



# Análisis y diseño de algoritmos

---

## Sesión 05

# *Logro de la sesión*

*Al finalizar la sesión, el estudiante realiza un análisis y desarrollo de algoritmos de búsqueda/selección utilizando un lenguaje de programación*

# Agenda

---

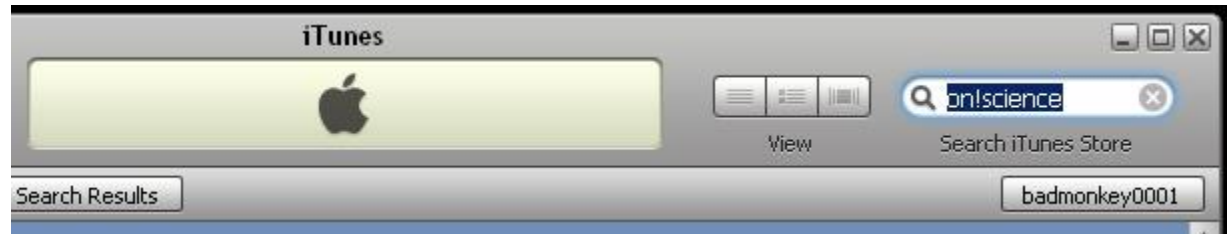
- Búsqueda
- Búsqueda Lineal o Secuencial
- Búsqueda Binaria
- Búsqueda por Transformación de claves o hash
- Búsqueda Indexada
- Quick-Select

# Búsqueda

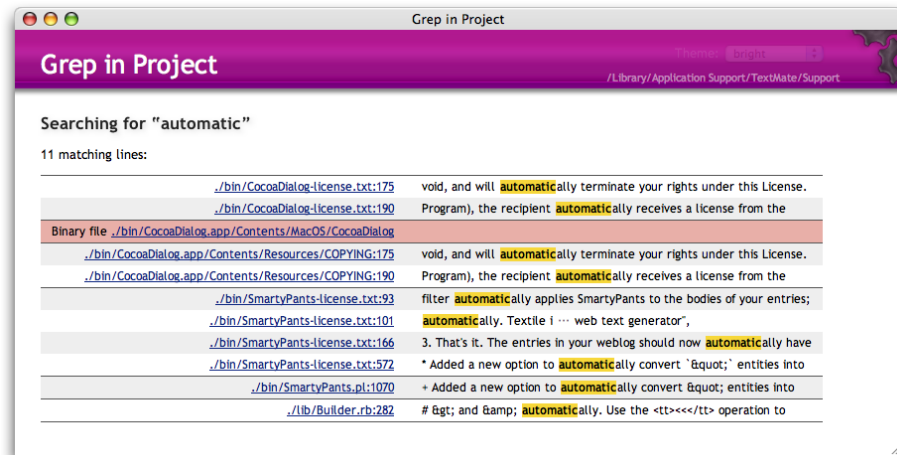
---

- Es un problema fundamental en las ciencias de la computación y programación.
- Múltiples algoritmos para resolver el mismo problema: ¿Cómo sabemos qué algoritmo es "mejor"?
- Los ejemplos usarán arreglos de números enteros para ilustrar los algoritmos.

# Búsqueda



[Advanced Search](#)  
[Preferences](#)  
[Language Tools](#)



## Búsqueda

Proceso para encontrar un elemento particular en un arreglo o lista de elementos.

Con frecuencia trabajábamos con grandes cantidades de información y necesitamos determinar si algún arreglo contiene un valor que sea igual a cierto valor clave.

- **Búsqueda interna**

Si todos los elementos se encuentran en memoria principal (por ejemplo, almacenados en arrays o listas enlazadas)

- **Búsqueda externa**

Si los elementos se encuentran en memoria secundaria.

# Búsqueda

- Dada una lista de datos, encuentre la ubicación de un valor particular o informe que ese valor no está presente

Estudiaremos las siguientes técnicas

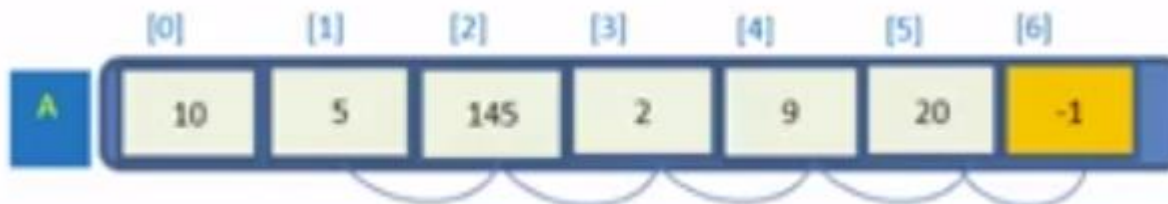
Búsquedas básicas :

- Búsqueda Secuencial
- Búsqueda binaria

Búsquedas avanzadas

- Búsqueda indexada

## Búsqueda Secuencial



- **Búsqueda del menor**

```
menor = a[0];  
for (i=1; i<n; i++)  
    if ( a[i]<menor )  
        menor=a[i];
```

- **Búsqueda del mayor**

```
mayor = a[n-1];  
for (i=0; i<n-1; i++)  
    if ( a[i]>mayor )  
        mayor=a[i];
```

- **Búsqueda de elemento**

```
encontrado=-1;  
for (i=0; i<n; i++)  
    if ( a[i]==elemento_buscado )  
        encontrado=i;
```

# Búsqueda Lineal o Secuencial

---

- Búsqueda lineal o secuencial
  - Enfoque intuitivo
  - Comenzar en el primer elemento ¿es el que estoy buscando?
  - Si no pasa al siguiente elemento, repita hasta que se encuentre o todos los elementos verificados
- Si los elementos no están ordenados o no se pueden ordenar, este enfoque es necesario

# Búsqueda Lineal o Secuencial

```
/* devuelve el índice de la primera aparición del objetivo en la lista o -1 si
el objetivo no está presente en lista */
public int BusquedaLineal(int[] lista, int objetivo) {
    for(int i = 0; i < lista.length; i++)
        if( lista[i] == objetivo )
            return i;
    return -1;
}
```

## Solución genérica

```
/* devuelve el índice de la primera aparición del objetivo en la lista o -1 si el
objetivo no está presente en lista*/

public int BusquedaLineal(Object[] lista, Object objetivo) {
    for(int i = 0; i < lista.length; i++)
        if( lista[i] != null && lista[i].equals(objetivo) )
            return i;
    return -1;
}
```



# Búsqueda Lineal – Lista desordenada y ordenada

```
boolean BusquedaLinealDesordenada(int[] arr, int tamaño, int objetivo) {  
    for(int i = 0 ; i < tamaño ; i++)  
    {  
        if(objetivo == arr[i] )  
            return true;  
    }  
    return false;  
}
```

## Lista ordenada

```
boolean BusquedaLinealOrdenada(int[] arr, int tamaño, int objetivo) {  
    for(int i = 0 ; i < tamaño ; i++){  
        if(objetivo == arr[i] )  
            return true;  
        else if( objetivo < arr[i] )  
            return false;  
    }  
    return false;  
}
```

# Búsqueda Lineal o Secuencial con recursividad

## Ejercicios

Crear un Algoritmo recursivo en Java que busque de forma secuencial un elemento en un arreglo desordenado

Hacer lo mismo para un arreglo ordenado

Utilizar los siguientes Pseudocódigos

### Secuencial\_desordenado\_recursivo ( $V, N, X, I$ )

{Este algoritmo busca secuencialmente, y de forma recursiva, al elemento  $X$  en el arreglo unidimensional desordenado  $V$ , de  $N$  componentes}

{ $I$  es un parámetro de tipo entero que inicialmente se encuentra en 1}

1. Si ( $I > N$ )  
    entonces  
        Escribir "La información no se encuentra en el arreglo"  
    si no  
        1.1 Si ( $V[I] = X$ )  
            entonces  
                Escribir "La información se encuentra en la posición",  $I$   
            si no  
                Regresar a Secuencial\_desordenado\_recursivo con  $V, N, X$  e  $I + 1$   
        1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}

### Secuencial\_ordenado\_recursivo ( $V, N, X, I$ )

{Este algoritmo busca en forma secuencial y recursiva al elemento  $X$  en un arreglo unidimensional ordenado  $V$ , de  $N$  componentes.  $V$  se encuentra ordenado de manera creciente:  $V[1] \leq V[2] \leq \dots \leq V[N]$ .  $I$  inicialmente tiene el valor de 1}

1. Si ( $(I \leq N)$  y ( $X > V[I]$ ))  
    entonces  
        Llamar a Secuencial\_ordenado\_recursivo con  $V, N, X$  e  $I + 1$   
    si no  
        1.1 Si ( $(I > N)$  o ( $X < V[I]$ ))  
            entonces  
                Escribir "La información no se encuentra en el arreglo"  
            si no  
                Escribir "La información se encuentra en la posición",  $I$   
        1.2 {Fin del condicional del paso 1.1}
2. {Fin del condicional del paso 1}

Ejercicio: Mostrar que la búsqueda lineal en un arreglo de n enteros tiene orden n.

```
public static int lsearch( int [] a, int n, int x )
{
    int i = 0;
    boolean encuentre = false;
    while( i<n && !encontre )
        if( a[i] == x )
            encuentre = true;
        else i++;
    if( encuentre ) return i;
    else return -1;
}
```

Tamaño de la entrada:  $n$  = cantidad de componentes de  $a$   
Peor caso:  $x$  no está en  $a$

```
public static int lsearch( int [] a, int n, int x )  
{
```

```
    int i = 0;
```

$c_1$

```
    boolean encuentre = false;
```

$c_2$

```
    while( i < n && !encuentre )
```

Peor caso:  $n$  iteraciones

```
        if( a[i] == x )
```

Tiempo de condición:  $c_3$

```
            encuentre = true;
```

Tiempo del cuerpo:  $c_4$

```
        else i++;
```

```
    if( encuentre ) return i;
```

Tiempo de este if:  $c_5$

```
    else return -1;
```

```
}
```

$$T(n) = c_1 + c_2 + (n(c_3 + c_4) + c_3) + c_5 = O(n)$$

El tiempo es *lineal* en el tamaño de la entrada.

# Búsqueda Lineal – Análisis de Complejidad

La complejidad de la búsqueda secuencial diferencia entre el comportamiento en el peor y mejor caso. El mejor caso se encuentra cuando aparece una coincidencia en el primer elemento de la lista, por lo que el tiempo de ejecución es  $O(1)$ . El peor caso se produce cuando el elemento no está en la lista o se encuentra al final de ella. Esto requiere buscar en todos los  $n$  términos, lo que implica una complejidad de  $O(n)$ .

$$C_{\min} = 1 \qquad C_{\text{med}} = \frac{(1+n)}{2} \qquad C_{\max} = N$$

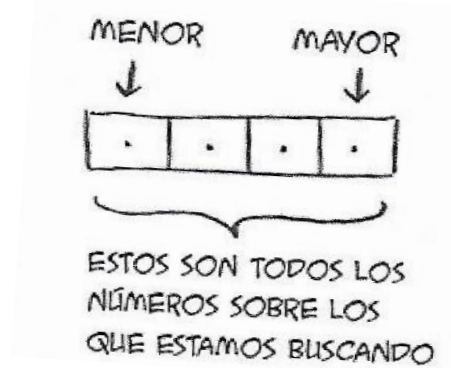
$N$	$C_{\min}$	$C_{\text{med}}$	$C_{\max}$
10	1	5.5	10
100	1	50.5	100
500	1	250.5	500
1 000	1	500.5	1 000
10 000	1	5 000.5	10 000

Rendimiento en el peor caso	$O(n)$
Rendimiento en el mejor caso	$O(1)$
Rendimiento en el caso promedio	$O(n)$

# Búsqueda Binaria

## Escribir el algoritmo de búsqueda binaria en Java

Escribiremos en Java el método `busqueda_binaria` que toma un arreglo ordenado (`lista`) y el elemento a buscar (`elemento`). Si el elemento está en el arreglo, el método retorna su posición. Harás un seguimiento de qué parte de la matriz o arreglo tienes que buscar.



Al principio, este es todo el arreglo

```
menor = 0
```

```
mayor = lista.length - 1
```

Cada intento de búsqueda escogería el elemento ubicado en mitad del arreglo y se haría hasta que se haya reducido búsqueda a un solo elemento:

```
La medio = (menor+mayor)/2  
La estimado = lista[medio]
```

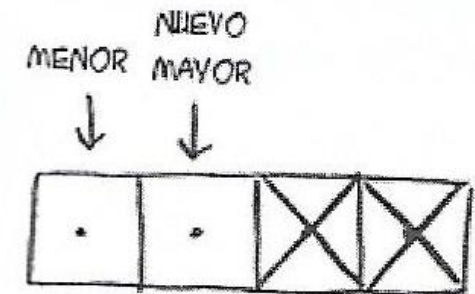
medio debe ser redondeado si (`menor+mayor`) no es un número par

Si el número encontrado es muy bajo, actualiza la variable `menor`, para intentar ahora con un arreglo reducido:

```
if (estimado < elemento){  
    menor = medio + 1  
}
```

Si el número encontrado es muy alto, actualiza la variable `mayor`, para intentar ahora con un arreglo reducido:

```
if (estimado > elemento){  
    mayor = medio - 1  
}
```




# Codificación de la Búsqueda Binaria

## El algoritmo de búsqueda binaria en Java

```
1 package Semanal;
2 import javax.swing.JOptionPane;
3 public class AED_Programa1 {
4     public static void main(String[] args) {
5         int[] lista={1,3,5,7,9,10,16,19,21,24,27,29,31,31,33,36,39,41,43,45};
6         AED_Programa1 ejemplo1=new AED_Programa1();
7         int elemento=Integer.parseInt(JOptionPane.showInputDialog("Ingrese el elemento que desea buscar del 1 al 45"));
8         int resultado=ejemplo1.búsqueda_binaria(lista, elemento);
9         if(resultado!=-1){
10            System.out.println("El numero buscado está en la posición "+ resultado);
11        }else{
12            System.out.println("El numero buscado no está en la lista");
13        }
14    }
15    private int búsqueda_binaria(int[] lista, int elemento){
16        int menor=0,medio;
17        int mayor=lista.length - 1;
18        while(menor<=mayor){
19            medio=menor+(mayor-menor)/2;
20            if(lista[medio]==elemento){
21                return medio;
22            }else if(lista[medio]>elemento){
23                mayor=medio-1;
24            }else{
25                menor=medio+1;
26            }
27        }
28        return -1;
29    }
30 }
```

Entrada

 Ingrese el elemento que desea buscar del 1 al 45

16

Aceptar Cancelar

Output - AED\_Programa1 (run) X

run:

El numero buscado está en la posición 6

BUILD SUCCESSFUL (total time: 32 seconds)

# Busqueda Binaria recursiva en Java

```
1 package Semana2;
2 import javax.swing.*;
3 public class AED_Programa11 {
4     public static void main(String[] args) {
5         int[] lista={1,3,5,7,9,10,16,19,21,24,27,29,31,31,33,36,39,41,43,45};
6         int elemento=Integer.parseInt(JOptionPane.showInputDialog("Ingrese el elemento que desea buscar del 1 al 45"));
7         int resultado=busqueda_binaria_recursiva(lista, elemento);
8         if(resultado!=-1){
9             System.out.println("El numero buscado está en la posición "+ resultado);
10        }else{
11            System.out.println("El numero buscado no está en la lista");
12        }
13    }
14    public static int busqueda_binaria_recursiva(int[] lista, int elemento){
15        return busqueda_binaria_recursiva(lista,elemento,0,lista.length-1);
16    }
17    public static int busqueda_binaria_recursiva(int[] lista, int elemento, int menor, int mayor){
18        int medio=menor+(mayor-menor)/2;
19        if((menor>=mayor)&&(lista[menor]!=elemento)){
20            return -1;
21        }else if(lista[medio]==elemento){
22            return medio;
23        }else if(elemento<lista[medio]){
24            return busqueda_binaria_recursiva(lista,elemento,menor,medio-1);
25        }
26        return busqueda_binaria_recursiva(lista,elemento,medio+1,mayor);
27    }
28 }
29 }
```



# Búsqueda Binaria con Bandera

## Ejercicios

Crear un Algoritmo en Java que busque de forma binaria un elemento en un arreglo ordenado

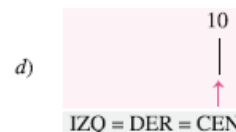
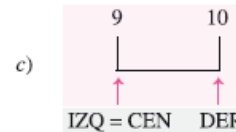
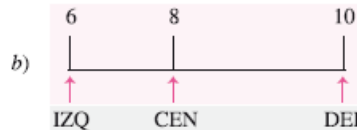
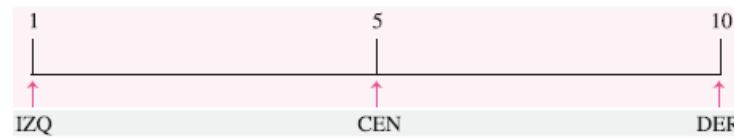
Utilizar el siguiente Pseudocódigo

Se busca  $X=615$

$V$

101	215	325	410	502	507	600	610	612	670
1	2	3	4	5	6	7	8	9	10

Paso	BAN	IZQ	DER	CEN	$X = V[CEN]$	$X > V[CEN]$
1	Falso	1	10	5	$615 = 502$ ? No	$615 > 502$ ? Sí
2	Falso	6	10	8	$615 = 610$ ? No	$615 > 610$ ? Sí
3	Falso	9	10	9	$615 = 612$ ? No	$615 > 612$ ? Sí
4	Falso	10	10	10	$615 = 670$ ? No	$615 > 670$ ? No
5	Falso	10	9			



### Binaria ( $V, N, X$ )

{Este algoritmo busca al elemento  $X$  en un arreglo unidimensional ordenado  $V$  de  $N$  componentes}

{IZQ, CEN y DER son variables de tipo entero. BAN es una variable de tipo booleano}

- Hacer  $IZQ \leftarrow 1$ ,  $DER \leftarrow N$  y  $BAN \leftarrow \text{FALSO}$
- Mientras  $((IZQ \leq DER) \text{ y } (BAN = \text{FALSO}))$  Repetir
  - Si  $(X = V[CEN])$   
entonces  
Hacer  $BAN \leftarrow \text{VERDADERO}$   
si no {Se redefine el intervalo de búsqueda}
  - Si  $(X > V[CEN])$   
entonces  
Hacer  $IZQ \leftarrow CEN + 1$   
si no  
Hacer  $DER \leftarrow CEN - 1$
  - {Fin del condicional del paso 2.1.1}
  - {Fin del condicional del paso 2.1}
- {Fin del ciclo del paso 2}
- Si  $(BAN = \text{VERDADERO})$   
entonces  
Escribir "La información está en la posición", CEN  
si no  
Escribir "La información no se encuentra en el arreglo"
- {Fin del condicional del paso 4}

Ejercicio: Mostrar que la búsqueda binaria (dicotómica) en un arreglo de  $n$  componentes ordenadas en forma ascendente tiene orden logarítmico de base 2 en la cantidad de elementos del arreglo.

```
public static int bsearch( int [] a, int n, int x ) {  
    int ini = 0, fin = n-1;  
    while( ini <= fin ) {  
        int medio = (ini + fin) / 2;  
        if( a[medio] == x ) return medio;  
        else if( a[medio] > x ) fin = medio-1;  
        else ini = medio + 1;  
    }  
    return -1;  
}
```

Tamaño de la entrada:  $n$  = cantidad de componentes de  $a$   
Peor caso:  $x$  no está en  $a$

```
public static int bsearch( int [] a, int n, int x ) {  
    int ini = 0, fin = n-1;            $c_1$   
    while( ini <= fin ) {             Tiempo de la condición:  $c_2$   
        int medio = (ini + fin) / 2;   Tiempo del cuerpo:  $c_3$   
        if( a[medio] == x ) return medio;  
        else if( a[medio] > x ) fin = medio-1;  
        else ini = medio + 1;  
    }  
    return -1;                          $c_4$   
}
```

Sea  $k$  = cantidad de iteraciones del while, entonces

$$T(n) = c_1 + k(c_2 + c_3) + c_2 + c_4.$$

La pregunta es cómo definir  $k$  en función de  $n$ .

## ¿Cómo estimar k en función de n?

El peor caso es cuando “x” no está en “a”.

Veamos cómo vamos descartando componentes del arreglo en función del número de iteración del while: Si tenemos n componentes, en cada iteración se descarta la componente del medio del arreglo, entonces de las n-1 componentes que falta revisar sólo se va considerar la mitad, entonces quedan  $(n-1)/2$  componentes para la siguiente iteración, y así sucesivamente.

Calculemos cuál es caso para la iteración genérica k.

Número de iteración	Componentes por revisar
1	n
2	$(n-1)/2$
3	$(n-3)/4$
4	$(n-7)/8$
5	$(n-15)/16$
...	...
k	$\frac{n - (2^{k-1} - 1)}{2^{k-1}}$

Entonces vimos que en la iteración k, la cantidad de componentes que quedan por revisar es:

$$\frac{n - (2^{k-1} - 1)}{2^{k-1}}$$

Como x no está en el arreglo a, en la última iteración completa que realiza el while queda una componente del arreglo por revisar.

Entonces:

$$\frac{n - (2^{k-1} - 1)}{2^{k-1}} = 1;$$

$$n - (2^{k-1} - 1) = 2^{k-1};$$

$$n - 2^{k-1} + 1 = 2^{k-1};$$

$$n + 1 = 2^{k-1} + 2^{k-1};$$

$$n + 1 = 2 \times 2^{k-1};$$

$$n + 1 = 2^k;$$

$$\log_2(n + 1) = k.$$

Con lo que  $T(n) = c_1 + \log_2(n+1)(c_2+c_3) + c_2 + c_4$ .

Luego, por regla de la suma,  $T(n) = O(\log_2(n))$  q.e.d.

# Búsqueda binaria

Problema: Buscar entero “x” en arreglo de enteros ordenado “a” de “n” componentes

```
public static int bsearch( int [] a, int n, int x ) {  
    return bsearch_aux( a, 0, n-1, x );  
}  
private static int bsearch_aux(int [] a, int ini, int fin, int x ) {  
    if( ini <= fin ) {  
        int medio = (ini + fin) / 2;  
        if( a[medio] == x ) return medio;  
        else if( a[medio] > x ) then  
            return bsearch_aux( a, ini, medio-1, x)  
        else  
            return bsearch_aux( a, medio+1, fin , x)  
        }  
    else return -1; // x no está en el arreglo  
}
```

- Paso 1: Entrada: Arreglo a y x
- Paso 2: Tamaño de entrada:  
n = cantidad de componentes de a
- Paso 3: Definición recursiva de T(n):

$$T(n) = \begin{cases} c_1 & , si \quad n = 0 \\ c_2 + T(\frac{n-1}{2}) & , si \quad n \geq 1 \end{cases}$$

• Paso 4: Obtener definición no recursiva de  $T(n)$

$$\begin{aligned} T(n) &= c_2 + T((n-1)/2) = c_2 + (c_2 + T(((n-1)/2 - 1)/2)) \\ &= 2c_2 + T((n-3)/4) = 2c_2 + (c_2 + T(((n-3)/4 - 1)/2)) \\ &= 3c_2 + T((n-7)/8) = 3c_2 + (c_2 + T(((n-7)/8 - 1)/2)) \\ &= 4c_2 + T((n-15)/16) = 4c_2 + (c_2 + T(((n-15)/16 - 1)/2)) \\ &= 5c_2 + T((n-31)/32) = \dots \\ &= ic_2 + T((n-(2^i-1))/2^i) \end{aligned} \quad (1)$$

Termina cuando  $(n-(2^i-1))/2^i = 0$ , luego  $n-(2^i-1) = 0$ .

Entonces,  $n-2^i+1=0$ ; por lo tanto,  $n+1 = 2^i$  y  $i = \log_2(n+1)$ . (2)

Reemplazo (2) en (1):

$$T(n) = \log_2(n+1)c_2 + T(0) = \log_2(n+1)c_2 + c_1.$$



- Paso 5: Dar orden de tiempo de ejecución:

$$T(n) = \log_2(n+1)c_2 + c_1 \text{ es } O(\log_2(n+1)).$$

- Paso 6: Prueba inductiva (por inducción transfinita)

Caso base:  $T(0) = c_1 = \log_2(0+1)c_2 + c_1 = c_1$

Caso inductivo:  $T(n) = c_2 + T((n-1)/2)$  *(x def. T(n) rec.)*

$$= c_2 + (\log_2((n-1)/2+1)c_2 + c_1)$$
 *(x hipótesis inductiva)*

$$= c_2 + (\log_2((n-1+2)/2))c_2 + c_1$$

$$= c_2 + (\log_2(n+1) - \log_2(2))c_2 + c_1$$

$$\text{(xq } \log(a/b) = \log(a) - \log(b))$$

$$= c_2 + (\log_2(n+1) - 1)c_2 + c_1$$

$$= c_2 + \log_2(n+1)c_2 - c_2 + c_1$$

$$= \log_2(n+1)c_2 + c_1.$$

# Búsqueda Binaria – Análisis de Complejidad

El mejor caso se presenta cuando una coincidencia se encuentra en el punto central de la lista. En este caso, la complejidad es  $O(1)$ , dado que sólo se realiza una prueba de comparación de igualdad. La complejidad del peor caso es  $O(\log_2 n)$ , que se produce cuando el elemento no está en la lista o el elemento se encuentra en la última comparación.

$$C_{\min} = 1 \quad C_{\text{med}} = \frac{(1 + \log_2(N))}{2} \quad C_{\max} = \log_2(N)$$

$N$	$C_{\min}$	$C_{\text{med}}$	$C_{\max}$
10	1	2.5	4
100	1	4	7
500	1	5	9
1 000	1	5.5	10
10 000	1	7.5	14

Rendimiento en el peor caso	$O(\log n)$
Rendimiento en el mejor caso	$O(1)$

# Búsqueda por Transformación de claves o hash

*Este método permite aumentar la velocidad de búsqueda sin necesidad de tener los elementos ordenados, cuenta con la ventaja de que el tiempo de búsqueda es independiente del número de componentes del arreglo:*

## Ejemplo simple

Supongamos que estamos usando nombres como nuestra clave:

❖ Toma la tercera letra del nombre, toma el valor int de la letra (a = 0, b = 1, ...), divide entre 6 y toma el resto

❖ ¿Qué número esconde el nombre “Bellers”?

✓ 1 -> 11 ->  $11 \% 6 = 5$

- Mike =  $(10 \% 6) = 4$
- Kelly =  $(11 \% 6) = 5$
- Olivia =  $(8 \% 6) = 2$
- Isabelle =  $(0 \% 6) = 0$
- David =  $(21 \% 6) = 3$
- Margaret =  $(17 \% 6) = 5$  (uh oh)
- Wendy =  $(13 \% 6) = 1$

- *Esta es una función hash imperfecta. Una función hash perfecta produce una asignación uno a uno de las claves a los valores hash. En este caso para Kelly y Margaret se genera una “colisión”*
- *¿Cuál es el número máximo de valores que esta función puede combinar perfectamente?*



# Búsqueda por Transformación de claves o hash

Índice	Valor
0	
1	
2	
3	
4	
5	
6	
7	

20	33	7	10	12	14	56	100
----	----	---	----	----	----	----	-----

# Búsqueda por Transformación de claves o hash

## Tabla Hash

- Es una Estructura de datos.
- Se basan en la asignación de una clave a cada elemento.
- Inserción y búsqueda rápida.
- Limitadas en tamaño ya que están basados en arreglos.
- Su tamaño  $- 1$  es recomendable que sea un numero primo
- Las claves son asignadas a elementos en una Tabla Hash usando una Función Hash.
- Una Función Hash ayuda a calcular el índice optimo en el cual un elemento debería ubicarse.
- El índice debe se menor que el tamaño de la tabla y mayor o igual a cero.
- No debe haber datos repetidos en una Tabla Hash.

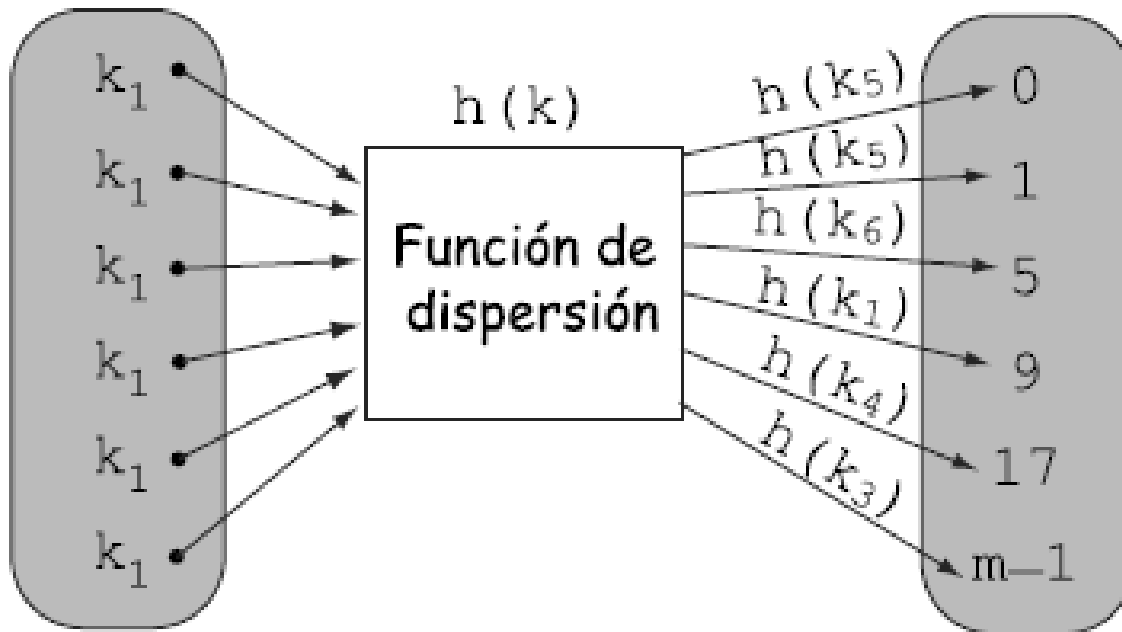
# Más sobre funciones hash o de dispersión

Normalmente una función hash o de dispersión es un proceso de dos pasos:

- Transforma la clave (que puede no ser un número entero) en un valor entero
- Asigna el entero resultante en un índice válido para la tabla hash o tabla de dispersión (donde se almacenan todos los elementos)

Variable independiente:  
campo clave

Índices de tabla



Para facilitar la búsqueda, una función hash debe generar posiciones diferentes dadas dos claves diferentes. Si esto último no ocurre ( $H(K1)=d, H(K2)=d$  y  $K1 \neq K2$ ) se produce una colisión.

Entonces una función hash debe ser fácil de calcular y distribuir uniformemente las claves y debe evitar las colisiones. Si éstas se presentan se contará con algún método que genere posiciones alternativas.

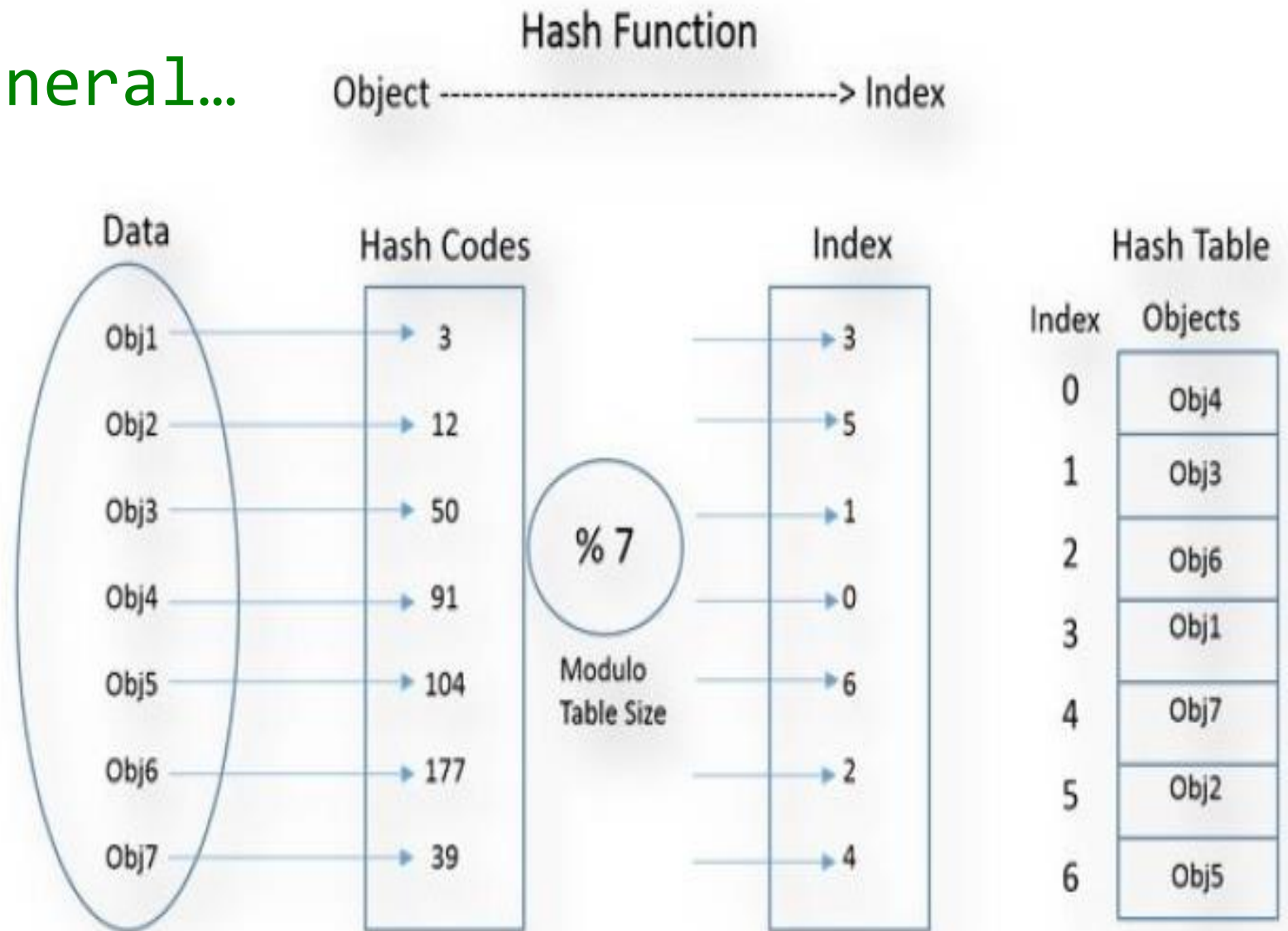
- Las funciones hash pueden basarse en una serie de técnicas, entre las más utilizadas tenemos: función hash por módulo, función hash cuadrado, función hash por plegamiento, función hash por truncamiento.

# Búsqueda por Transformación de claves o hash

## Un proceso mas general...

*El proceso de almacenar objetos usando una función hash es el siguiente:*

- 1. Cree un arreglo de tamaño  $M$  para almacenar objetos, este arreglo se llama Hash-Table.*
- 2. Encuentre un código hash de un objeto pasándolo a través de la función hash.*
- 3. Tome el módulo de código hash por el tamaño de Hash-Table para obtener el índice de la tabla donde se almacenarán los objetos.*
- 4. Finalmente, almacene estos objetos en el índice designado.*



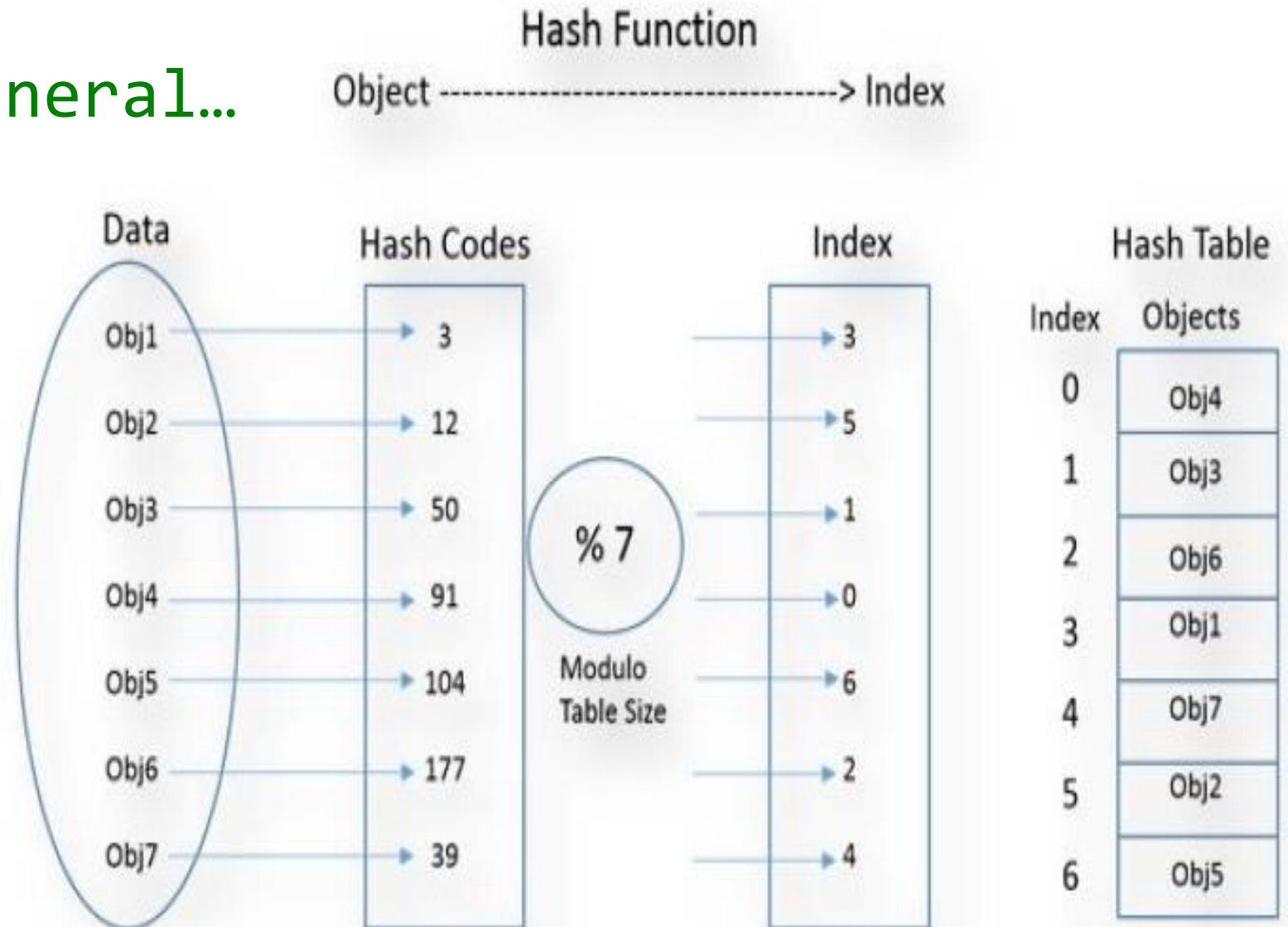


# Búsqueda por Transformación de claves o hash

## Un proceso mas general...

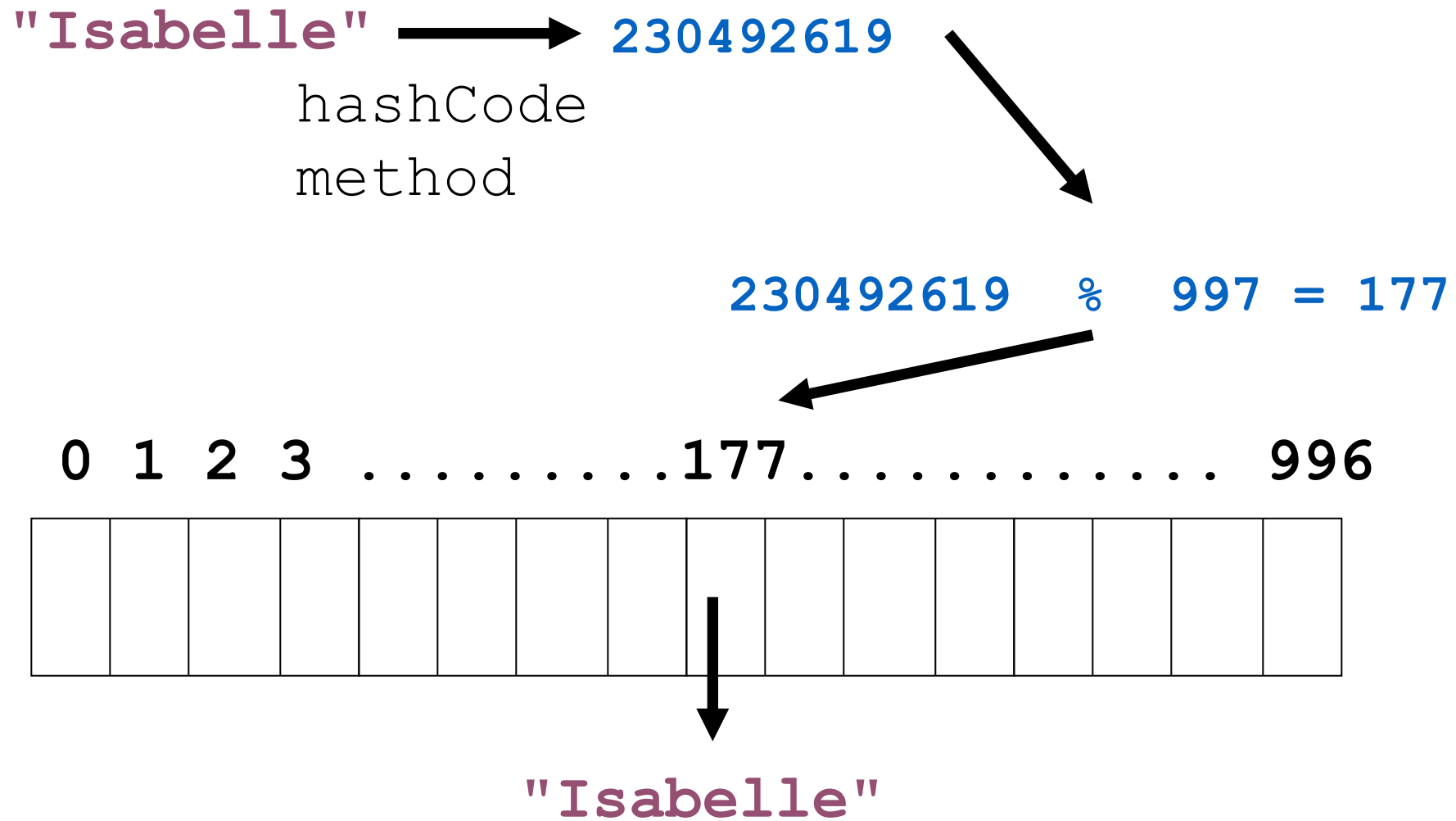
El proceso de búsqueda de objetos en una Hash-Table usando una función hash es el siguiente:

1. Encuentre un código hash del objeto que estamos buscando pasándolo a través de la función hash.
2. Tome el módulo de código hash por el tamaño de la tabla hash para obtener el índice de la tabla donde se almacenan los objetos.
3. Finalmente, recupere el objeto del índice designado.





# Búsqueda por Transformación de claves o hash



# Codificación de la Búsqueda por Transformación de claves o hash

## Clase HashTable

```
1 package Clases;
2 public class HashTable {
3     private static int EMPTY_NODE = -1;
4     private static int LAZY_DELETED = -2;
5     private static int FILLED_NODE = 0;
6     private int tableSize;
7     int[] Arr;
8     int[] Flag;
9     public HashTable(int tSize) {
10         tableSize = tSize;
11         Arr = new int[tSize + 1];
12         Flag = new int[tSize + 1];
13         for (int i = 0; i <= tSize; i++) {
14             Flag[i] = EMPTY_NODE;
15         }
16     }
17     int ComputeHash(int key) {
18         return key % tableSize;
19     }
20     int resolverFun(int index) {
21         return index;
22     }
23     boolean InsertNode(int value) {
24         int hashValue = ComputeHash(value);
25         for (int i = 0; i < tableSize; i++) {
26             if (Flag[hashValue] == EMPTY_NODE || Flag[hashValue] == LAZY_DELETED) {
27                 Arr[hashValue] = value;
28                 Flag[hashValue] = FILLED_NODE;
29                 return true;
30             }
31             hashValue += resolverFun(i);
32             hashValue %= tableSize;
33         }
34         return false;
35     }
36 }
```

# Codificación de la Búsqueda por Transformación de claves o hash

## Clase HashTable

```
36  boolean FindNode(int value) {
37      int hashValue = ComputeHash(value);
38      for (int i = 0; i < tableSize; i++) {
39          if (Flag[hashValue] == EMPTY_NODE) {
40              return false;
41          }
42          if (Flag[hashValue] == FILLED_NODE && Arr[hashValue] == value) {
43              return true;
44          }
45          hashValue += resolverFun(i);
46          hashValue %= tableSize;
47      }
48      return false;
49  }
50  void Print() {
51      for (int i = 0; i < tableSize; i++) {
52          if (Flag[i] == FILLED_NODE) {
53              System.out.println("Node at index [" + i + " ] :: " + Arr[i]);
54          }
55      }
56  }
```

# Codificación de la Búsqueda por Transformación de claves o hash

## Clase UsoHash

```
1  package Clases;
2  import java.util.Scanner;
3  public class Usohash {
4      public static void main(String[] args) {
5          Scanner entrada=new Scanner(System.in);
6          System.out.println("Ingrese el tamaño de la tabla");
7          int tamaño=entrada.nextInt();
8          int[] listaoriginal=new int[tamaño];
9          for(int i=0;i<tamaño;i++){
10              listaoriginal[i]=(int) (Math.random()*1000);
11          }
12          HashTable ht=new HashTable(tamaño);
13          System.out.println("\nTabla original\n");
14          for(int i=0;i<tamaño;i++){
15              System.out.println("Original at index[" +i+ "]::"+listaoriginal[i]);
16          }
17          System.out.println("\nTabla Hash con función módulo\n");
18          for(int i=0;i<tamaño;i++){
19              ht.InsertNode(listaoriginal[i]);
20          }
21          ht.Print();
22          System.out.println("\nSearch 243 ::"+ht.FindNode(243));
23      }
24  }
```

# Ejercicio

Implementar el siguiente Pseudocódigo con una función hash por módulo o división:

- La función hash por módulo o división consiste en tomar el residuo de la división de la clave entre el número de componentes del arreglo donde se almacenarán las claves.
- Supongamos que tenemos un arreglo de  $N$  elementos y  $K$  es la clave del dato a buscar. La función hash queda definida por la siguiente fórmula:  $H(K) = (K \bmod N) + 1$
- Para lograr mayor uniformidad es importante que  $N$  sea un número primo o divisible entre muy pocos números

1	80
2	
3	
4	43
5	54
6	25
7	56
8	35
9	13
10	104

$K$	$H(K)$
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

Buscando el 13

## Prueba\_lineal ( $V, N, K$ )

{Este algoritmo busca al dato con clave  $K$  en el arreglo unidimensional  $V$  de  $N$  elementos.  
Resuelve el problema de las colisiones por medio del método de prueba lineal}  
{ $D$  y  $DX$  son variables de tipo entero}

- Hacer  $D \leftarrow H(K)$  {Genera dirección}
- Si  $((V[DX] \neq \text{VACÍO}) \text{ y } (V[D] = K))$   
entonces  
Escribir "La información está en la posición",  $D$   
si no  
Hacer  $DX \leftarrow D + 1$ 
  - Mientras  $((DX \leq N) \text{ y } (V[DX] \neq \text{VACÍO}) \text{ y } (V[DX] \neq K) \text{ y } (DX \neq D))$   
Repetir  
Hacer  $DX \leftarrow DX + 1$ 
    - Si  $(DX = N + 1)$  entonces  
Hacer  $DX \leftarrow 1$
    - {Fin del condicional del paso 2.1.1}
  - {Fin del ciclo del paso 2.1}
  - Si  $((V[DX] = \text{VACÍO}) \text{ o } (DX = D))$   
entonces  
Escribir "La información no se encuentra en el arreglo"  
si no  
Escribir "La información está en la posición",  $DX$
  - {Fin del condicional del paso 2.3}
- {Fin del condicional del paso 2}

$D$	$DX$
4	5
	6
	7
	8
	9

# Big O Colecciones Java

- Implementaciones de Listas

	get	add	contains	next	remove(0)	iterator.remove
ArrayList	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
CopyOnWrite-ArrayList	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$

- Implementaciones de Conjuntos (Set)

	add	contains	next	notes
HashSet	$O(1)$	$O(1)$	$O(h/n)$	$h$ is the table capacity
LinkedHashSet	$O(1)$	$O(1)$	$O(1)$	
CopyOnWriteArraySet	$O(n)$	$O(n)$	$O(1)$	
EnumSet	$O(1)$	$O(1)$	$O(1)$	
TreeSet	$O(\log n)$	$O(\log n)$	$O(\log n)$	
ConcurrentSkipListSet	$O(\log n)$	$O(\log n)$	$O(1)$	

- Implementaciones Mapas o Diccionarios (Map)

	get	containsKey	next	Notes
HashMap	$O(1)$	$O(1)$	$O(h/n)$	$h$ is the table capacity
LinkedHashMap	$O(1)$	$O(1)$	$O(1)$	
IdentityHashMap	$O(1)$	$O(1)$	$O(h/n)$	$h$ is the table capacity
EnumMap	$O(1)$	$O(1)$	$O(1)$	
TreeMap	$O(\log n)$	$O(\log n)$	$O(\log n)$	
ConcurrentHashMap	$O(1)$	$O(1)$	$O(h/n)$	$h$ is the table capacity
ConcurrentSkipListMap	$O(\log n)$	$O(\log n)$	$O(1)$	

- Implementaciones de pilas y colas (queue)

	offer	peek	poll	size
PriorityQueue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
ConcurrentLinkedQueue	$O(1)$	$O(1)$	$O(1)$	$O(n)$
ArrayBlockingQueue	$O(1)$	$O(1)$	$O(1)$	$O(1)$
LinkedBlockingQueue	$O(1)$	$O(1)$	$O(1)$	$O(1)$
PriorityBlockingQueue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
DelayQueue	$O(\log n)$	$O(1)$	$O(\log n)$	$O(1)$
LinkedList	$O(1)$	$O(1)$	$O(1)$	$O(1)$
ArrayDeque	$O(1)$	$O(1)$	$O(1)$	$O(1)$
LinkedBlockingDeque	$O(1)$	$O(1)$	$O(1)$	$O(1)$

<https://www.bigocheatsheet.com/>

# Búsqueda Indexada

Mediante cada elemento del array índice se asocian grupos de elementos del array inicial.

Los elementos en el índice y en el array deben estar ordenados.

El método consta de dos pasos:

1. Buscar en el array índice el intervalo correspondiente al elemento buscado
2. restringir la búsqueda a los elementos del intervalo que se localizó previamente.

La ventaja es que la búsqueda se realiza en el array de índices y no en el array de elementos.

Cuando se ha encontrado el intervalo correcto se hace segunda búsqueda en una parte reducida del array.





# Búsqueda Indexada

3	4	5	6	7	8	9	10	14	16	19
---	---	---	---	---	---	---	----	----	----	----

Se decide dividir el array inicial en bloques de tres elementos. El array índice estará formado por  $n/k + 1$  elementos, por no ser el último bloque de igual tamaño, es decir,  $11/3 + 1 = 4$

La cantidad de elementos que debe conformar cada grupo o bloque (k) debe ser la raíz cuadrada del tamaño del arreglo

Clave	→	3	6	9	16
Posición	→	0	3	6	9



# Codificación de la Búsqueda Indexada

## Clase index

```
1 package busquedainterna;
2 public class index {
3     int[][] indice;
4     int tam;
5     public index(int k, int A[]){
6         int n_indices,n=A.length,cont=0;
7         if (n%k == 0) n_indices=n/k;
8         else n_indices=n/k + 1;
9         this.indice=new int[n_indices][2];
10        this.tam=k;
11        for(int i=0;i<n_indices;i++){
12            this.indice[i][0]=cont;
13            this.indice[i][1]=A[cont];
14            cont+=k;
15        }
16    }
17    public int getindice(int X){
18        int inicio=0;
19        int fin=this.indice.length-1;
20        for(int i=inicio;i<=fin;i++){
21            System.out.println(i+"==>" +this.indice[i][0]+"---"+this.indice[i][1]);
22            if(this.indice[i][1]>=X){
23                i=i-1;
24                if(i<0) i=0;
25                return this.indice[i][0];
26            }
27        }
28        return this.indice[fin][0];
29    }
30 }
```

```
31 public static int BusquedaIndexada(int[] A,int X,index indice,int bloque){
32     int inicio=indice.getindice(X);
33     int fin=inicio+bloque;
34     System.out.println("inicio : "+inicio);
35     if(fin>=A.length)
36         fin=A.length-1;
37     for(int i=inicio;i<=fin;i++){
38         if(A[i]==X)
39             return i;
40     }
41     return -1;
42 }
43 }
44 }
```

## Uso de la búsqueda indexada

```
64 k1 = System.currentTimeMillis();
65 int bloque = (int) (Math.sqrt(c.length));
66 index IndiceA = new index(bloque, c);
67 int posicion5 = index.BusquedaIndexada(c, objetivo, IndiceA, bloque);
68 if (posicion5 != -1) {
69     System.out.println("\nEl numero está en la posicion " + (posicion5 + 1) + " del arreglo ordenado");
70 } else {
71     System.out.println("\nEl numero no se encuentra");
72 }
73 k2 = System.currentTimeMillis();
74 System.out.println("Tiempo de busqueda indexada "
75     + (k2 - k1));
```

# El problema de la selección

- Dado un entero  $k$  y  $n$  elementos  $x_1, x_2, \dots, x_n$ , desordenados, encontrar el  $k$ -ésimo elemento más pequeño de este conjunto.
- También llamadas estadísticas de orden, la  $i$ -ésima estadística de orden es el  $i$ -ésimo elemento más pequeño
- Mínimo  $k=1$ : Estadístico de primer orden
- Máximo  $k=n$ : Estadístico de orden  $n$
- Mediana  $k=n/2$
- etc.

# El problema de la selección

- Solución ingenua: ¡ORDENAR!
- Podemos ordenar el conjunto en tiempo  $O(n \log n)$  y luego obtenemos el  $k$ -ésimo elemento en el índice  $k$

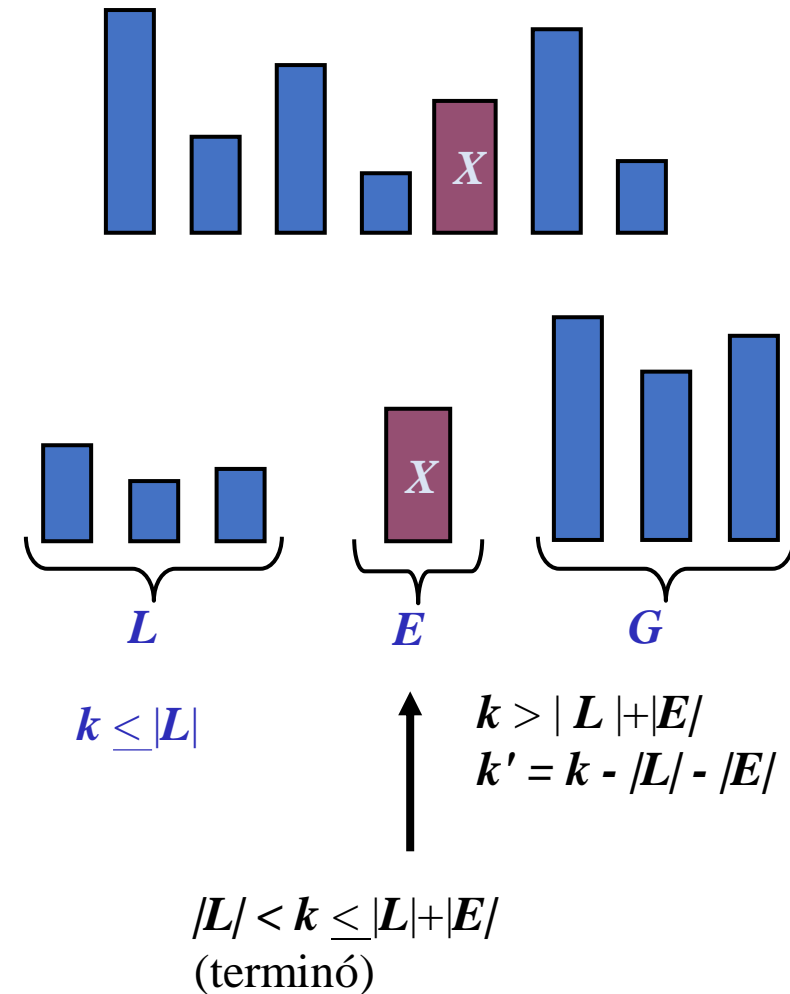
7 4 9 6 2 → 2 4 6 7 9

$k=3$

- ¿Podemos resolver el problema de selección más rápido?

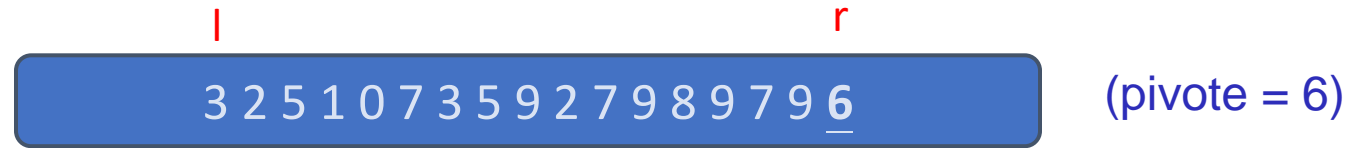
# Quick-Select

- Quick-Select es un algoritmo de selección **aleatoria** basado en el paradigma de podar y buscar, para seleccionar el  $k$ -esimo menor( $k$ ) :
  - Podar** : elija un elemento aleatorio  $x$  (llamado **pivote**) y divida  $S$  en
    - $L$  elementos menos que  $x$
    - $E$  elementos iguales a  $x$
    - $G$  elementos mayores que  $x$
  - Buscar** : dependiendo de  $k$ , la respuesta está en  $E$ , o necesitamos repetir en  $L$  o  $G$
  - Nota: La partición es igual que en Quicksort

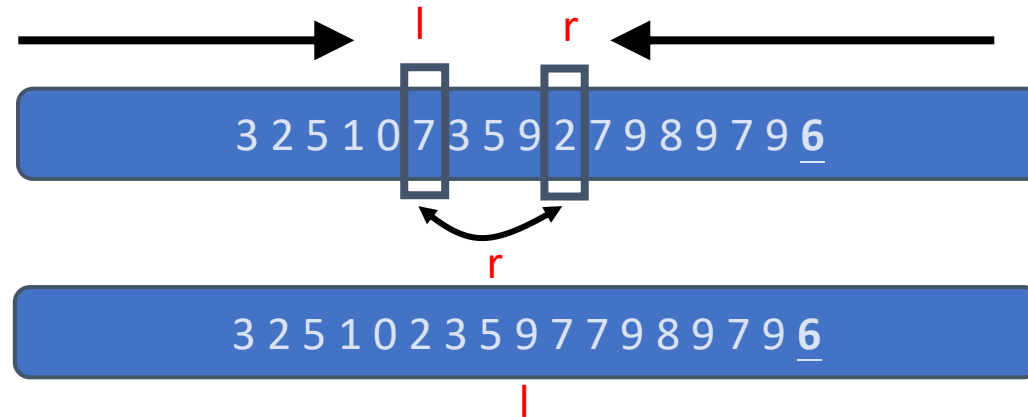


# La Partición en Quick-select

- Realice la partición usando dos índices para dividir **S** en dos subarreglos L y E & G (L: Less, E & G: Equal & Greater)



- Repita hasta que `l` y `r` se crucen:
  - Escanea `l` hacia la derecha hasta encontrar un elemento  $\geq x$  (pivote).
  - Escanea `r` hacia la izquierda hasta encontrar un elemento  $< x$  (pivote).
  - Intercambia elementos en los índices `l` y `r`. Cuando se cruzan `l` y `r` devuelve `r`



# Pseudocódigo de la partición en Quick-Select

- $q = \text{inPlacePartition}(S, \text{primero}, \text{ultimo})$  devuelve el índice  $q$  tal que
- $L = S[\text{primero} \dots q]$
- $E\&G = S[q+1 \dots \text{último}]$

//Este Algo usa el último elemento como pivote

**Algoritmo**  $q = \text{inPlacePartition} ( S, \text{primero}, \text{ultimo} )$

**Entrada:**

secuencia **Array**  $S$  , el índice inicial es primero, y el índice final es ultimo

**Salida:**

Devuelve un índice  $q$  tal que

$S[\text{primero} \dots q] \leq S[q + 1, \dots \text{ultimo}]$

**Código:**

```
pivote ←  $S[\text{ultimo}]$ ;  
 $l$  ←  $\text{primero} - 1$ ; // l se incrementa a continuación  
 $r$  ←  $\text{ultimo}$ ; // comienza desde el final dejando el pivote  
While ( VERDADERO ) {  
    do { $l++$ ;} while(  $S[l] < \text{pivote}$ );  
    do { $r--$ ;} while (  $S[r] \geq \text{pivote}$ );  
    if (  $l \geq r$  ) {  
        return  $r$ ;  
    }  
    intercambiar (  $S, l, r$  );  
}
```

# Pseudocódigo Quick-Select

//Entrada: Array S[primero .. ultimo] y el rango(k)

//Salida: devuelve el k-ésimo elemento más pequeño en S

Quick\_select(S, primero, ultimo, k)

{

if (primero == último) // Caso base: arreglo de un elemento

return S[primero];

q = *inPlacePartition*( S, primero, ultimo );

L\_Tamaño = q – primero + 1 //Tamaño del arreglo L

if (k <= L\_Tamaño) {

return Quick\_select(S, primero, q, k); //El arreglo L

} else

return Quick\_select(S, q+1, ultimo, k - L\_Tamaño) //Arreglo E&G

}

}

El tiempo de ejecución  
esperado es  $O(n)$

k=5, S=(7 4 9 2 6 5 1 8 3)

k=2, S=(7 4 9 6 5 8)

k=2, S=(7 4 6 5)

k=1, S=(7 6 5)

5

# Análisis de Complejidad Temporal Quick-Select

**El peor de los casos:** El peor de los casos ocurre cuando elegimos el elemento más grande / más pequeño como pivote.

$$\begin{aligned} T(n) &= T(n-1) + cn \\ &= T(n-1) + T(n-2) + cn + c(n-1) \\ &= O(n^2) \end{aligned}$$

**El mejor caso:** El mejor caso ocurre cuando dividimos la lista en dos mitades y continuamos con solo la mitad que nos interesa.

```
//Entrada: Array S[primero .. ultimo] y el rango(k)
//Salida: devuelve el k-ésimo elemento más pequeño en S
Quick_select(S, primero, ultimo, k)
{
    if (primero == último) // Caso base: arreglo de un elemento
        return S[primero];
    q = inPlacePartition(S, primero, ultimo);
    L_Tamaño = q - primero + 1 //Tamaño del arreglo L
    if (k <= L_Tamaño) {
        return Quick_select(S, primero, q, k); //El arreglo L
    } else
        return Quick_select(S, q+1, ultimo, k - L_Tamaño) //Arreglo E&G
}
```

$$\begin{aligned} T(n) &= T(n/2) + cn \\ &= T(n/4) + c(n/2) + cn \\ &= n(1 + 1/2 + 1/4 + \dots) \\ &= 2n \\ &= O(n) \end{aligned}$$



# *¿Preguntas?*



***FIN***