

Búsqueda y Ordenación

Contenido del Tema

1.1. Introducción

1.2. Búsqueda

1.2.1. Búsqueda Secuencial

1.2.2. Búsqueda Binaria

1.2.3. Búsqueda en Cadenas

1.3. Ordenación

1.3.1. Ordenación por Inserción

1.3.2. Ordenación por Selección

1.3.3. Ordenación por Intercambio



Introducción

Objetivos

- Uso de las estructuras de datos antes
- Estructura de datos seleccionada ➡ Algoritmo diseñado
- Tipos de Algoritmos ➡

Interno / Externo



Búsqueda

- Operación frecuente en Programación
 - ➡ Diversidad de Algoritmos
- Diferentes Técnicas de Búsqueda
 - ➡ Búsqueda en Listas: Algoritmos, Eficiencia

Lista ➡ Elementos componentes: Tipo de Datos Simple

Vector = **ARRAY**[0..N-1]**DE** TipoElemento

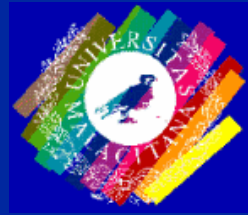


Búsqueda Secuencial

Aplicabilidad:

- Desconocimiento acerca de la organización de los datos
- Estructura solo accedida
- Visitar todas las posiciones del
encuentre el elemento o se
llegue al final del mismo
(elemento no esta)

```
PROC Secuencial(↓V:Vector;  
                ↓x:TipoElemento)  
VAR ind:NATURAL  
Inicio  
    ind ← 0  
    MIENTRAS (ind<N) Û (V[ind]  
                        <> x) HACER  
  
        ind ← ind+1  
  
    FINMIENTRAS  
    SI ind = N ENTONCES  
        escribir("no  
                encontrado")  
  
    EN OTRO CASO  
        escribir("encontrado")  
  
    FINSI  
Fin
```



Búsqueda Secuencial

Consideraciones

- La expresión lógica sólo es correcta si el segundo término sólo se evalúa cuando el primero es TRUE ()
- - Mejor Caso \Rightarrow 1 comparación
 - Peor Caso \Rightarrow N comparaciones
 - Caso Promedio $\Rightarrow \sim N/2$ comparaciones
- Fin de Búsqueda:
 - Elemento hallado $\Rightarrow V[ind]=x$
 - Elemento no hallado $\Rightarrow ind=N$



Búsqueda Secuencial con Centinela

- Posible **optimización** del algoritmo anterior:

Eliminar el chequeo

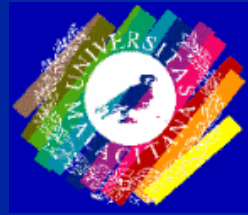
$(ind < N)$

- Asegurarnos que x está en V **¿Como?**

Añadiendo x al final del array (**centinela**)

- Vector=**ARRAY**[0..N]**DE** TipoElemento

```
PROC secuencial_Op( ↓V:Vector;  
                    ↓x:TipoElemento)  
VAR    ind:NATURAL  
Inicio  
    V[N] ← x  
    ind ← 0  
    MIENTRAS (V[ind] <> x) HACER  
        ind ← ind+1  
    FINMIENTRAS  
    SI ind = N ENTONCES  
        escribir("no  
                encontrado")  
    EN OTRO CASO  
        escribir("encontrado")  
    FINSI  
Fin
```



Búsqueda Binaria

- **Aplicabilidad:**

- Información adicional: **Cómo están organizados los datos.**

- Búsqueda más eficiente \Rightarrow Datos Ordenados.

- $\forall k$ tal que $1 \leq k \leq N-1$, se cumple que $V[k-1] \leq V[k]$

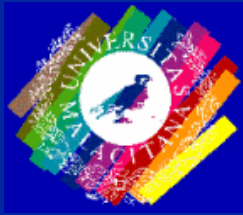
- **Idea Clave:**

Inspeccionar un elemento de índice **m** elegido al azar (x elemento a buscar):

- Si $V[m]=x \Rightarrow$ Fin Búsqueda

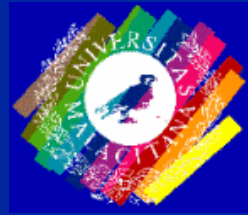
- Si $V[m]<x \Rightarrow \forall k/ k \leq m, V[k]$ eliminados

- Si $V[m]>x \Rightarrow \forall k/ k \geq m, V[k]$ eliminados



Búsqueda Binaria

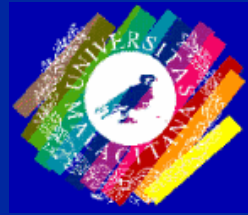
```
PROC Binaria ( $\downarrow$ V:Vector;  $\downarrow$ x:TipoElemento)
VAR
    izq,der,m:NATURAL;    encontrado:LOGICO
Inicio
    Izq  $\leftarrow$  0
    Der  $\leftarrow$  N - 1
    Encontrado  $\leftarrow$  FALSO
    MIENTRAS (Izq  $\leq$  Der)  $\dot{\cup}$  ( $\emptyset$  Encontrado) HACER
        m  $\leftarrow$  (*cualquier valor entre Izq y Der*)
        SI V[m] = x ENTONCES
            Encontrado  $\leftarrow$  TRUE
        EN OTRO CASO
            SI V[m] < x ENTONCES
                Izq  $\leftarrow$  m + 1
            EN OTRO CASO
                Der  $\leftarrow$  m - 1
        FINSI
    FINSI
    FINMIENTRAS
Fin
```

Búsqueda Binaria

Consideraciones ➡ Elección de m

- No afecta a la **corrección** del Algoritmo
- **Objetivo:** Eliminar el mayor número de elementos en cada iteración
- Elección Optima ➡ $m \leftarrow (\text{Izq} + \text{der})/2$
- Eficiencia(**Peor Caso**) ➡ $\text{Trunc}(\log_2 N) + 1$



Búsqueda en Cadenas

- **Objetivo:** Localizar la presencia de una cadena de longitud M dentro de otra de



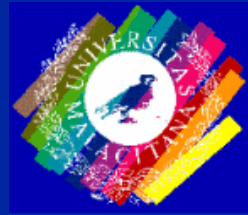
$\leq N$

- **Tipos:**

TipoElemento=**CARACTER**

Texto=**ARRAY**[0..N-1]**DE** TipoElemento

Patrón=**ARRAY**[0..M-1]**DE** TipoElemento



Búsqueda Directa Hacia Delante

Algoritmo de búsqueda en cadenas

- **Idea Clave:** Comparar carácter a carácter texto y patrón comenzando por el extremo izquierdo de ambos

¿Coinciden ?

- **Si** ➡ se compara el siguiente carácter
- **No** ➡ el proceso se reinicia comenzado en la posición siguiente a la que se inició la

```
FUNC BusqCadena(↓P:Patron;  
                ↓T:texto):ENTERO  
VAR    i,j:NATURAL  
        res:ENTERO  
Inicio  
    i ← 0; j ← 0  
    MIENTRAS(i<N) Y (j<M) HACER  
        SI T[i] = P[j] ENTONCES  
            i ← i + 1  
            j ← j + 1  
        EN OTRO CASO  
            i ← i - j + 1  
            j ← 0  
        FINSI  
    FINMIENTRAS  
    SI j = M ENTONCES  
        Res ← i - M  
    EN OTRO CASO  
        Res ← -1  
    FINSI  
    RESULTADO ← Res  
Fin
```



Búsqueda Directa Hacia Atrás

Algoritmo de búsqueda en cadenas

- **Idea Clave:** Comparar carácter a carácter texto y patrón comenzando por el extremo derecho de ambos

¿Coinciden ?

- **Si** ➡ se compara el carácter anterior
- **No** ➡ el proceso se reinicia comenzado en la anterior posición a la que se inició la

El diseño del Algoritmo se deja propuesto al alumno como ejercicio



-

- **Objetivo:** Mejorar Eficiencia ➡ 1 iteración



Hashing

Opciones:

- Relación perfecta entre el **registro** ➡ y la posición de un registro ➡ Impracticable en muchas ocasiones
- Ordenar los elementos aplicando alguna ➡

Usos de la Función Hash:

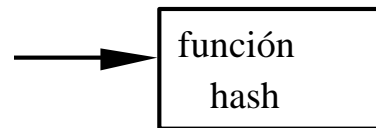
- El resultado de la misma se usa para la
- Se utiliza como método de acceso (



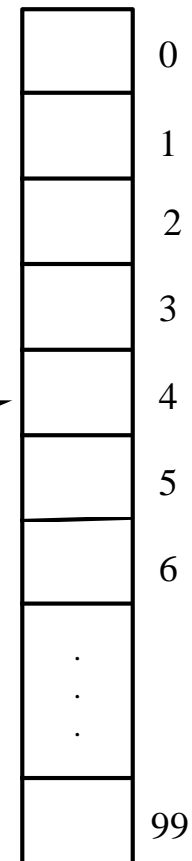
Hashing

Función Hash \leftarrow clave MOD 100

clave
459250704



clave MOD 100





Hashing

- Una Función Hash no garantiza direcciones únicas
→ **Colisiones**
- **Sinónimos:** valores clave que producen colisiones al aplicarle una Función

“Colisiones difíciles de evitar”



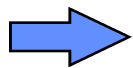
Algoritmos de Manejo de Colisiones
Esquema Almacenamiento = Búsqueda



Algoritmos de manejo de Colisiones

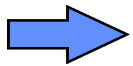
Hash y Búsqueda

- **Almacenamiento:** Almacenar el elemento colisionado en el siguiente espacio libre



Estructura Circular

- : Aplicar la función

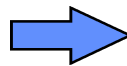


Obtenemos una posición

Comparación de claves,

Si 

Fin. Elemento hallado

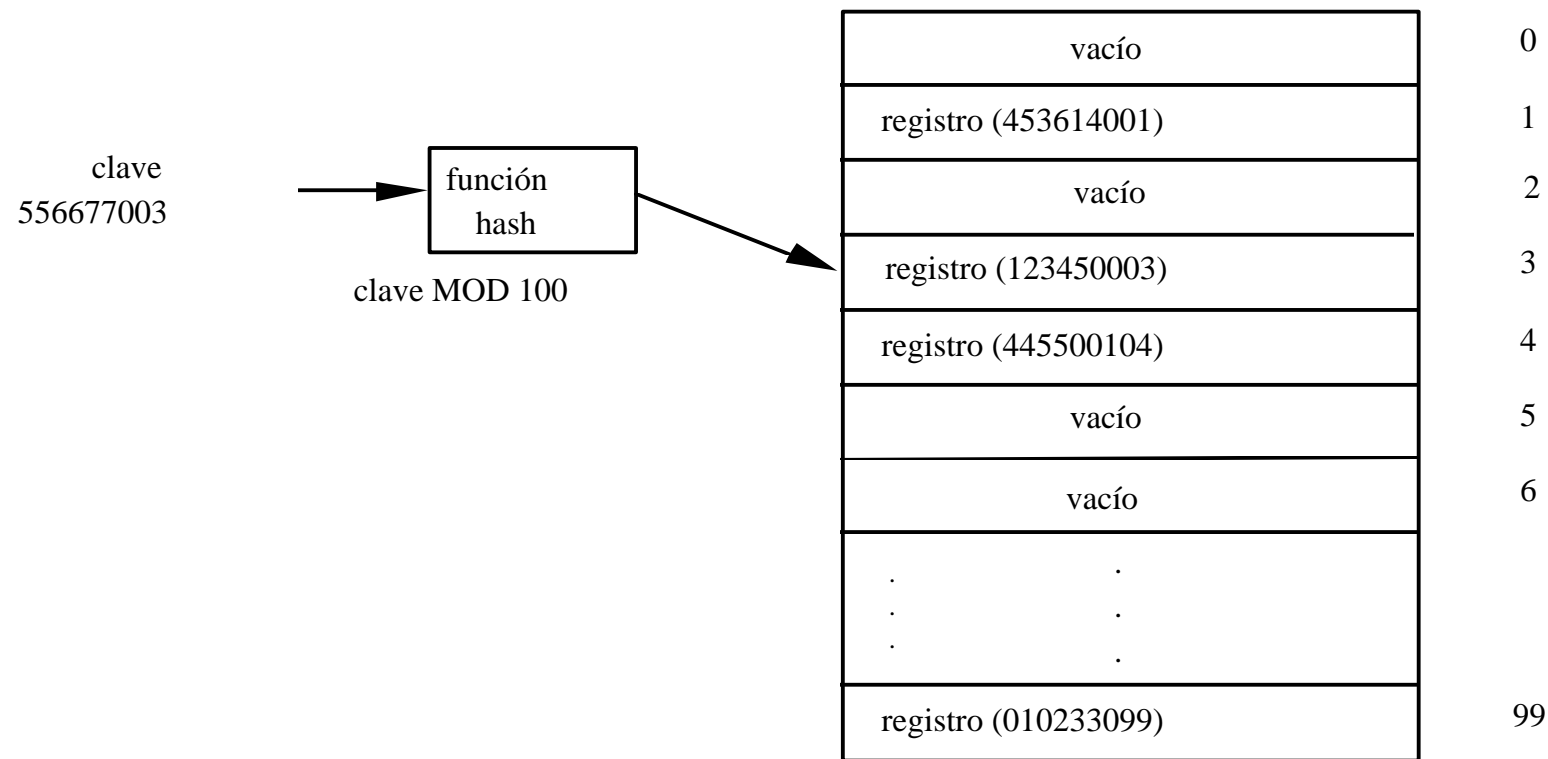


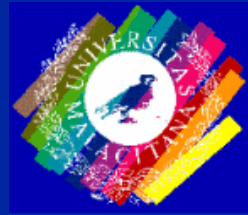
Búsqueda Secuencial



Algoritmos de Manejo de Colisiones

Hash y Búsqueda





Algoritmos de Manejo de Colisiones

Hash y Búsqueda

```
PROC Almacenamiento(↓↑ ArrayEmpleados:TipoArray;  
                    ↓ NuevoValor:TipoReg; ↓↑ LugarEncontrado:LOGICO);  
  
VAR  
    LugarInicio, IntentarLugar : ENTERO  
  
Inicio  
    LugarInicio ← NuevoValor.clave MOD 100  
    IntentarLugar ← LugarInicio; LugarEncontrado ← FALSO  
  
REPETIR  
    SI (ArrayEmpleados[IntentarLugar]=Vacío) ENTONCES  
        ArrayEmpleados[IntentarLugar] ← NuevoValor  
        LugarEncontrado ← CIERTO  
    EN OTRO CASO  
        IntentarLugar ← (IntentarLugar+1) MOD 100  
    FINSI  
    HASTA QUE LugarEncontrado Ú (IntentarLugar=LugarInicio)  
  
Fin
```



Algoritmos de Manejo de Colisiones

Hash y Búsqueda

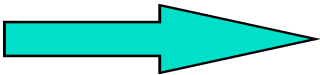
TipoReg = **REGISTRO**

clave:**Z**,....

FINREGISTRO

TipoArray=**ARRAY** [0..99] **DE** TipoReg

Función Hash

- Ejemplo 
- Campo clave: cadena de caracteres

$(S_i \text{ ORD}(\text{clave}[i])) \text{ MOD tamaño}$

FUNC ValorHash(\downarrow clave:Cadena)
: **ENTERO**

VAR

i,long:**NATURAL**

valor:**ENTERO**

Inicio

long \leftarrow longitud(clave)

valor \leftarrow 0

PARA i \leftarrow 1 **HASTA** long **HACER**
 valor \leftarrow valor+ORD(clave[i])

FINPARA

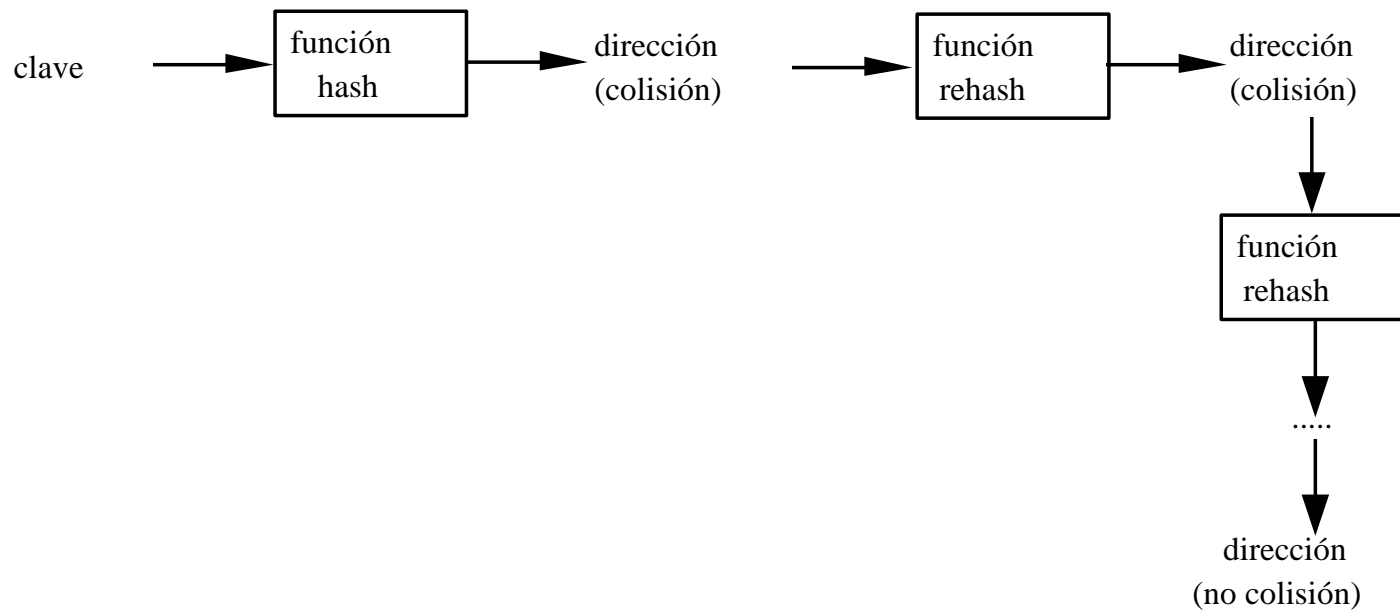
RESULTADO \leftarrow valor MOD 100

Fin



Algoritmos de Manejo de Colisiones

Rehashing

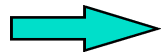




Algoritmos de Manejo de Colisiones

Rehashing

- **Almacenamiento**: Se usa la dirección que produjo la colisión como entrada a otra función



El proceso se repite hasta que **no**

-

:

-

: Idéntico proceso pero invertido



Algoritmos de Manejo de Colisiones

Cubos y Encadenamientos

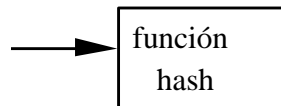
- **Cubos:** Cada dirección transformada contiene espacio para múltiples registros ➡ Cubo
: Cubo lleno ➡ Colisiones
- **Encadenamientos:** Cada dirección transformada se usa como índice de una posición del nos permite acceder a una cadena de registros
: Recorrido secuencial de la Cadena



Algoritmos de Manejo de Colisiones

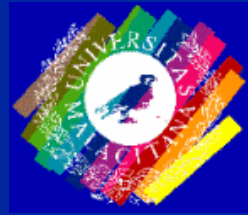
Cubos

clave
556677003



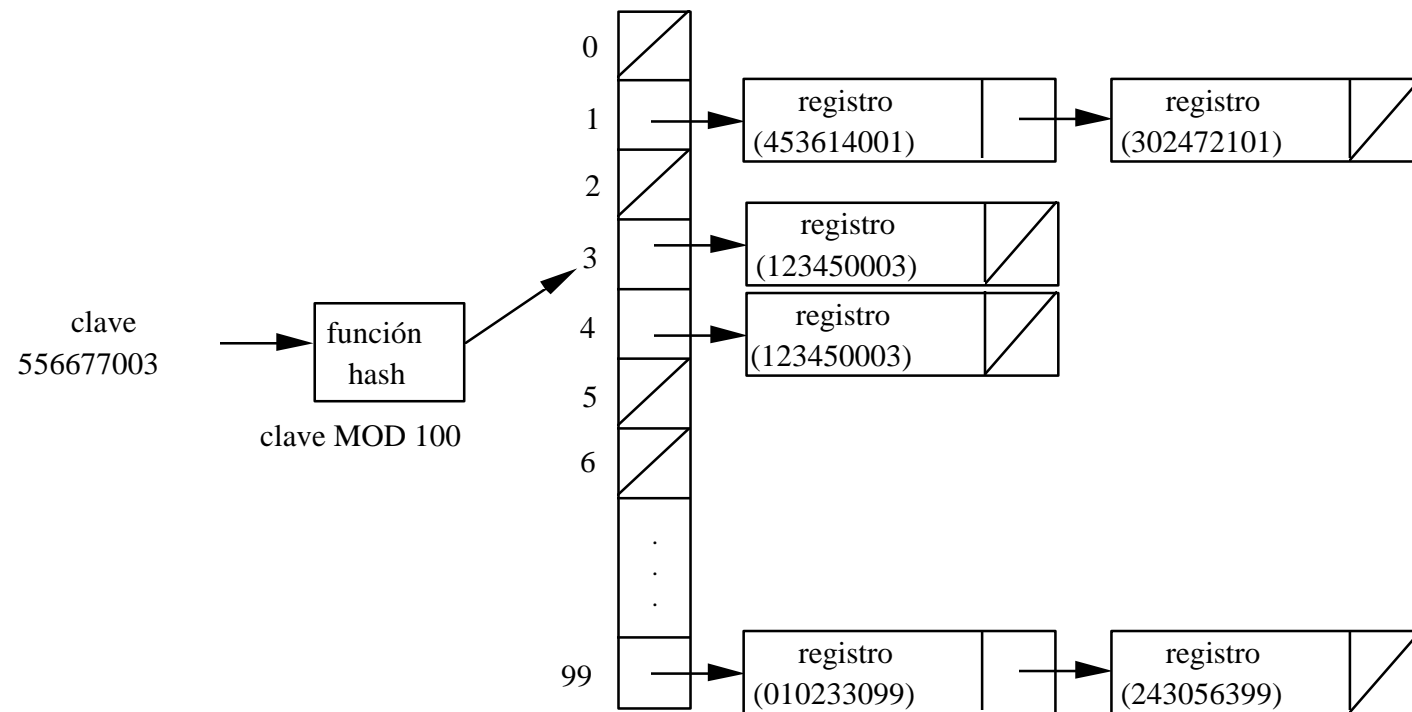
clave MOD 100

vacío	vacío	vacío	0
registro (453614001)	registro (302472101)	vacío	1
vacío	vacío	vacío	2
registro (123450003)	vacío	vacío	3
registro (445500104)	vacío	vacío	4
vacío	vacío	vacío	5
vacío	vacío	vacío	6
.	.	.	
.	.	.	
.	.	.	
registro (010233099)	registro (243056399)	vacío	99



Algoritmos de Manejo de Colisiones

Encadenamientos





Elección de una buena Función Hash

- **Objetivos de diseño:**

Minimizar Colisiones ➡ Distribuir de manera uniforme los elementos a lo largo de la estructura.

- **¿Como?**

Usar estructuras de datos con más espacio del
20%)

Conocimiento sobre la **distribución, dominio y**



Ordenación

- Actividad **esencial** y **muy relevante** en Programación → Ocupa más del **25%** del Tiempo de computación
- Problema **ampliamente estudiado** → Diversidad de Algoritmos
- → Buscar el mejor algoritmo de Ordenación



Ordenación

Eficiencia

- Algoritmos que **economicen** la memoria disponible
- Algoritmo más **eficiente**: Quickshort $\Rightarrow n \cdot \log(n)$
- Algoritmos **directos**: Inserción, Selección e Intercambio $\Rightarrow n^2$
 - Menos eficientes
 - Adecuados para dilucidar las principales características de los algoritmos de Ordenación

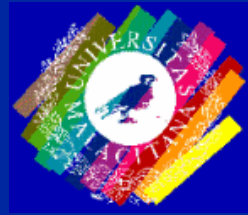


Ordenación

Algoritmos de Ordenación

- Tipo:
Indice=[1..N]
Vector=**ARRAY** Indice **DE** TipoElemento
- TipoElemento: Tipo sobre el que hay definida una
- El problema de ordenación es encontrar una permutación s , tal que si V es una variable del tipo Vector :
$$V[s_i] \leq V[s_{i+1}], 1 \leq i \leq N-1$$

El orden deseado será: $V[s_1], V[s_2], \dots, V[s_N]$.



Ordenación por Inserción

- **Idea Clave:** para cada paso i , los elementos V_1, \dots, V_{i-1} están ordenados y se inserta entre ellos V_i de forma que después de la inserción los V_1, \dots, V_i estén ordenados.
- **Ejemplo:** Se ha de ordenar la siguiente colección

3 1 9 7 5 23 15 20



Ordenación por Inserción

Paso1:

$i:=2$. Suponemos V_1, \dots, V_1 ordenados,
insertamos $V_2 \Rightarrow V_1, \dots, V_2$ ordenados.

1 3 9 7 5 23 15 20

$1, \dots, V_2$ ordenados,
insertamos $V_3 \Rightarrow V_1, \dots, V_3$ ordenados.

1 3 9 7 5 23 15 20

$1, \dots, V_3$ ordenados,
insertamos $V_4 \Rightarrow V_1, \dots, V_4$ ordenados.

1 3 7 9 5 23 15 20

$1, \dots, V_4$ ordenados,
insertamos $V_5 \Rightarrow V_1, \dots, V_5$ ordenados.

1 3 5 7 9 23 15 20

$1, \dots, V_5$ ordenados,
insertamos $V_6 \Rightarrow V_1, \dots, V_6$ ordenados.

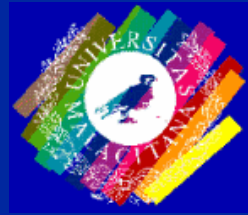
1 3 5 7 9 23 15 20

$1, \dots, V_6$ ordenados,
insertamos $V_7 \Rightarrow V_1, \dots, V_7$ ordenados.

1 3 5 7 9 15 23 20

$1, \dots, V_7$ ordenados,
insertamos $V_8 \Rightarrow V_1, \dots, V_8$ ordenados.

1 3 5 7 9 15 20 23



Ordenación por Inserción

¿Cómo realizar la inserción?

- Inserción directa → Abrir un hueco en la $1, \dots, V_{i-1}$ para encajar V_i

Ejemplo:

Paso4:

$i:=5$. Suponemos V_1, \dots, V_4 ordenados, insertamos $V_5 \Rightarrow V_1, \dots, V_5$ ordenados.

1	3	7	9	5	23	15	20
1	3	5	7	9	23	15	20



Ordenación por Inserción

```
FUNC buscar_posicion (↓valor:  
    TipoElemento; ↓ V:Vector;  
    ↓ fin: Indice): Indice
```

```
VAR
```

```
    i:Indice
```

```
Inicio
```

```
    i ← 1
```

```
    MIENTRAS (i ≤ fin) Û  
        (V[i] < valor) HACER
```

```
        i ← i+1
```

```
    FINMIENTRAS
```

```
    RESULTADO ← i
```

```
Fin
```

```
PROC abrir_hueco (↓↑V:Vector;  
    ↓ inicio, ↓ fin:Indice)
```

```
VAR
```

```
    i:Indice
```

```
Inicio
```

```
    PARA i←fin HASTA inicio+1  
        (PASO -1) HACER
```

```
        V[i] ← V[i-1]
```

```
    FINPARA
```

```
Fin
```



Ordenación por Inserción

PROC Insercion ($\downarrow\uparrow$ V:Vector)

VAR

i, pos:Indice

aux:TipoElemento

Inicio

PARA i \leftarrow 2 **HASTA** N **HACER**

aux \leftarrow V[i]

pos \leftarrow buscar_posicion
(aux, V, i-1)

abrir_hueco(V, pos, i)

V[pos] \leftarrow aux

FINPARA

Fin

Una Forma más

RAPIDA

de realizar la inserción



Búsqueda Binaria

posición inserción valor

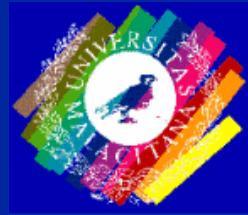
Mejora \rightarrow Número de
Comparaciones



Ordenación por Inserción

```
FUNC buscar_posicion  
    (↓↑valor:TipoElemento;  
     ↓V:Vector; ↓fin: Indice)  
        :Indice  
  
VAR  
    izd, der, med:Indice  
Inicio  
    izq ← 1  
    der ← fin  
    MIENTRAS izd <= der HACER  
        med ← (izq + der) / 2
```

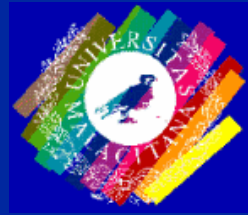
```
        SI valor < V[med]  
            ENTONCES  
                der ← med - 1  
        EN OTRO CASO  
            izq ← med + 1  
        FINSI  
    FINMIENTRAS  
    RESULTADO ← izq  
Fin
```



Ordenación por Selección

- Idea Clave:

- 1) Determinar la **posición** del **menor** elemento del array
- 2) **Intercambiar** dicho elemento por el elemento que hay en la **primera** posición V_1
- 3) **Repetir** esta operación con los $N-1$ elementos restantes V_2, \dots, V_N



Ordenación por Selección

Ejemplo

320 96 16 90 120 80 200 64

320 96 16 90 120 80 200 64
↑ ↑

16 96 320 90 120 80 200 64
 ↑ ↑

16 64 320 90 120 80 200 96
 ↑ ↑

16 64 80 90 120 320 200 96
 ↑

Paso5:

16 64 80 90 120 320 200 96
 ↑ ↑

16 64 80 90 96 320 200 120
 ↑ ↑

Paso7:

16 64 80 90 96 120 200 320
 ↑

Resultado

16 64 80 90 96 120 200 320

N elementos → N-1 Intercambios



Ordenación por Selección

```
FUNC posicion_menor_elem  
  (↓V:Vector;  
   ↓inicio:Indice):Indice  
VAR  
  pos_menor, i:Indice  
Inicio  
  pos_menor ← inicio  
  PARA i ← inicio+1 HASTA N  
    HACER  
      SI V[i] < V[pos_menor]  
        ENTONCES  
          pos_menor ← i  
      FINSI  
  FINPARA
```

```
  RESULTADO ← pos_menor  
Fin  
  
PROC Intercambiar (↓↑ x, y:  
                  TipoElemento)  
VAR  
  aux: TipoElemento  
Inicio  
  aux ← x  
  x ← y  
  y ← aux  
Fin
```



Ordenación por Selección

```
PROC Seleccion(↓↑ v:Vector)
VAR
    i:Indice
Inicio
    PARA i ← 1 HASTA N-1 HACER
        subir_menor_seleccion(V,i)
    FINPARA
Fin

PROC subir_menor_seleccion
    (↓↑ v:Vector;
    ↓ posicion:Indice)
```

```
VAR
    pos_menor:Indice
Inicio
    pos_menor ←
        posicion_menor_elem(V,
                            posicion+1)

    SI V[posicion] >
        V[pos_menor] ENTONCES
        intercambiar(V[posicion],
                    V[pos_menor])
    FINSI
Fin
```



Ordenación por Intercambio

- **Idea Clave:** Comparar pares de elementos
e

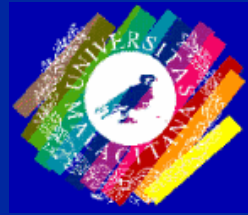


- 1) Comparar V_N y V_{N-1} , si no están ordenados, intercambiarlos
- 2) Comparar V_{N-1} y V_{N-2} , repitiendo el proceso
- 3) El proceso continua hasta que cada elemento del haya sido comparado con sus elementos



Ordenación por Intercambio

- En el **primer recorrido** el elemento más **pequeño** del array sube posición a posición hasta ocupar la
→
- **segundo recorrido** el segundo elemento mayor llegará a la
- $N-1$ Recorridos, $N-i$ Comparaciones, $N-i$ Intercambios como máximo (recorrido $i-$)



Ordenación por Intercambio

Ejemplo

72	64	50	23	85	18	37	99	45	8
Paso 1:									
72	72	72	72	72	72	72	72	72	8
64	64	64	64	64	64	64	64	8	72
50	50	50	50	50	50	50	8	64	64
23	23	23	23	23	23	23	8	50	50
85	85	85	85	85	8	23	23	23	23
18	18	18	18	18	8	85	85	85	85
37	37	37	8	18	18	128	18	18	18
99	99	8	37	37	37	37	37	37	37
45	8	99	99	99	99	99	99	99	99
8	45	45	45	45	45	45	45	45	45



Ordenación por Intercambio

Paso2:

8	8	8	8	8	8	8	8	8
72	72	72	72	72	72	72	72	18
64	64	64	64	64	64	64	18	72
50	50	50	50	50	50	18	64	64
23	23	23	23	23	18	50	50	50
85	85	85	85	18	23	23	23	23
18	18	18	18	85	85	85	85	85
37	37	37	37	37	37	37	37	37
99	45	45	45	45	45	45	45	45
45	99	99	99	99	99	99	99	99

Paso3,....., Paso9:

8 18 23 37 45 50 64 72 85 99



Ordenación por Intercambio

PROC Intercambio ($\downarrow\uparrow$ V:Vector)

VAR

i:Indice

Inicio

PARA i \leftarrow 1 **HASTA** N-1 **HACER**

subir_menor_burbuja(V,i)

FINPARA

Fin

PROC subir_menor_burbuja(
 $\downarrow\uparrow$ V:Vector;
 \downarrow posicion:Indice)

VAR

i:Indice

Inicio

PARA i \leftarrow N **HASTA** posicion+1
(PASO -1) **HACER**

SI V[i-1] > V[i]

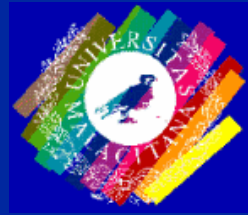
ENTONCES

intercambiar(
V[i-1], V[i])

FINSI

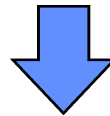
FINPARA

Fin



Ordenación por Intercambio

Mejora



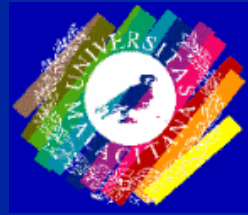
- Eliminar recorridos innecesarios
- Elementos ya ordenados → en un recorrido no se ha hecho ningún intercambio



Ordenación por Intercambio

```
PROC Intercambio_2 (  $\downarrow\uparrow$   
                    V:Vector)  
  
VAR  
    i:Indice  
    intercambio:LOGICO  
    aux:TipoElemento  
Inicio  
    intercambio  $\leftarrow$  CIERTO  
    i  $\leftarrow$  1  
    MIENTRAS i  $\leq$  N-1  $\hat{U}$   
        intercambio HACER  
            intercambio  $\leftarrow$  FALSO
```

```
        PARA j  $\leftarrow$  1 HASTA N-i  
            HACER  
                SI V[j]>V[j+1] ENTONCES  
                    intercambio $\leftarrow$ CIERTO  
                    aux  $\leftarrow$  V[j]  
                    V[j]  $\leftarrow$  V[j+1]  
                    V[j+1]  $\leftarrow$  aux  
            FINSI  
        FINPARA  
        i  $\leftarrow$  i+1  
    FINMIENTRAS  
Fin
```



Bibliografía

- **Pascal y Estructuras de Datos.** Dale, N. y Lilly, S. Ed. McGraw Hill 1989
- **Fundamentals of Data Structures in Pascal.** Horowitz, E. y Sahni, S. Computer Science Press, 1994.
- **Algoritmos y Estructuras de Datos.** Wirth, N. Prentice-Hall, 1987.
- **Estructuras de Datos en Pascal.** Tenenbaum, A. y Augenstein, M. Prentice-Hall, 1983.