ORDENAMIENTO

- El concepto de conjunto ordenado de datos tiene un alto impacto en nuestra vida diaria.
- Imaginen buscar el teléfono de una persona en una guía desordenada.
- Dada la relación entre búsqueda y ordenamiento, la primera cuestión es si debería o no ordenarse un archivo.
- Una vez decidido el programador si es conveniente o no ordenar, debe definir que ordenar y que método es más conveniente utilizar.

ORDENAMIENTO: Terminología

Un ARCHIVO de tamaño n es una secuencia de n elementos r[0], r[1], r[2]...., r[n-1]

Cada elemento del archivo se denomina REGISTRO.

A cada registro r[i] esta asociada una llave denominada k[i]. Por lo general la llave es un subcampo del archivo entero.

Se dice que el archivo está ordenado de acuerdo a la llave "k", si para todos los registros del archivo se cumple que para i < j implica que k[i] < k[j]

Se llama ordenamiento INTERNO si los registros que se están ordenando están en la memoria principal, se llama EXTERNO si los registros están en un almacenamiento auxiliar.

ORDENAMIENTO

Registro 1	4	DDD
Registro 2	2	AAA
Registro 3	1	TTT
Registro 4	5	EEE
Registro 5	3	PPP

ARCHIVO ORIGINAL

1	TTT
2	AAA
3	PPP
4	DDD
5	EEE

ARCHIVO ORDENADO

Ordenamiento de registros reales.

ORDENAMIENTO

TABLA ORIGINAL DE PUNTEROS

ARCHIVO

TABLA ORDENADA DE PUNTEROS

Ordenamiento usando una tabla auxiliar de punteros.

ORDENAMIENTO: Consideraciones de Eficiencia

Existen varios métodos que pueden usarse para ordenar un archivo.

Los elementos a considerar para la decisión del método a utilizar para el ordenamiento de un archivos son: tiempo de programación, tiempo de máquina para ejecutarlo y memoria requerida para la ejecución.

Si el archivo es pequeño las técnicas utilizadas disminuyen la eficiencia en valores pequeños. Así mismo si un proceso se correrá sólo una vez no tiene sentido un gran esfuerzo en técnicas de programación.

El programador debe conocer diferentes técnicas de ordenamiento.

Método de Burbuja

Es quizás el más sencillo de entender, aunque puede ser el menos eficiente.

El método consiste en pasar a través del archivo varias veces en forma secuencial. En cada paso se compara x[i] con x[i+1] e intercambiarlos en caso de que estén desordenados.

Supongamos la sig. lista de valores: 25 57 48 37 12 92 86 33

X[0]	Con	x[1]	(25 con 57)	No intercambio
X[1]	Con	x[2]	(57 con 48)	Intercambio
X[2]	Con	x[3]	(57 con 37)	Intercambio
X[3]	Con	x[4]	(57 con 12)	Intercambio
X[4]	Con	x[5]	(57 con 92)	No intercambio
X[5]	Con	x[6]	(92 con 86)	Intercambio
X[6]	Con	x[7]	(92 con 33)	Intercambio

Método de Burbuja

Iteración 0	25	57	48	37	12	92	86	33
Iteración 1	25	48	37	12	57	86	33	92
Iteración 2	25	37	12	48	57	33	86	92
Iteración 3	25	12	37	48	33	57	86	92
Iteración 4	12	25	37	33	48	57	86	92
Iteración 5	12	25	33	37	48	57	86	92
Iteración 6	12	25	33	37	48	57	86	92
Iteración 7	12	25	33	37	48	57	86	92

Antes de codificar el método, observemos que:

- luego de la iteración 1, el número 92 quedo en su lugar
- luego de la iteración 2, el número 86 lo hizo.
- al cabo de la iteración 5, la lista estaba ordenada.

Intentemos codificar el método.

Método de Burbuja

```
Func Burbuja ( Arreglo )
Largo = len( Arreglo )
Repetir desde pasada = 0 hasta Largo-1
        pasada++
        Repetir desde j=0 hasta Largo-1
                j++
                SI Arreglo[j] > Arreglo[j+1]
                        comodin = Arreglo[j]
                        Arreglo[j] = Arreglo[j+1]
                        Arreglo[j+1] = comodin
                FIN
        FIN
FIN
```

ANALIZAMOS Y DEFINIMOS CONCEPTO O(n)

Método de Burbuja mejorado

```
Func Burbuja ( Arreglo )
Largo = len( Arreglo )
Repetir desde pasada= 0 hasta Largo-1 mientras Intercambios = TRUE
        pasada++
        Intercambios = False
        Repetir desde j=0 hasta Largo-pasada-1
                j++
                SI Arreglo[j] > Arreglo[j+1]
                        Intercambios = True
                        comodin = Arreglo[j]
                        Arreglo[j] = Arreglo[j+1]
                        Arreglo[j+1] = comodin
                FIN
        FIN
FIN
```

Método de Burbuja: comentarios

Hay Largo-1 pasadas y Largo-1 comparaciones. Entonces el total de comparaciones es (Largo-1) * (Largo-1) = $Largo^2$ – 2 Largo + 1 que es del orden de n^2

Con las consideraciones realizadas antes de iniciar podemos observar que el número de comparaciones (Largo-pasada) o sea (Largo-1) + (Largo-2) +...+ (Largo-pasada) = (2*pasada*Largo + $pasada^2$ – pasada) / 2 con lo cual si bien el multiplicador es menor que antes, se sobrecargo el tiempo de trabajo con la variable Intercambio que se inicializa y revisa una vez por pasada.
Además la función sigue siendo del orden n^2

Lo más interesante del método es que no requiere espacio adicional de almacenamiento de datos y que si la tabla está ordenada en una revisión de Largo-1 comparaciones se detecta.

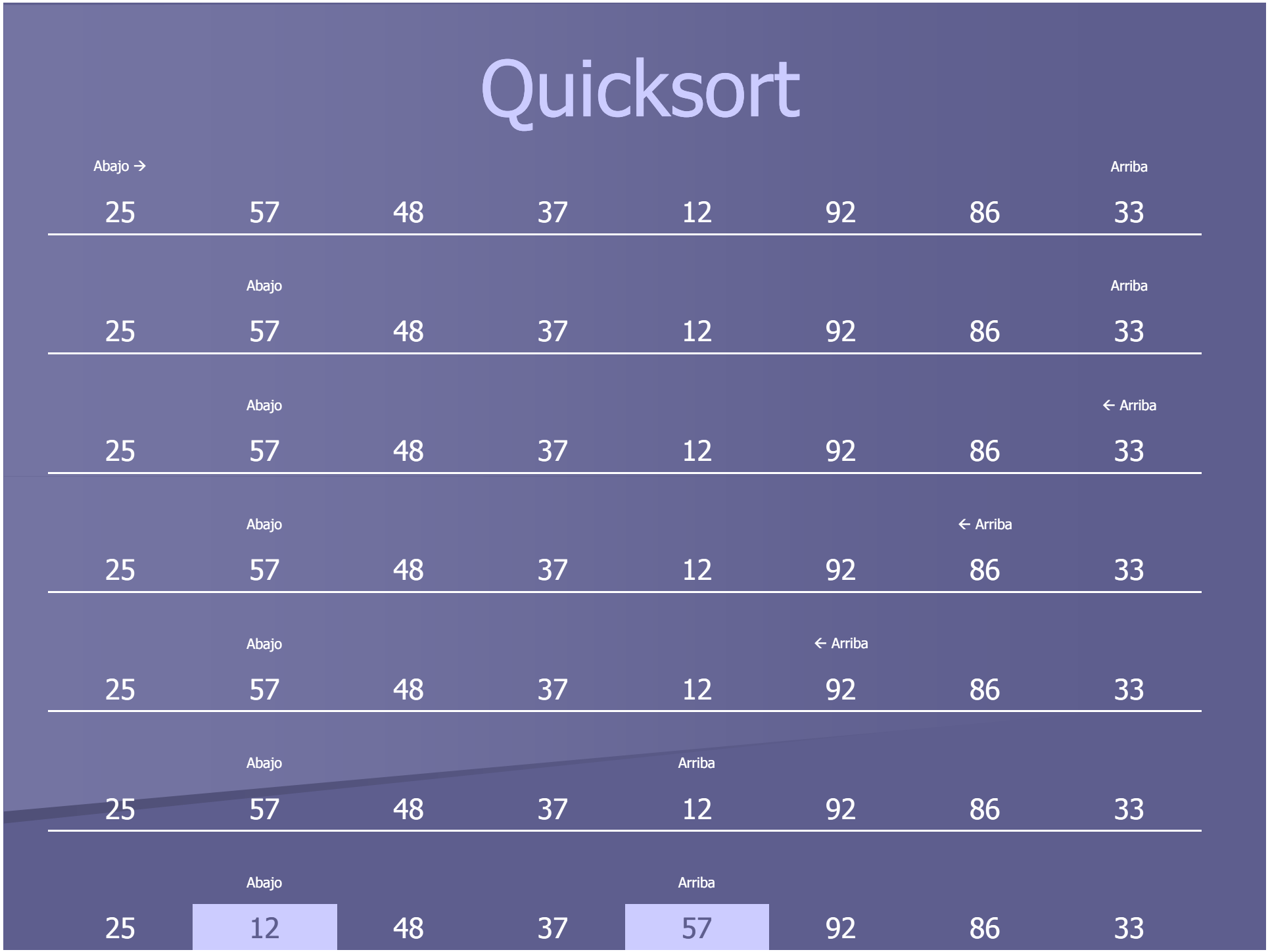
Acotación: se puede optimizar haciendo que en la misma pasada se lleven los elementos menores rápidamente hacia su posición en la parte inferior de la tabla, esto se observa en el caso del número 33.

Quicksort

Sea un Arreglo y Largo el número de elementos del arreglo que debe ser ordenado. Elegir un elemento A de una posición específica en el arreglo (ej. A = Arreglo[0]).
Suponiendo que A está en la posición j entonces:

- Cada uno de los elementos en las posiciones de 0 a j-1 es menor o igual que A
- Cada uno de los elementos en las posiciones j+1 a n-1 es mayor o igual que A

Quicksort


Abajo →
Arriba
25 57 48 37 12 92 86 33
Abajo
Arriba
25 57 48 37 12 92 86 33
Abajo
← Arriba
25 57 48 37 12 92 86 33
Abajo
← Arriba
25 57 48 37 12 92 86 33
Abajo
← Arriba
25 57 48 37 12 92 86 33
Abajo
Arriba
25 57 48 37 12 92 86 33
Abajo
Arriba
25 12 48 37 57 92 86 33

Quicksort

	Abajo →			Arriba			
25	12	48	37	57	92	86	33

		Abajo		Arriba			
25	12	48	37	57	92	86	33

		Abajo		← Arriba			
25	12	48	37	57	92	86	33

		Abajo	← Arriba				
25	12	48	37	57	92	86	33

		← Arriba - Abajo					
25	12	48	37	57	92	86	33

	Arriba	Abajo					
25	12	48	37	57	92	86	33

	Arriba	Abajo					
12	25	48	37	57	92	86	33

Quicksort

```
Func Quicksort ( Arreglo, lb, ub )

Si lb <= ub FIN

Pivote = Arreglo[ lb ]                    (se puede optimizar buscando un mejor Pivote)
Arriba = ub
Abajo = lb
Repetir mientras Abajo < Arriba
        Repetir mientras Arreglo[ abajo ] <= Pivote y abajo < ub
                Abajo ++
        Repetir mientras Arreglo[ arriba ] > Pivote
                Arriba - -
        Si Abajo < Arriba
                temp = Arreglo[ abajo ]
                Arreglo[ abajo ] = Arreglo[ arriba ]
                Arreglo[ arriba ] = temp
Arreglo[ lb ] = Arreglo[ arriba ]
Arreglo[ arriba ] = Pivote

Quicksort( Arreglo, lb, arriba-1 )
Quicksort( Arreglo, arriba+1, ub )
```

Quicksort

Esta rutina puede llamarse de manera recursiva, aunque por la cantidad de requerimiento de memoria no es conveniente para el caso de funciones de ordenamiento.

Una técnica interesante para optimizar el proceso es hacer una recorrida por el arreglo calculando el valor promedio del arreglo, e ingresando este valor como elemento pivot para la función de ordenamiento.

Supongamos que tenemos un archivo de tamaño $n = 2^m$. Si dividimos el archivo justo en mitades cada vez que encontramos el pivot, entonces haremos en la primer recorrida n comparaciones, luego haremos 2 veces n/2 comparaciones y luego para 4 subarchivos n/4 comparaciones. Así en cada recorrida se realizaran n comparaciones tantas veces como m.

$$n + 2 * (n / 2) + 4 * (n / 4) + 8 * (n / 8) .....$$
$$n + n + n + n..... (m \text{ veces})$$

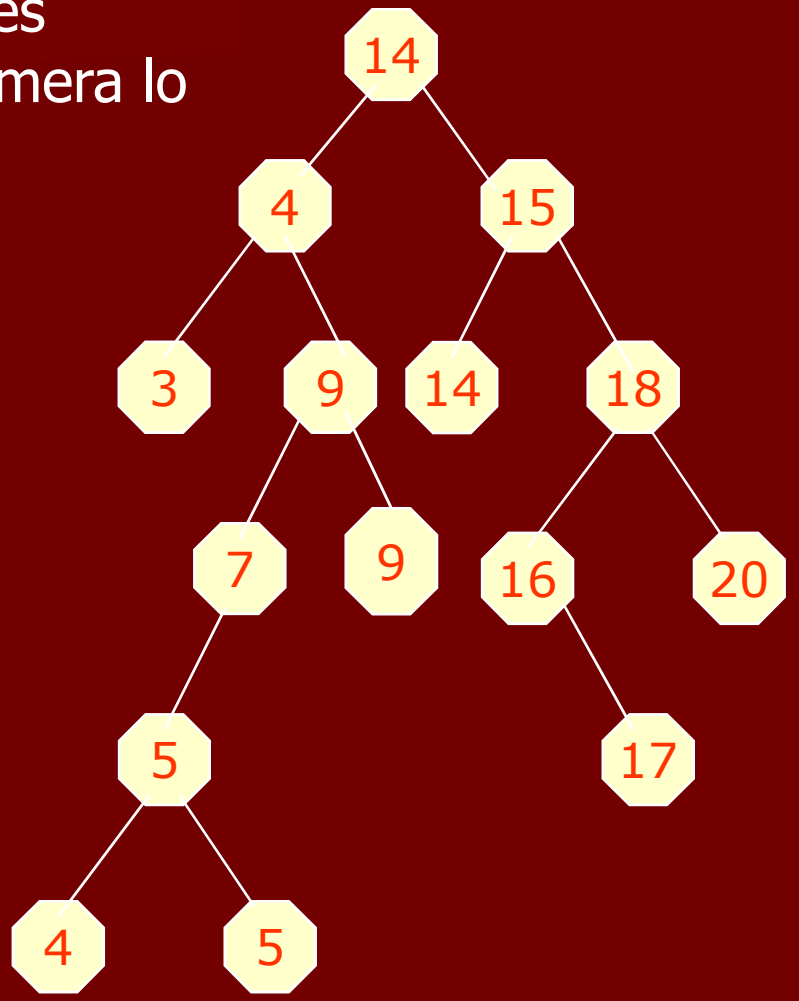
De manera que la función es de orden $n * m$, y como $m = \log_2 n$, se concluye que la función QUICKSORT es de orden $n \log_2 n$. Lo que la hace muy eficiente.

ALICACIONES: Árboles Binarios

Muchas aplicaciones que utilizan árboles binarios proceden en 2 fases, en la primera lo construyen en la segunda lo recorren.

Supongamos que dada una lista de números quiero imprimirlos en orden ascendente.

Al ingresar los datos se cargan en un árbol binario, donde a la izquierda se ponen los números menores a la raíz, y a la derecha se ponen los números mayores o iguales a la raíz.



Datos Ingresados: 14 15 4 9 7 18 3 5 16 4 20 17 9 14 5

Ordenamiento por árboles binarios

Para medir la eficiencia debemos considerar el caso en que los datos ingresen ordenados, en ese caso se realizarán las siguientes comparaciones:

$$2 + 3 + ... + n = n * (n + 1) / 2 - 1$$

O sea un proceso de orden n^2

En caso de que el árbol este con los datos balanceados, (caso óptimo) las comparaciones son del orden n log n.

Además debemos considerar para recorrer el árbol la necesidad de punteros y luego la necesidad de una pila para almacenamiento del orden de recorrido que se lleva del árbol.

Shell sort (revisar en JOYANES)

Es un método de ordenamiento por disminución del incremento, este método ordena subarchivos separados del archivo original. Se parte de un valor de incremento k, que es el paso que se da entre los elementos, entonces el método ordenará k subarchivos donde cada subarchivo contiene n/k elementos a ordenar.

Luego de ordenar los k subarchivos por inserción simple, se vuelve a elegir un valor menor de k, se particiona el archivo en k subarchivos hasta que k toma el valor 1, que es la última acción.

Sigamos por ejemplo con nuestro archivo:

25 57 48 37 12 92 86 33

Shell sort

Supongamos k = 5 entonces los subarchivos son:

x[0], x[5] - x[1], x[6] - x[2], x[7] - x[3] - x[4]

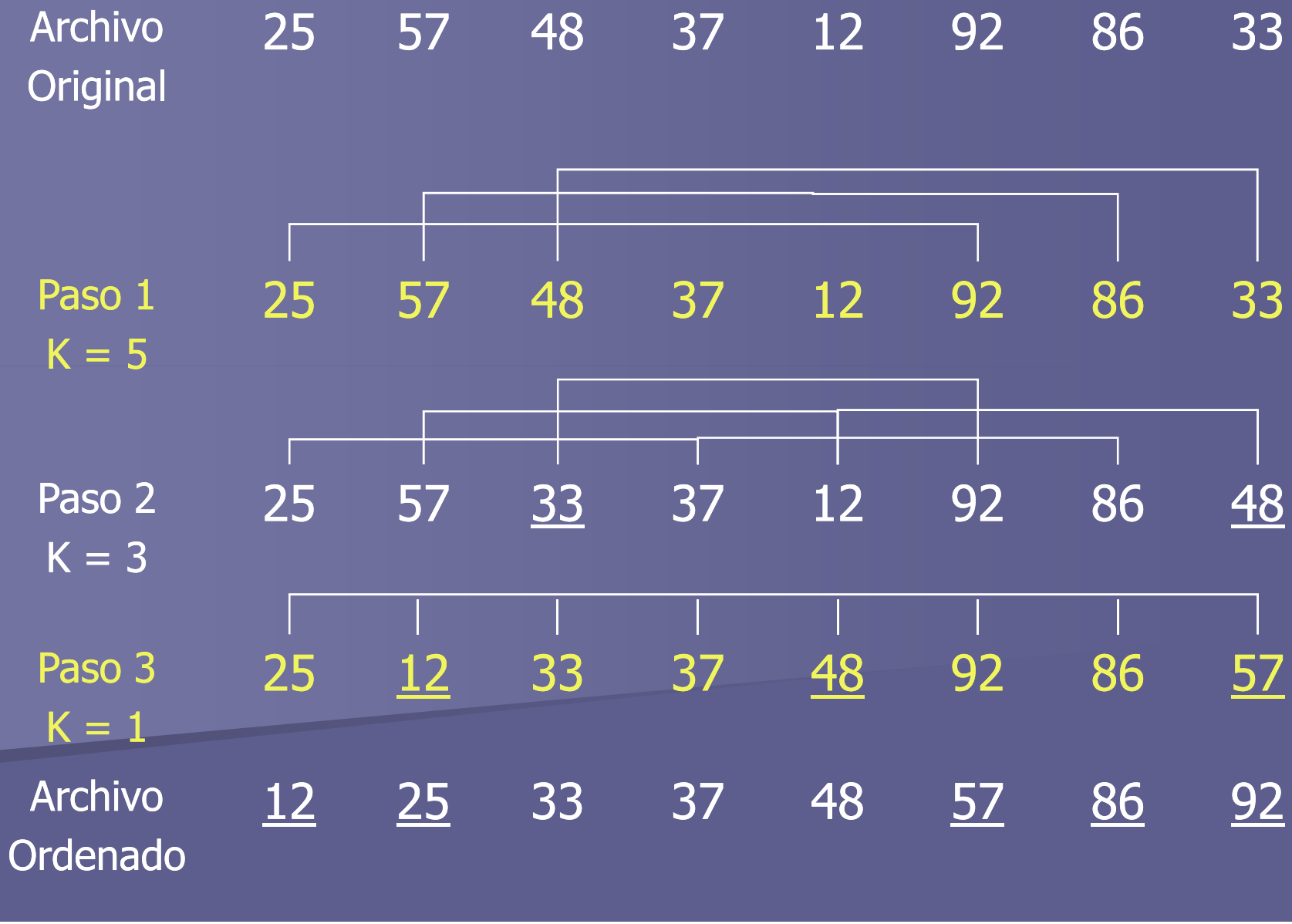
Luego en la segunda iteración k = 3 entonces:

x[0], x[3], x[6] - x[1], x[4], x[7] - x[2], x[5]

Por último si selecciona k = 1, se debe ordenar el siguiente archivo:

X[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7]

Shell sort


Archivo Original
25 57 48 37 12 92 86 33
Paso 1
K = 5
25 57 48 37 12 92 86 33
Paso 2
K = 3
25 57 33 37 12 92 86 48
Paso 3
K = 1
25 12 33 37 48 92 86 57
Archivo Ordenado
12 25 33 37 48 57 86 92

Shell sort

```
Shellsort ( Arreglo, largo, incrementosdecrecientes, numincr )

paso = 0
Repetir mientras paso < numincr
        k = incrementosdecrecientes[ paso ]
        j = k
        Repetir j < largo
                a_ubicar = x[ j ]
                temp = j – k
                Repetir mientras temp >= 0 y a_ubicar < x[ temp ]
                        x[ temp + k ] = x[ temp ]
                        temp -= k
                x[ temp + k ] = a_ubicar
                j ++
        incre ++
```

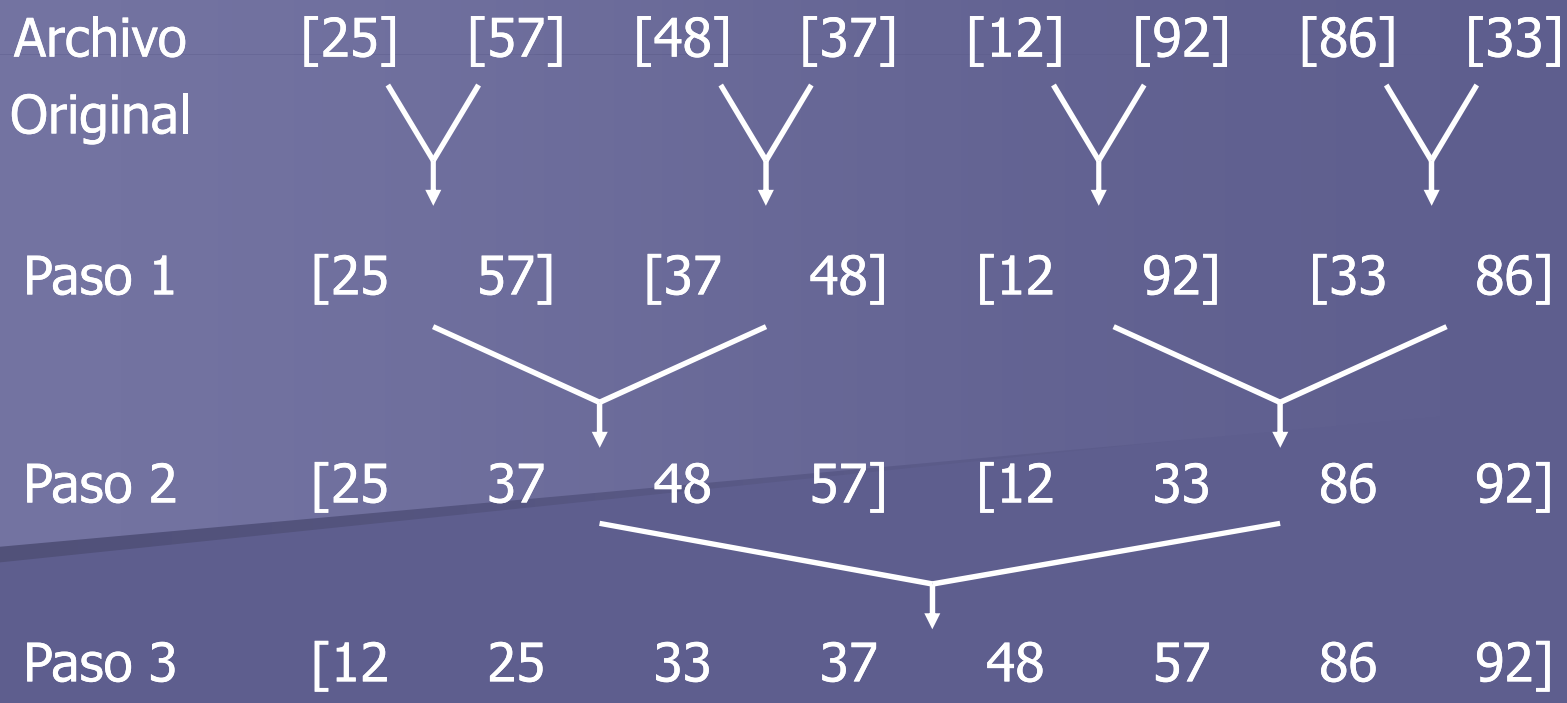
Observar que si k = 1 entonces el ordenamiento es por inserción simple.

Shell sort

- La idea de Shell Sort es muy simple.
- El ordenamiento por inserción simple es muy eficiente cuando el archivo está casi ordenado.
- Además cuando el archivo es pequeño a veces es más eficiente un ordenamiento del tipo n^2 que un ordenamiento del tipo n log n.
- Como el primer incremento de Shell Sort es grande, los subarchivos son pequeños, de manera que el ordenamiento por inserción simple es bastante rápido.
- A medida que los incrementos decrecen el archivo se encuentra más ordenado y entonces los subarchivos que son grandes están ordenados y por ende el método de ordenamiento también es muy eficiente.
- El cálculo matemático es complejo, pero se ha demostrado que Shell Sort se aproxima al orden n $(\log n)^2$, en general el método de Shell Sort, se recomienda para archivos de tamaño moderado del orden de los mil elementos.

Ordenamiento por Intercalación

- Intercalación es el proceso de combinar 2 o más archivos ordenados en un tercer archivo ordenado.
- La idea que se nos ocurre consiste en dividir el archivo tomando elementos de a pares adyacentes, entonces tenemos n/2 archivos de tamaño 2.



Ordenamiento por Intercalación

- El ordenamiento por intercalación no hay más de n log2 n pasos, y cada uno implica n o menos comparaciones. Así este método requiere no más de n log2 n comparaciones.
- Esto se asemeja bastante al método de quick sort, lo que pasa es que requiere muchas más intercalaciones de cambio de variables.
- Además se requiere más espacio de memoria para el armado de los subarchivos de trabajo.
- Ambos métodos implican la división de los archivos en partes, luego el ordenamiento de ambos y luego la unificación de ambos archivos.