

							672		
		721	431				572		891
0	1	2	3	4	5	6	7	8	9

Ejemplo (Fase II): Detalles

834 345 425

365 836 236 746 216 467 247 529 389 ==>

							672		194
		721	431				572		891
0	1	2	3	4	5	6	7	8	9

Ejemplo (Fase II): Detalles

```
                                    345 425
365 836 236 746 216 467 247 529 389   ==>
```

			834				672		194
		721	431				572		891
0	1	2	3	4	5	6	7	8	9

Ejemplo (Fase II): Detalles

```
                                                      425
365 836 236 746 216 467 247 529 389   ==>



                          834                            672           194
                  721     431     345                    572           891
_____   _____   _____   _____   _____   _____   _____   _____   _____   _____
  0       1       2       3       4       5       6       7       8       9
```

Ejemplo (Fase II): Detalles

```
365 836 236 746 216 467 247 529 389  ==>
```

		425	834				672		194
		721	431	345			572		891
0	1	2	3	4	5	6	7	8	9

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Ejemplo (Fase II): Detalles

```
836 236 746 216 467 247 529 389   ==>
```

		425	834				672		194
		721	431	345		365	572		891
0	1	2	3	4	5	6	7	8	9

Ejemplo (Fase II): Detalles

236 746 216 467 247 529 389 ==>

			836						
		425	834				672		194
		721	431	345		365	572		891
0	1	2	3	4	5	6	7	8	9

Ejemplo (Fase II): Detalles

```
746 216 467 247 529 389  ==>
```

0	1	2	3	4	5	6	7	8	9
		721	431	345		365	572		891
		425	834				672		194
			836						
			236						

Ejemplo (Fase II): Detalles

216 467 247 529 389 ==>

			236						
			836						
		425	834	746			672		194
		721	431	345		365	572		891
0	1	2	3	4	5	6	7	8	9

Ejemplo (Fase II): Detalles

467 247 529 389 ==>

			236						
			836						
		425	834	746			672		194
	216	721	431	345		365	572		891
0	1	2	3	4	5	6	7	8	9

Ejemplo (Fase II): Detalles

247 529 389 ==>

0	1	2	3	4	5	6	7	8	9
			236						
			836						
		425	834	746		467	672		194
	216	721	431	345		365	572		891

Ejemplo (Fase II): Detalles

529 389 ==>

			236						
			836	247					
		425	834	746		467	672		194
	216	721	431	345		365	572		891
0	1	2	3	4	5	6	7	8	9

Ejemplo (Fase II): Detalles

389 ==>

			236						
		529	836	247					
		425	834	746		467	672		194
	216	721	431	345		365	572		891
0	1	2	3	4	5	6	7	8	9

Ejemplo (Fase II): Detalles

==>

			236						
		529	836	247					
		425	834	746		467	672		194
	216	721	431	345		365	572	389	891
0	1	2	3	4	5	6	7	8	9

Ejemplo (Fase II): Detalles

==>

			236						
		529	836	247					
		425	834	746		467	672		194
	216	721	431	345		365	572	389	891
0	1	2	3	4	5	6	7	8	9

Ejemplo (Fase II): Detalles

			236						
		529	836	247					
		425	834	746		467	672		194
		721	431	345		365	572	389	891
0	1	2	3	4	5	6	7	8	9

==> 216

Ejemplo (Fase II): Detalles

			236						
			836	247					
			834	746		467	672		194
			431	345		365	572	389	891
0	1	2	3	4	5	6	7	8	9

==> 216 721 425 529

Ejemplo (Fase II): Detalles

				247					
				746		467	672		194
				345		365	572	389	891
0	1	2	3	4	5	6	7	8	9

==> 216 721 425 529 431 834 836 236

Ejemplo (Fase II): Detalles

						467	672		194
						365	572	389	891
0	1	2	3	4	5	6	7	8	9

==> 216 721 425 529 431 834 836 236 345
746 247

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

Ejemplo (Fase II): Detalles

							672		194
							572	389	891
0	1	2	3	4	5	6	7	8	9

==> 216 721 425 529 431 834 836 236 345
746 247 365 467

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

Ejemplo (Fase II): Detalles

0	1	2	3	4	5	6	7	8	9
									194
								389	891

==> 216 721 425 529 431 834 836 236 345
746 247 365 467 572 672

Ejemplo (Fase II): Detalles

									194
									891
0	1	2	3	4	5	6	7	8	9

```
==>  216 721 425 529 431 834 836 236 345
     746 247 365 467 572 672 389
```

Ejemplo (Fase II): Detalles

____	____	____	____	____	____	____	____	____	____
0	1	2	3	4	5	6	7	8	9

```
==>   216 721 425 529 431 834 836 236 345
      746 247 365 467 572 672 389 891 194
```

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

ALGORITMOS DE ORDENACIÓN

1. ALGORITMOS CUADRÁTICOS DE ORDENACIÓN

   √ Inserción, Selección, por Intercambio

2. ALGORITMOS LOGARÍTMICOS DE ORDENACIÓN

   √ HeapSort, QuickSort, MergeSort

3. **OTROS ALGORITMOS DE ORDENACIÓN**
   √ Ordenación por Incrementos decrecientes; *ShellSort*
   √ Ordenación por Radicales; *RadixSort*
   ➢ **Ordenación de Python y Java; *TimSort***

Departamento de Ingeniería Informática y de Sistemas
Universidad de La Laguna

El método TimSort

- El **TimSort** es el método de ordenación propuesto por Tim Peters en 2002 que se usa en **Python** y **Java** para ordenar con la función ***sort***; así como en la plataforma ***Android*** y en el GNU ***Octave***
- Es un método pensado para funcionar bien en **datos reales** que frecuentemente tienen algunas partes ***ya ordenadas***
- El método TimSort combina los métodos de ordenación por **mezcla** y por **inserción** pero significativamente mejorados
- El algoritmo primero busca **rachas** (trozos) de la secuencia que ya estén ordenados para ser **mezcladas** eficientemente.
- Las rachas **crecientes** o **decrecientes** de elementos se identifican con una pasada sobre la secuencia a ordenar; las decrecientes se **invierten**
- Si las rachas son demasiado pequeñas, se le van añadiendo elementos por el método de **inserción** hasta que lleguen a un ***tamaño mínimo***, que depende del tamaño inicial y está entre $32 = 2^5$ y $\mathbf{65 = 2^6 + 1}$

La pila de rachas

- Las rachas que alcancen el tamaño mínimo se van almacenando una pila hasta que se haya recorrido todo el vector a ordenar
- Una vez que se tiene la **pila de rachas**, se van mezclando las que están en la cabecera de la pila que serán rachas consecutivas
- La mezcla de rachas **consecutivas** ya ordenadas es más ***eficiente*** si las rachas están aproximadamente equilibradas en tamaño
- Para conseguir que las rachas a mezclar tengan **tamaños similares** se monitoriza el tamaño de las rachas del tope de la pila
- Se controla el tamaño de las **tres primeras rachas** de la pila, denominadas $\boldsymbol{R_1}$, $\boldsymbol{R_2}$ y $\boldsymbol{R_3}$, de forma que el tamaño de la segunda sea mayor que el de la primera ($|\boldsymbol{R_2}| > |\boldsymbol{R_1}|$) y el de la tercera sea mayor que la suma de los tamaños de las dos primeras ($|\boldsymbol{R_3}| > |\boldsymbol{R_1}| + |\boldsymbol{R_2}|$)
- Si esto ***se cumple*** se mezclan las dos primeras rachas $\boldsymbol{R_1}$ y $\boldsymbol{R_2}$
- En otro caso, la segunda racha $\boldsymbol{R_2}$ se mezcla con la más pequeña de las otras dos $\boldsymbol{R_1}$ o $\boldsymbol{R_3}$

Mezcla de TimSort

- Al mezclar dos rachas consecutivas se copia en el vector auxiliar la más pequeña de ellas y la mezcla se va colocando en la secuencia original; por la derecha o por la izquierda, según corresponda
- La mezcla de las dos rachas consecutivas R_1 y R_2 se realiza de la siguiente forma:
- Primero se aplican dos búsquedas binarias para encontrar:
    - La posición i_1 de inserción del **primer** elemento de la **segunda** racha R_2 en la **primera** racha R_1, y
    - La posición i_2 de inserción del **último** elemento de la **primera** racha R_1 en la **segunda** racha R_2
- El trozo de R_1 delante de i_1 y el trozo de R_2 detrás de i_2 no necesitan ser comparados porque están ya bien colocados
- La búsqueda binaria se sustituye por una búsqueda exponencial que busca primero entre qué dos potencias de 2 se encuentra el valor a insertar y en ese trozo se realiza la búsqueda binaria

Departamento de Ingeniería
Informática y de Sistemas
Universidad de La Laguna

ORDENACIÓN

1. **MÉTODOS CUADRÁTICOS DE ORDENACIÓN**
   - √ Inserción, Selección, por Intercambio
2. **ALGORITMOS LOGARÍTMICOS DE ORDENACIÓN**
   - √ HeapSort, QuickSort, MergeSort
3. **OTROS PROCEDIMIENTOS DE ORDENACIÓN**
   - √ Ordenación por Incrementos decrecientes; *ShellSort*
   - √ Ordenación por Radicales; *RadixSort*
   - √ Ordenación de Python y Java; *TimSort*

Universidad
de La Laguna